EFFICIENT REGULAR PATH QUERY EVALUATION IN PGX

Meng, X.

Award date:
2016

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Link to publication
Efficient Regular Path Query Evaluation in PGX

Author: Xuming Meng

Academic Supervisor: dr. G.H.L. Fletcher

Industrial Supervisor: dr. Sungpack Hong
  dr. Hassan Chafi
  ir. Oskar van Rest

Assessment committee: dr. G.H.L. Fletcher
  dr. H.J. Haverkort
  dr. ir. J. Vanschoren

July 31, 2016
“Make the most of yourself... for that is all there is of you.”

Ralph Waldo Emerson
TECHNISCHE UNIVERSITEIT EINDHOVEN

Abstract

Department of Mathematics and Computer Science

Master of Science

Efficient Regular Path Query Evaluation in PGX

by Xuming Meng

Data on the Internet is growing rapidly in terms of volume, velocity and variety. Such characteristics make traditional database management systems fail to fit in different user cases. Especially in the relational data model, links between data are represented as foreign keys in different tables, and hence queries involving many links are executed using expensive join operation. This approach can only work with small datasets and very restricted queries. To solve this particular issue, graph database has been proposed, where graph representation is adopted. Many research problems are accompanied with the emergence. The fundamental one is graph query language. In this thesis, we will focus on a special query, namely regular path query (RPQ), that identifies pairs of vertices connected via paths conforming to a specified regular expression over edge labels. Since this work is done in PGX that is a parallel in-memory graph analytic framework, we identify three requirements: space requirement, performance requirement, and commitment to deliver result. To efficient process RPQ while satisfying all requirements, we identify single-source transitive closure as desired algorithms and make them applicable to labeled digraphs by introducing a graph summary structure called reachability graph. Two particular implementations of single-source transitive closure are used interchangeably according to different types of reachability graph. Furthermore, an empirical study of min-hash sketch is conducted for transitive closure size estimate that used for effective query plan generation. At the end, we conduct experiments about performance of different approaches and show the improvements and limitations.
# Contents

## Abstract

### 1 Introduction

1.1 Introduction .......................................................... 1
1.2 Requirements ......................................................... 2
1.3 Contribution .......................................................... 3
1.4 Overview ............................................................. 3

### 2 Preliminaries

2.1 Graph ................................................................. 5
2.2 Path ................................................................. 6
2.3 Regular Path Query .................................................. 6
2.4 Regular Path Query with Inverse ................................. 6
2.5 PGX ................................................................. 7

### 3 Literature Review

3.1 Path Index .......................................................... 9
3.2 Reachability Index .................................................. 10
3.3 Automata-based Approach ......................................... 11
3.4 Datalog-based Relational Database ............................... 12
3.5 Transitive Closure .................................................. 12

### 4 RPQ Operator Design

4.1 Operator Overview .................................................. 15
4.2 Space Requirement .................................................. 17
4.3 Performance Requirement ......................................... 18
4.4 Commitment to Deliver Result ................................. 19

### 5 RPQ Operator Implementation

5.1 Infrastructures in PGX .............................................. 21
5.1.1 RPQ Parser ....................................................... 21
5.1.2 Non-Kleene Star Clause Evaluation .......................... 22
5.1.3 Graph Representation ......................................... 23
5.2 Reachability Graph Size Estimator ............................. 23
5.3 Reachability Graph Construction ............................... 24
5.4 Transitive Closure Implementation ............................. 25
5.4.1 Single-source TC on Reachability Graph .................. 25
5.4.2 Multi-source TC on Reachability Graph ................... 26
5.4.3 TC without Reachability Graph ............................ 28
5.5 Query Plan Optimization ........................................ 28
5.5.1 Overview of TC Size Estimate ............................. 29
5.5.2 Min-hash Sketch ............................................... 29
5.5.3 Empirical Study ............................................... 30
5.5.4 Min-hash Sketch for TC Size Estimate ............... 32
5.5.5 Summary ........................................ 33

6 Experiment ........................................... 35
6.1 Objective .......................................... 35
6.2 Datasets ............................................ 35
6.2.1 LDBC-SNB ...................................... 35
6.2.2 LUBM ............................................ 37
6.2.3 Summary of Datasets .................................... 37
6.3 System Specification .................................. 38
6.4 Result Analysis ..................................... 38
6.4.1 LDBC ............................................. 39
6.4.2 LUBM ............................................. 40

7 Conclusion ............................................ 43
7.1 Overview ............................................. 43
7.2 Future Work .......................................... 44

Bibliography ............................................. 45

A Benchmark Queries ................................. 51
A.1 LDBC ................................................ 51
A.1.1 RPQs Designed For LDBC Dataset ..................... 51
A.1.2 LDBC Complete Result ................................ 52
A.2 LUBM ................................................ 52
A.2.1 RPQs Designed For LUBM Dataset ..................... 52
A.2.2 LUBM Complete Result ................................ 53
List of Figures

2.1 A simple social network revealing like relationship between six person. ................................. 5

5.1 Normalized Root Square Error for five min-hash algorithms $k$-min averaging, $k$-min selection, HyperLogLog, $k$-partition, bottom-$k$ with three cardinality settings - $n = 1000$ (top), $n = 10000$ (middle), $n = 100000$ (bottom) and parameter $k$ ranging from 16 to 1024 ......................................................... 31

6.1 LDBC Social Network Benchmark Data Schema ....................................... 36
List of Tables

2.1 Result of the evaluation of RPQ \((x, like^*, y)\) on the graph presented in Figure 2.1. The left column \textit{individual} presents individual person \(x\) and the right column \textit{individual set} presents the set of individuals \(y\) that he/she likes. .......................... 6

2.2 Result of the evaluation of RPQ \((x, (like^{-})^*, y)\) on the graph presented in Figure 2.1. The left column \textit{individual} presents individual person \(x\) and the right column \textit{Set of Individual like him/her} presents the set of individuals \(y\) that like him/her. .......................... 7

6.1 LDBC-SNB Scale Factor Characteristics ............................................ 36
6.2 LDBC Query Path Length ................................................................. 37
6.3 LUBM Query Path Length ................................................................. 37
6.4 Characteristics of Test Data Set ......................................................... 38
6.5 LDBC-SNB: Ratios between the performance of bitmap-based BFS with and without reachability graph (i.e. \textit{BFSbRG}, \textit{BFSb} respectively). Reachability graph construction time (\textit{RGCons}) is not included in \textit{Speedup\textsubscript{1}} but included in \textit{Speedup\textsubscript{2}}. ............ 39
6.6 LDBC-SNB: Share of reachability graph construction in total query processing. \textit{Share\textsubscript{1}} indicates the graph construction share in bitmap-based BFS solution. \textit{Share\textsubscript{2}} indicates the graph construction share in MS-BFS solution. ................... 40
6.7 LUBM: Ratios between the performance of BFSb with and without reachability graph materialization. Reachability graph construction time is not included in \textit{Speedup\textsubscript{1}} but included in \textit{Speedup\textsubscript{2}}. ................................................. 41
6.8 LUBM: Share of reachability graph construction in total query processing. \textit{Share\textsubscript{1}} indicates the graph construction share in bitmap-based BFS solution. \textit{Share\textsubscript{2}} indicates the graph construction share in MS-BFS solution. ................................. 41

A.1 LDBC-SNB query evaluation time in milliseconds. Reachability graph construction time and memory consumption .......................... 52
A.2 LUBM query evaluation time in milliseconds. Reachability graph construction time and memory consumption .......................... 54
List of Algorithms

1. RPQ evaluation using TC algorithms: RPQ................. 16
2. BFS using bitmap ........................................ 26
3. Multi-Source BFS ........................................ 27
4. TC Size Estimate with K-min ............................ 32
5. TC Size Estimate with bottom-k ......................... 33
Chapter 1

Introduction

1.1 Introduction

Nowadays, available data on the Internet grows extremely rapidly in terms of volume, velocity and variety. Efficient management of such huge amount of data has appealed to both industry and academia. However, the data model used by traditional relational database management system does not fit in all user cases, such as semantic web \([7]\), social network \([5, 49, 42]\), biological network \([39, 40]\). The relational model \([43]\) represents data as tuples, groups them into relations. Each relation has unique primary/foreign key used to indicate hidden data connections. The major drawback of this model is expensive discovery of data connection via join operation. Usually, queries involving many link discovery tend to be processed inefficiently. Another downside of relational data model is lack of explicit representation of connection between data items, resulting in limits on expressiveness of accompanied query language (i.e. SQL). To overcome these issue, a nature extension to relational data model is proposed, namely graph data model \([6]\), where data items are modeled as vertices and data connections are explicitly represented as edges between vertices. As a consequence, a new database management paradigm based on graph data model, called graph database, start to appear. In contrast with traditional relational database, the advantages of graph database lie on richer representation of underlying data, a navigational language with more powerful expressiveness. Additionally, queries regarding link discovery can be executed efficiently with explicit representation of connection in graph data model.

The emergence of graph database poses many interesting research problems: how to efficiently store and retrieve data? how to query a graph database? how to utilize the special storage structure to efficiently answer a given query? In this work, we are interested in the fundamental question - graph database querying. A comprehensive characterization of different types of graph query is presented in \([8]\), in which RPQ (i.e. regular path query) is studied as basic navigation mechanism for graph database. Regular path queries are navigational queries that check the existence of paths between vertices. The determined path has to conform to the specified path pattern using regular expression. Due to the significant increase on expressiveness, many query languages try to realize this feature, such as SPARQL, SQL, PGQL etc. As an efficient query language on RDF graph, SPARQL attains regular path query by introducing SPARQL property path \([38]\) that allows specifying regular expression-like queries over properties.
Chapter 1. Introduction

One can also use SQL [21] to specify such query using recursive view. PGQL [48], as a newly proposed graph query language, introduces path query that allows intuitive definition of recursive path pattern.

Another recent trend observed is that many applications require real-time processing of terabytes or even perabytes data that is stored on both internal and external disks in different formats. Hence, the latency of disk access becomes a serious performance issue. With the decrease in the cost of memory, a new paradigm of database management system is presented in [29], namely in-memory database. By storing all data in memory, in-memory database eliminates expensive disk access cost, dramatically improves data reading and writing, and requires no complex cache mechanism as well as data synchronization. A disk-based graph database could yield worse performance compared with relational database due to random disk access by many graph traversals. Hence, combination between graph database and in-memory database is an attractive option due to the substantial performance gain.

This work focuses on adding RPQ evaluation feature into PGX\(^1\) - an in-memory graph analytic framework that has rich built-in graph algorithms, allows flexible customization of algorithms. PGX allows fast and parallel execution of both built-in and customized graph algorithms on graphs in local memory or on Hadoop\(^2\). A concrete development context is provided as well as corresponding performance requirements.

1.2 Requirements

We have identified three specific requirements from the perspectives of space, performance and user experience based on the given context of main memory.

**Space requirement** Although there has been drastic decline on the price of memory in recent years, the ratio of cost to space capability of memory is still higher than that of disk in terms of single machine. Thus, given the context of being main memory-resident, the ideal memory utilization of RPQ evaluation should be as low as possible.

**Performance requirement** As an in-memory graph analytic framework, processing speed of PGX has always be one of its major advantages. The established solution should take advantage of memory data access yielding a fast evaluation speed. Additionally, the solution should also be capable of fully utilizing the power of modern multicore architecture giving a high data throughput.

---

\(^1\)http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics/overview/index.html

\(^2\)http://hadoop.apache.org/
1.3 Contribution

Commitment to deliver result  One important requirement is that given any context (mainly available memory), delivering result is the top priority, since it is frustrating when a query engine fails to return results. Such requirement necessitates the mechanism of trading efficiency for a low space consumption in order to deliver the result in amount of tolerable time.

1.3 Contribution

The main contributions of this work are:

- We design a main-memory RPQ operator that can efficiently answer RPQ over a large graph with limited memory.
- We apply transitive closure algorithm to RPQ processing by introducing an auxiliary data structure, called reachability graph.
- We present a specific implementation of the operator to answer RPQ and conduct a performance comparison experiment based on two graph benchmarks - LDBC\(^3\) and LUBM\(^4\).
- We show an empirical study of min-hash sketch for the purpose of generating effective query plan.
- We present an adapted min-hash sketch of bottom-k flavor for effective transitive closure size estimate on a given graph.

1.4 Overview

This chapter give a brief introduction of the major topics of this work - graph database, graph query language and regular path query. Common preliminaries and theory about those topics are presented in Chapter 2. Chapter 3 shows a comprehensive literature study regarding different evaluation methods of RPQ. Chapter 4 gives the design of the RPQ operator and explain how such design satisfy all given requirements. Chapter 5 presents a specific implementation of the designed RPQ operator and the empirical study of min-hash sketch. Then, in Chapter 6, conducted experiments and detailed result analysis are provided. Finally, conclusion and future work is indicted at the last Chapter.

\(^3\)http://ldbcouncil.org
\(^4\)http://swat.cse.lehigh.edu/projects/lubm/
Chapter 2

Preliminaries

In this Chapter, we give formal definitions for employed theoretical terms. Relevant concepts regarding graph and path are introduced in section 2.1 and 2.2 respectively. Section 2.3 and 2.4 present the definition of regular path query as well as its extension with invert edge. The last section 2.5 gives a brief introduction of PGX, the context of this work, and PGQL, a powerful navigational language.

2.1 Graph

A graph $G$ is a triple $\langle V, E, \Sigma \rangle$, where $V$ is a finite set of vertices in the graph, $E \subseteq V \times \Sigma \times V$ is a finite set of directed and labeled edges, and $\Sigma$ is a finite alphabet of edge label. Figure 2.1 presents a simple like graph where individuals are represented by vertices and like relationship is indicated by directed edges with label. For instance, the edge pointing from the vertex Jonas to the vertex Vicky indicates that a person named Jonas likes the person named Vicky. However, there is no edge pointing from Vicky to Jonas, thus the statement Vicky likes Jonas is not held.

![Figure 2.1: A simple social network revealing like relationship between six person.](image)

In this Chapter, we give formal definitions for employed theoretical terms. Relevant concepts regarding graph and path are introduced in section 2.1 and 2.2 respectively. Section 2.3 and 2.4 present the definition of regular path query as well as its extension with invert edge. The last section 2.5 gives a brief introduction of PGX, the context of this work, and PGQL, a powerful navigational language.

2.1 Graph

A graph $G$ is a triple $\langle V, E, \Sigma \rangle$, where $V$ is a finite set of vertices in the graph, $E \subseteq V \times \Sigma \times V$ is a finite set of directed and labeled edges, and $\Sigma$ is a finite alphabet of edge label. Figure 2.1 presents a simple like graph where individuals are represented by vertices and like relationship is indicated by directed edges with label. For instance, the edge pointing from the vertex Jonas to the vertex Vicky indicates that a person named Jonas likes the person named Vicky. However, there is no edge pointing from Vicky to Jonas, thus the statement Vicky likes Jonas is not held.

![Figure 2.1: A simple social network revealing like relationship between six person.](image)
2.2 Path

Given a graph \( G = (V, E, \Sigma) \), a path is a sequence of interleaving vertices and edge labels: \( \rho = v_0 \alpha_0 \nu_1 \alpha_1 \cdots \alpha_{k-1} \nu_k \alpha_k \ (k \geq 0) \), where each triple \( \nu_i, \alpha_i, \nu_{i+1} \in E \) for \( 0 \leq i \leq k - 1 \). The label of path is defined as a function over the path \( \lambda(\rho) = \alpha_0 \alpha_1 \cdots \alpha_{k-1} \). A concrete example of path from Figure 2.1 is \( \text{John} \cdot \text{like} \cdot \text{Oskar} \cdot \text{like} \cdot \text{George} \cdot \text{like} \cdot \text{Jason} \cdot \text{like} \cdot \text{Jonas} \cdot \text{like} \cdot \text{Vicky} \), where \( \cdot \) is used as delimiter for the sake of clarity. The path example contains six vertices and five edges with identical edge label. Clearly, the corresponding path label is \( \text{like} \cdot \text{like} \cdot \text{like} \cdot \text{like} \cdot \text{like} \).

2.3 Regular Path Query

Given a graph \( G = (V, E, \Sigma) \), regular path queries \([12, 18, 55]\) are queries in form of \((x \mapsto R \mapsto y)\), where \( x \) and \( y \) are variables that bound to data vertices and \( R \) is a regular expression over \( \Sigma \). The result of a regular path query is the set of all source-destination pairs \((x \mapsto y)\), each of which is connected by at least one path \( \rho \) whose path label \( \lambda(\rho) \) belongs to the regular language expressed by the regular expression \( R \). By convention, we use \( | \) for disjunction, \( \circ \) for concatenation, and Kleene star \( \star \) for recursion. A proper example of regular path query based on Figure 2.1 will be \((x \mapsto \text{like}^* \mapsto y)\) that \( x \) and \( y \) are bound to pairs of vertices connected by at least one path \( \rho \) that \( \lambda(\rho) \in \text{like}^* \), where

\[
\text{like}^* = \left\{ \underbrace{\text{like} \circ \text{like} \circ \cdots \circ \text{like}}_{\leq k} \right\}.
\]

Table 2.1 shows the execution result of the query. The left column lists individual names and corresponding entries in the right column present a set of transitively liked persons.

<table>
<thead>
<tr>
<th>Individual</th>
<th>Liked Individual Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>{John, Oskar, George, Jason, Jonas, Vicky}</td>
</tr>
<tr>
<td>Oskar</td>
<td>{Oskar, George, Jason, Jonas, Vicky}</td>
</tr>
<tr>
<td>George</td>
<td>{George, Jason, Jonas, Vicky}</td>
</tr>
<tr>
<td>Jason</td>
<td>{Jason, Jonas, Vicky}</td>
</tr>
<tr>
<td>Jonas</td>
<td>{Jonas, Vicky}</td>
</tr>
<tr>
<td>Vicky</td>
<td>{Vicky}</td>
</tr>
</tbody>
</table>

Table 2.1: Result of the evaluation of \( \text{RPQ} (x, \text{like}^*, y) \) on the graph presented in Figure 2.1. The left column individual presents individual person \( x \) and the right column individual set presents the set of individuals \( y \) that he/she likes.

2.4 Regular Path Query with Inverse

The definition of regular path query above limits the expressive power. For instance, invert edge traversal cannot be expressed in an RPQ. To support such expression, \( \Sigma \) is extended by adding symbol \( \alpha^- \) for each \( \alpha \in \Sigma \). The extended \( \Sigma \) is denoted as \( \Sigma^\pm \). We use 2RPQ \([11, 8]\) as the shorthand of regular path query with inverse. As a consequence, a 2RPQ over a graph

2.5. **PGX**

\[ G = (V, E, \Sigma^\pm) \] is a triple \((x, R^\pm, y)\), where \(R^\pm\) is a regular expression over \(\Sigma^\pm\). Above, we present a RPQ that able to find all people that one person like. Sometimes it is also interesting to find all people likes a specific person. One can employ 2RPQ to express such query as \((x, (like^-)^+, y)\). In the processing of this query, edges with like label is traversed reversely, revealing the information of being liked. The result of this 2RPQ shown in Table 2.2 is quite different from Table 2.1. Based on two resulting tables, one interesting observation is that John likes everyone but no one likes him. Clearly, 2RPQ extends the expressiveness of RPQ and hence is able to find more information.

<table>
<thead>
<tr>
<th>Individual</th>
<th>Set of Individual like him/her</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>{John}</td>
</tr>
<tr>
<td>Oskar</td>
<td>{John, Oskar}</td>
</tr>
<tr>
<td>George</td>
<td>{John, Oskar, George}</td>
</tr>
<tr>
<td>Jason</td>
<td>{John, Oskar, George, Jason}</td>
</tr>
<tr>
<td>Jonas</td>
<td>{John, Oskar, George, Jason, Jonas}</td>
</tr>
<tr>
<td>Vicky</td>
<td>{John, Oskar, George, Jason, Jonas, Vicky}</td>
</tr>
</tbody>
</table>

**Table 2.2:** Result of the evaluation of RPQ \((x, (like^-)^+, y)\) on the graph presented in Figure 2.1. The left column individual presents individual person \(x\) and the right column Set of Individual like him/her presents the set of individuals \(y\) that like him/her.

2.5 **PGX**

PGX is a fast, parallel, in-memory graph analytic framework developed by Oracle Lab, which allows user to load customized graphs into memory, run efficient algorithms and query them. PGX provides rich features such as graph pattern matching, built-in parallel graph algorithms, customization of graph algorithms, and interactive shell. It even have a powerful graph query language **PGQL** [48] that provides many familiar functionality in a SQL-like syntax, such as aggregation, order by, group by, and negation. Furthermore, PGQL provides a graphic representation of graph pattern matching easing expression. Since we are going to implement the RPQ evaluation solution inside PGX, all tested queries are expressed in PGQL syntax\(^1\). For the ease of understanding, we here introduce a bit about PGQL syntax.

\[
\begin{align*}
1 & \text{SELECT COUNT}(\ast) \\
2 & \text{WHERE } (a:"John") -[:like] ->(b)
\end{align*}
\]

**Listing 2.1:** PGQL exemplary query

The PGQL query in Listing 2.1 returns the counts of pairs of vertices \((a, b)\), where the vertex \(a\) has a label John and there is an direct edge labeled as like from \(a\) to \(b\). Vertex variables and edge variables in PGQL query are surrounded by parentheses and brackets respectively. Hyphen and arrow

\(^1\text{https://docs.oracle.com/cd/E56133_01/1.2.0/PGQL_Specification.pdf}\)
indicates the direction of the edge. Such intuitive representation of graph pattern reduces the difficulty of specifying complex queries. Additionally, SQL-like syntax makes it readily accepted by customers and developers whose are familiar with SQL.

In addition, we also present the path pattern syntax of PGQL that used to express RPQ in Listing 2.2, where new keyword \texttt{PATH} is introduced to define a path pattern. In PGQL, a RPQ comprise two type of clauses: Kleene star clause that consists of a path patter and a Kleene star and non-Kleene star clause that is same as graph pattern matching. The query in Listing 2.1 shows a non-Kleene star clause, whereas the query in Listing 2.2 demonstrates a Kleene star clause where the path pattern under Kleene star operator defines a regular language of a single edge label \textit{like}. Such that, only pairs of vertices connected by at least one path whose label belongs to the defined regular language are valid.

\begin{verbatim}
1 PATH likes := () -[: like]--> ()
2 SELECT a.name, b.name
3 WHERE (a:"John")-[:likes *]--> (b)
\end{verbatim}

Listing 2.2: RPQ example in PGQL
Chapter 3

Literature Review

In this Chapter, we review many existing solutions for regular path query evaluation and analyze their advantages and limits in the context of main memory resident. Section 3.1 discusses the state-of-art path index solution towards RPQ evaluation and indicates two major drawbacks of this approach in main memory environment. Then, we examine reachability index methods in Section 3.2 and point out the class of methods suffers from substantial number of possible subgraphs. Automata-based and datalog-based solutions are discussed in Section 3.3 and 3.4 respectively. Finally, we talk about transitive closure algorithms in Section 3.5 that are considered as the best option for our cases.

3.1 Path Index

Just like in relational database, using unique key as index can greatly improve query processing performance. In graph database, feature structures (e.g. path, tree, or special subgraph) are also used as index to facilitate graph query processing. In this section, we discuss the state-of-the-art approach of RPQ processing with $k$-path index [25]. For a detailed investigation of path index, we would refer to [47]. $K$-path index approach consists of three phases: first, building index for path length up to $k$ ($k \geq 0$) and storing it in an efficient data structure such as $B^+$ tree; second, transforming RPQ with min-max limits to a union of recursion-free expression (i.e. path), each of which is formed by repetitively concatenating expression under Kleene star; third, querying the obtained $k$-path index with each expression or sub-expression if the length of expression is larger than $k$.

Evidently, $k$-path index transforms RPQ evaluation problem to a set of path finding problems, which can be efficiently solved with mature relational database techniques. Although a great performance gain has been observed, $k$-path index is not suitable in the context of in-memory database due to following reasons:

- **Graph loading time.** Clearly, the $k$-path index building procedure takes place either in graph loading period or instantly after a graph is loaded, which is also considered as part of graph loading time. The $k$-path index construction requires exponential time $O(m^k)$, where $m$ is the number of edge and $k$ is the max path length. Although the RPQ processing is benefited from the index, such long initialization time is unacceptable for an in-memory graph database.
• **Space requirement.** K-path index can be viewed as adding edges between vertices linked by a path whose length is less than or equal to \( k \). As a consequence, the index size is very likely to be several times larger than the querying graph itself. The impractical space requirement of k-path index hinders the adaptation in the context of in-memory graph database. Additionally, storing k-path index as cache files in disk is also undesired because the performance of a in-memory database will be impacted by slow disk I/O and thus becomes unpredictable and unstable.

### 3.2 Reachability Index

A restricted form of RPQ, called *regular expression based reachability query*, has attracted attention from industry and academia. Since in RPQ only end vertex variables are bound to data vertices, given a pair of vertices \( \langle a, b \rangle \), a reachability query determines whether \( b \) is reachable from \( a \) via one path conforming to the specified regular expression. The advantage of this type of query is tractable and commonly used. In [27], a highly compacted in-memory reachability index called FERRARI is employed to answer reachability queries. The reachability index is used as an oracle that is capable of indicating reachability between two vertices in a constant time \( O(1) \). Many reachability index invented in past few decades can be classified into three categories.

- **Transitive closure compression.** This category focuses on representing discovered transitive closure with specialized compression techniques, such as interval representation [2], bit-vector compression [51].

- **Hop labeling.** Instead of materializing the entire transitive closure, \( k \)-hop index [33] and a join procedure is employed in this category. By indexing only vertices that are reachable within \( k \) hops, a smaller reachability index is generated, and detection of reachability beyond \( k \)-hop is resolved by joining multiple \( k \)-hop indices.

- **Refined online search.** Compared with previous two solutions, this category uses reachability index as heuristic mechanism to prune useless branches reducing the search space [58]. The advantage of this approach is reduction on index size and speed-up on index construction.

For more detailed survey regarding reachability index, we would refer to [34, 58].

As indicated above that reachability index can only boost processing of reachability queries on unlabeled graphs, utilizing such structure for RPQ evaluation necessitates the creation of index instance for unlabeled subgraphs induced from the querying property graph for each \( k \)-sized subset of \( \Sigma^\pm \), \( 1 \leq k \leq |\Sigma^\pm| \). This approach has a poor scalability with increase on graph size and number of labels due to numerous possible subgraphs and consequently a vast number of reachability index instances.
3.3 Automata-based Approach

In past decades, many automata-based RPQ evaluation methods have been proposed on loosely linked data structures, such as XML [45]. The essential idea of this approach is that given a RPQ \( \langle x, R, y \rangle \), first, transforming the regular expression \( R \) into an automata, where transition between states is determined by edge labels; second, marking arbitrary vertex \( x \) as start state, the automata continuously consumes the following input symbols (i.e. labels from the querying data structure), associates resulting state with encountered vertices, and eventually outputs those with end state as corresponding resulting vertices \( y \).

A recent study in [37] claims that traditional automata-based RPQ solutions make no use of basic graph statistic information such as label frequency. Instead of starting from arbitrary vertices, the end vertices of edges with infrequent labels are used as fixed points that virtually partition the querying graph into smaller subgraphs and hence reduces the search space for each vertices. However, the performance of this method heavily depends on selection of threshold of infrequency because a poor selection of rare edge cannot efficiently reduce the quantity of path that need to be searched.

One major drawback of all automata-based approach for RPQ evaluation on graph is the possible enormous space consumption required for storing states, which in the worst case could goes up to \( O(|V| \times |S|) \), where \( |V| \) is the number of vertices and \( |S| \) is the number of possible states in the automata. It simply implies that under some extreme circumstances, each vertex of the querying graph could be associated with \( |S| \) states, which is undesirable for an in-memory graph database.

A query plan optimization presented in [56] aims at solving the memory issue of storing states for general automata-based solutions. The optimization is primarily based on three observations: a) Correct choice of evaluation direction leads to less intermediate results. As a consequence, less states are generated; b) Each pair of vertices can only appear once in the final result, which implies that solutions including paths connecting discovered pairs of vertices are redundant and only result in more intermediate states. Hence, it is useful to prevent such redundant path discovery at early stage; c) Large extent of sub-path sharing in discovered solutions. Identifying such sub-path and materializing them for re-use can effectively improve performance.

As indicated in the paper, in order to filter out redundant solutions preventing re-discovery of vertex pair, caching discovered partial solutions are required. However, even with the right evaluation direction, such cache can gradually grows too large to be held in memory. Besides, though it is possible to estimate degree of sub-path sharing with basic graph statistics, it tends to be imprecise when the sub-query involving recursive parts. Such imprecision consequently affects the effectiveness of the strategy - sub-path redundancy elimination.
3.4 Datalog-based Relational Database

Datalog is a declarative logic language enhancing expressiveness of relational model. A typical Datalog program consists of several clauses, each of which comprises a head and a body and variables in head must appear in bodies. The major advantage of Datalog over standard SQL is being capable of concisely expressing recursion in a stratification form. Below we give an example of recursive Datalog (Listing 3.1) finding all values less than 10 in a directed graph $G = \langle V, E, \Sigma \rangle$, where $V = \{i|1 \leq i \leq 20, i \in \mathbb{Z}\}$, $E = \{(i - 1, \text{LessThanOne}, i)|1 \leq i \leq 20\}$, and $\Sigma = \{\text{LessThanOne}\}$.

\begin{verbatim}
1 LessThan(X,Y) :- LessThanOne(X,Y).
2 LessThan(X,Y) :- LessThanOne(X,Z), LessThan(Z,Y).
3 ?- LessThan(X,10).
\end{verbatim}

LISTING 3.1: Datalog example

Here, we consider that edges in the graph as facts. The first rule states that if $X$ is less than $Y$ by one, then there is a $\text{Less Than}$ relationship between $X$ and $Y$. The second rule determines the $\text{Less Than}$ relationship between $\langle X, Y \rangle$ if there exists an $Z$ that $\langle X, Z \rangle$ and $\langle Z, Y \rangle$ satisfy predicates $\text{LessThanOne}$ and $\text{LessThan}$ respectively. Finally, we execute the Datalog program by replace $Y$ with numerical value 10.

In [20], an efficient RPQ engine is developed by transforming RPQs into standard Datalog query or SQL query extended with iteration and recursion. The novelty of this approach lies on that provenance is leveraged to facilitate the evaluation of later stage. However, during the evaluation procedure, a large set of intermediate results will be generated and joined using expensive join operator in relational model. It is not suitable for in-memory database because of memory-inefficiency and expensive join operation at final stages.

3.5 Transitive Closure

Given the context of in-memory database, aforementioned approaches all have imperfections from the perspective of space consumption and processing time. In this work, we will focus on studying how to utilize transitive closure algorithm to solve the reachability version of RPQ over labeled digraphs. Due to the simplicity and conciseness of Datalog, exemplary query in this section are all expressed in Datalog syntax. Note that $TC$ is used as shorthand for transitive closure.

Given a digraph $G = \langle V, E, \Sigma \rangle$, the transitive closure of $G$ is all pairs of vertices $\langle X, Y \rangle$ that $Y$ is reachable from $X$. Note that, edge label is not concerned here. Below we present a Datalog program that evaluates transitive closure. The first rule states that all vertices are reachable from themselves. The second rule is a left recursive expansion rule that extends existing paths between $\langle X, Z \rangle$ in the transitive closure with one more edge $\langle Z, Y \rangle$ and yields a new tuple $\langle X, Y \rangle$. The program terminates when there is no new tuple generated.
3.5. Transitive Closure

Listing 3.2: Naive Transitive Closure

This described procedure is called *naive transitive closure* that normal results in substantial redundant computation because of repeated path discovery. Let $TC^k$ denotes the head of $k$-th execution of the second rule and is regarded as a set of tuples. To avoid this wasted computation, [31] presents an optimization called *semi-naive transitive closure* that only extends by one edge those tuples in $TC^k \setminus TC^{k-1}$ with $k \geq 1$, where $\setminus$ is set complement operator.

Both naive TC and semi-naive TC are linear TC rules that require $n$ iteration, where $n$ is the longest path length in the graph. Instead of always extending path by one edge, [31, 53] presents an optimization (i.e. *SMART TC*) that recursive concatenating identified paths in TC effectively reduces the total iteration time. We illustrate the procedure using the following Datalog query.

Listing 3.3: SMART TC

After execution of first two rules, all necessary linking information of the graph is included in $TC$. We denote that the $TC$ of the second rule as $TC^1$. Then, the following statement is held that at the beginning of $k$-th iteration of third rule, $TC^k$ contains all paths of length at most $2^k - 1$ and thus $TC^{k+1}$ includes paths of length at most $2^k$. As a consequence, the iteration time is reduced to $O(\log_2 n)$. However, the major drawback of SMART TC reported in [57] is possible tremendous space consumption due to large intermediate results in the execution of third rule.

Previous $TC$ algorithms start from all vertices concurrently and tend to make use of large amount of space, whereas for some RPQs only few vertices matter. For this type of RPQ, *single-source closure algorithm* [32] is more cost-effective in terms of space. *Single-source closure algorithm* is nothing but a single-source version of naive transitive closure as presented in the following Datalog query.

Listing 3.4: Single-Source TC

The result of this query is a set of vertices that are reachable from $X$. A transitive closure can be obtained by executing single-source closure algorithm on each vertex. The *Semi-naive* optimization is also applicable to single-source closure.

We would refer to [19] for a more detailed transitive closure survey. Many of those TC algorithms are invented and optimized for relational database

```
1 TC(X,X) :- Edge(X,_) .
2 TC(X,Y) :- TC(X,Z) , Edge(Z,Y) .
```

```
1 TC(X) :- Edge(X,_) .
2 TC(Y) :- TC(X) , Edge(X,Y) .
```

```
1 TC(X,X) :- Edge(X,_) .
2 TC(X,Y) :- TC(X,Z) , Edge(Z,Y) .
```
environment where I/O cost is the performance bottleneck. However, such environment is quite different from the context of this work - memory-resident and multicore. Therefore, many optimizations are regarded as useless in our context. In this work, we would like to study how to apply single-source TC algorithms to labeled digraph and what are the best implementations as well as its advantages and limits.
Chapter 4

RPQ Operator Design

In Chapter 1, we have shown the three specific requirements for main-memory RPQ operator: a) **space requirement**, using minimal amount of memory; b) **performance requirement**, excellent performance by taking advantage of memory access and multicore architecture; c) **commitment to deliver result**, assuring result delivery by trading performance for low space consumption if necessary. In the following sections, we first present an overview of our operator design, then discuss about how such design can satisfy each requirement. In remaining part of the thesis, the term “RPQ” is slightly abused to refer to 2RPQ as shorthand.

4.1 Operator Overview

Before starting to present and discuss about the design, we define few useful terms. A sequence of edge labels that concatenated by \( \circ \) and \( \mid \) in RPQs is also referred as **path pattern**. A RPQ can be partitioned into **clauses** by considering Kleene star of root level as splitting points. A Kleene star is of root level if and only if it is not nested to another Kleene star. For instance, the clauses of the RPQ \( (x, fatherOf \circ (knows^* \circ likes)^* \circ fatherOf^-, y) \) are \( fatherOf, (knows^* \circ likes)^* \), and \( fatherOf^- \). We classify clauses into three categories **non-Kleene star clause** that contains no Kleene star operator (e.g. \( fatherOf \)), **non-nested Kleene star clause** that contains only one Kleene star (e.g. \( knows^* \circ likes \)\(^* \)), and **nested Kleene star clause** that contains nested Kleene star (e.g. \( (knows^* \circ likes)^* \)). A **non-Kleene star clause** can also be considered as **path pattern** and the **path pattern** of a **non-nested Kleene star clause** is obtained by removing the root level Kleene star. A **nested Kleene star clause** after removing the root level Kleene star can be regarded as another RPQ and hence can be evaluated using same operator recursively. Besides, the term **reachability graph** might occur frequently below, which is essentially a special index that contains all pairs of vertices \((x, y)\) that \( y \) can be reached from \( x \) through a path conforming to a specific pattern. We would refer to Section 4.3 for a more refined explanation.

The Algorithm 1 presents the design of our RPQ operator that consumes three input parameters, the **querying property graph**, the specified **RPQ** in form of string, and the maximal available amount memory, and produces the set of all tuples \( (x, y) \), where \( y \) is reachable from \( x \) through a path conforming to the specified RPQ.
Algorithm 1: RPQ evaluation using TC algorithms: RPQ

Input: property graph $G = (V, E, \Sigma^*)$,  
regular path query $Q$,  
maximal available memory size $M$

Output: The set of all reachable tuples $(x, y)$

\begin{verbatim}
result ← \{\};
while NextClause(Q) is available do
    clause ← NextClause(Q);
    if clause is Nested Kleene star clause then
        $Q' ←$ clause without * of root level;
        $RG ←$ RPQ($G, Q', M$);
        $result' ←$ TC($RG$);
    else if clause is Non-nested Kleene star clause then
        $path ← P(clause)$;
        $size ← E(path)$;
        if $size < M$ then
            materializedRG ← MRG(path, G);
            $result' ←$ TC(materializedRG);
        else
            virtualRG ← VRG(path, G);
            $result' ←$ TC(virtualRG);
        end
    else
        $path ← P(clause)$;
        $result' ←$ MRG(path, G);
        $result ←$ join(result, result');
    end
    return $result$;
\end{verbatim}

Below we explain those newly introduced functions in Algorithm 1:

- **NextClause** - extract next clause from the given regular path query. An RPQ can be partitioned into several clauses by considering the involved Kleene star as splitting point. For instance, the RPQ $\text{like} \cdot \text{dislike}^* \cdot \text{love} \cdot \text{like}$ comprises three clauses: $\text{like}$, $\text{dislike}^*$ and $\text{love} \cdot \text{like}$, where only $\text{dislike}^*$ is called Kleene star clause.

- **P** - extract the path pattern from the given clause. If clause is Kleene star clause, the path pattern is obtained by simply removing Kleene star operator. For instance, $\text{dislike}$ is the path pattern of $\text{dislike}^*$. Otherwise, the entire clause is the path pattern.

- **E** - estimate the result size of those path patterns on the digraph $G$ using pattern matching techniques.

- **MRG** - return a handler that stores materialized result of those path patterns. MRG stands for materialized reachability graph since the materialized result of path patterns can be regarded as an “unlabeled digraph”, and the term reachability emphasizes that this graph carries
4.2 Space Requirement

reachability information of the underlying digraph with respect to the given Kleene star clause.

- **VRG** - return a handler that evaluates path patterns on-the-fly for a given vertex. **VRG** stands for virtual reachability graph that is essentially the same as **MRG** except not being materialized.

- **TC** - compute transitive closure on an unlabeled digraph, which in this particular case is the handler returned by either **MRG** or **VRG**.

- **join** - return \( \{(x, y) \mid \exists z \in V. (x, z) \in S_1 \land (z, y) \in S_2\} \), where \( V \) is the vertex set, \( S_1 \) is the RPQ result set, and \( S_2 \) is the clause result set.

Clearly, in our design, a RPQ is computed by iterating over its clauses and join obtained clause result sets with the RPQ result set till no next clause is available. Each clause is handled according to its clause category: 

- **a)** non-Kleene star clause is computed by extracting corresponding path pattern and passing it into function **MRG**. Notice that here we consider the result of **MRG** as the result of the clause instead of a “unlabeled graph”; 

- **b)** non-nested Kleene star clause, is evaluated by running **TC** on top of the resulting reachability graph of either **MRG** or **VRG**; 

- **c)** as for nested Kleene star clause, we consider the inner sub-query as an independent RPQ that can be evaluated using the operator and then apply the **TC** over the resulting set of vertex tuples (viewed as graph).

Note that the most crucial part of RPQ evaluation lies on non-nested Kleene star clause, hence we mainly concern the performance and space cost regarding this clause. In addition to that, the function **MRG** and **VRG** is essentially a pattern matching problem [8] on graph, which is another heavily studied research problem. Therefore, we will not talk too much about **MRG** and **VRG** but assume the use of simple BFS/DFS with respect to predicates. In this following sections, we will discuss about why we design the operator like this and how it satisfies the three specified requirements. We would reference to Chapter 5 for a detailed discussion regarding implementation.

### 4.2 Space Requirement

Memory has always been a double-edged sword because of the possibility of efficient data processing and the limited space capacity. Compared with traditional disk space, it is disadvantaged with respects to space capability and cost. Therefore, to prevent from system crash caused by memory overuse, the basic requirement of each individual computation procedure of any in-memory databases is low memory utilization. Specifically, since **PGX** loads large property graphs into memory, it is likely that more than 50 percent of total memory is consumed by those graphs leading to even less available memory for computation. Hence, the algorithm used to solve RPQ should be as memory-efficient as possible. Below, we roughly compare the memory consumption of all previous discussed approaches and choose the most suitable algorithm.
First of all, both \textit{k-path index} (Section 3.1) and \textit{reachability index} (Section 3.2) suffers from the space issue caused by the exponential combination between edge labels. Both of them try to build some sort of “index” for each subgraph induced from the querying graph for each possible label combination. Such approach has the property of being query-independent and is capable of answering any incoming query efficiently. However, the number of label combination can be deemed as a permutation problem that grows exponentially. Therefore, both approaches are not suitable for in-memory RPQ evaluation. Although with \textit{k-path index} it is possible to only build and store path of length at most \(k\) leading a relative smaller space requirement, it is still could be several times larger than the querying graph and result in memory overuse. \textit{Automata-based approach} (Section 3.3) seems overcome the space issue since it does not need store of index. In spite of that, the major drawback of this approach is the huge space consumption for storing \textit{states} because each vertex could be associated with many states, which implies a potential space complexity of \(O(|V| \times |S|)\). The \textit{Datalog-based approach} (Section 3.4) is simply ignored since it is developed based on relational database, which is very different from our computation model. The remaining option, \textit{transitive closure} (Section 3.5), seems to be promising in terms of memory cost, since TC algorithms generally requires no pre-computation of index and generate negligible amount of intermediate results. Furthermore, with the help of \textit{VRG} and \textit{MRG}, RPQs on labeled digraphs can also be effectively transformed to TC computation on unlabeled digraphs. Based on the observations in [57], both semi-naive and SMART algorithms are not suitable for large graph processing due to substantial memory consumption caused by intermediate results. Therefore, we decide to adopt single-source transitive closure as our main option.

### 4.3 Performance Requirement

Performance has always be the key requirement for database management system, and it should become a significant advantage of in-memory database over disk-based databases considering fast memory access. In addition, it is also important for a database management system to be empowered by multicore architecture, i.e. good scalability as the number of cores increases. In Section 4.2, we have discussed all possible approaches with respect to \textit{space requirement} and find out that TC algorithms are the most suitable one in the context of in-memory databases. However, none of TC algorithms is directly applicable to a labeled digraph. We resolve this issue by introducing two help functions \textit{VRG} and \textit{MRG}. In this Section, we will discuss how we adapt TC algorithms to labeled digraph and how to improve its performance to satisfy the \textit{performance requirement}.

\begin{Verbatim}
1 TC(X,Y) := TC(X,Z), Connection(Z,Y).
2 TC(X,X) := Connection(X, _).
\end{Verbatim}

**Listing 4.1: Naive TC**

Given a set of vertices \(V\) and a set of connections \(C \subseteq V \times V\), the naive TC algorithm is able to identify all tuples \((X, Y)\), such that \(Y\) is reachable
from $X$ via a sequence of connection $\langle a_1, a_2 \rangle, \langle a_2, a_3 \rangle, \cdots, \langle a_{k-1}, a_k \rangle$, where $k \leq |V|$ and $\langle a_{i-1}, a_i \rangle \in C, 1 < i \leq k$, as presented in Listing 4.1. Clearly, there is no concept of predicate in TC algorithm, therefore we use the term **connection** instead of **edge** because edge label can be used as predicates. For a given RPQ, more specifically a given Kleene star clause, the connection $C$ is the set of pairs of vertex $\langle X, Y \rangle$ connected by a path following the **path pattern** in Kleene star clause. The term **reachability graph** ($RG$) is used to refer to the connection $C$ indicating that this structure carries reachability information with respect to a particular Kleene star clause. There are two flavors of reachability graph: 
a) **virtual reachability graph** ($VRG$) that evaluates on-the-fly the **path pattern** on the property graph and returns the set of vertices connected to the given vertex.
b) **materialized reachability graph** ($MRG$) that materialize the result of the **path pattern** and return the destination vertex set in constant time. Evidently, using $MRG$ can obtain significant performance gain since no repeated evaluation of predicates are required on edges/paths.

In Section 4.2, semi-naive and SMART TC algorithm is regarded unsuitable from the perspective of memory cost. In [57], it is pointed out that as the number of available CPU/thread increases, both algorithms show worse scalability compared with their single-source counterpart. Since one of the desired property of the established solution is leveraging multicore architecture, thus semi-naive and SMART algorithm are rejected with respect to performance.

### 4.4 Commitment to Deliver Result

As indicated in Chapter 1, this requirement assures that results have to be return for given RPQs, which implies that it allows to run inefficient algorithms that have low space consumption in case of not enough memory. Here is where the function $E$ comes to play. Clearly, function $MRG$ can significantly improve the evaluation performance of the entire RPQ operator by returning connections in constant time, whereas the online evaluation of $VRG$ results in negative impact on final performance. However, from the perspective of space consumption, the **materialized reachability graph** of $MRG$ could be tremendous and thus causes memory issue even before evaluation. Therefore, the function $E$ estimates the size of corresponding **reachability graph** with respect to the specific **path pattern**, and then determine whether to materialize the reachability graph or not. Detailed implementation of the function $E$ is discussed in Chapter 5.
Chapter 5

RPQ Operator Implementation

In this Chapter, we focus on the implementation details of the RPQ operator. Note that the final target is to implement this operator inside PGX, therefore we either adopt or adapt existing infrastructures in PGX, such as query parser, graph representation and pattern matching operators. It is worth to mention that PGX adopts property graph model internally where each vertex and edge is assigned a unique identifier and can have multiple associated properties. We will simply ignore the property part but concentrate on the labeled digraph because of our focus on RPQ evaluation in this work. In Section 5.1, those used infrastructures are discussed in a reasonable degree. Section 5.2 and 5.3 presents the implementation of size estimate of reachability graph and its construction. Finally, we present and discuss three different version of transitive closure algorithms as well as corresponding situations.

5.1 Infrastructures in PGX

5.1.1 RPQ Parser

As we mentioned above, this work is done in PGX, more specifically for a newly proposed language PGQL [48]. In contrast with SPARQL, PGQL is a query language designed for querying property graph, providing an intuitive syntax for specifying pattern matching and path queries and supporting useful SQL-like functionality such as group by, order by, negation etc.

The expression of regular path query in PGQL is made up of two types of clause - Kleene star and non-Kleene star. A Kleene star clause comprises two parts: first, path pattern declaration started with the keyword (PATH) and followed by a rule that the head is a reference identifier and the body is a normal pattern matching in PGQL; second, embedding the reference followed by a Kleene star into query body indicating recursion. Non-Kleene star clause is expressed in the same form as the body of the rule in path declaration but is directly embedded into query body.

Below we present an example PGQL path query (i.e. Listing 5.1) identifying all pairs of researchers sharing co-authorship on a given author-publication
graph. Clearly the syntax is quite similar to SQL style yet enhanced with the ability of expressing patterns in a succinct form. The path pattern declaration starts with a keyword `PATH` and is given the reference identifier `CA` (abbreviation of co-authorship). Vertices in PGQL are represented by surrounding with parentheses and can be associated with variables, like `p1`. Edge is surrounded by brackets and edge labels are specified after colon just like `published`. Arrows are used to indicate edge directions, and it means bidirectional edge if there is no occurrence of arrow. Recursion is expressed by surrounding with slash a colon followed by a path pattern reference and a Kleene star. For instance, `-/:ca*/-` means a recursive path pattern matching for co-authorship.

\[
\begin{align*}
1 & \text{PATH CA:=() } <-[[:\text{publishBy}]-[:\text{publishBy}]->() \\
2 & \text{SELECT r1, r2, p1} \\
3 & \text{FROM AuthorPublicationNetwork} \\
4 & \text{WHERE (r1) } -/:\text{CA}*/- (r2) <-[[:\text{publishedBy}]- (p1)
\end{align*}
\]

Listing 5.1: Recursive co-authorship query in PGQL

With the brief introduction of PGQL syntax, we now define what is a clause returned by the method `NextClause` in Algorithm 1. A PGQL is segmented by Kleene star clause that covers one end vertex and the recursive path pattern. Hence, each segment is a group of vertices and edges regarded as non-Kleene star clause. There is a special clause called root clause that contains single vertex as the begin of evaluation. Taking the above query as example, a possible group of clauses are: a) `(r1)` as root clause; b) `-/:ca*/- (r2)` as Kleene star clause; c) `<-[[:publishedBy]- (p1)` as non-Kleene star clause. Although there is a query planner inside PGX that determines the right processing order and thus determines next clause need to be evaluated, here we will only focus on parsing PGQL queries, while a more detailed discussion regarding query plan can be found in Section 5.5. Therefore, now we assume that the `NextClause` returns clause from left to right.

5.1.2 Non-Kleene Star Clause Evaluation

For non-Kleene star clause that is in fact a pattern matching problem, instead of building our own solution, we use highly optimized pattern matching operators provided by PGX. Given a non-Kleene star clause, matching is conducted in BFS format over the graph following the determined order of vertices and edges. For instance, assuming that matching order of the query in Listing 5.2 is left-to-right, first all vertices in graph having a name of `John` are matched, then corresponding outgoing edges with label `friend` are identified. Finally all end vertices `b` are found. Results of each stage are materialized and stored for next stage. Parallelism is heavily exploited in each stage and matching algorithms are highly optimized for scalability.

\[
\begin{align*}
1 & \text{SELECT a, b} \\
2 & \text{FROM SocialNetwork} \\
3 & \text{WHERE (a WITH name='John') } -[:\text{friend}]- (b)
\end{align*}
\]

Listing 5.2: Pattern matching query
5.2. Reachability Graph Size Estimator

Reachability graph can be regarded as a simplified graph of the original graph yet preserves necessary reachability information with respect to a particular Kleene star clause. However, the materialization could consume huge amount of memory leading to out of memory issue in the remaining processing. For instance, the reachability graph could be the complete graph based on the querying graph, consuming $O(|V| \times |V|)$ amount of memory. Obviously, such behavior violates the requirement of result delivery. One can solve this issue by using TC algorithms over virtual reachability graph VRG. Hence, it is desirable that to decide whether to materialize the reachability graph based on an estimate.

In this work, we adopt the approach in [1] designed for selectivity estimate of XML path query, which use a data structure called path tree. A path tree is a tree representation of XML document and each vertex represents a path starting from the root vertex of the XML document. Root vertex of the path tree is the root element of the XML document and each path tree vertex has sub-vertices representing XML elements directly nested under the path represented by the vertex. Each vertex has an associated value called frequency - the time of occurrence of the path. Clearly, a path tree preserves all information required to produce a precise estimate. However, a path

---

1https://www.oracle.com/database/index.html
2https://hbase.apache.org/
3http://hadoop.apache.org/
4https://www.w3.org/TR/rdf-testcases/#ntriples
5https://www.w3.org/TeamSubmission/turtle/
tree could easily be larger than the available memory. Hence, instead of
storing the entire path tree, a tabular form of path tree is used that contains
frequency of path of length up to \( n \), where \( n \) is a parameter larger than 2.
So selectivity estimate regarding path of length less than or equal to \( n \) can
be directly answer by looking up in the table, whereas the estimate of path
of length \( m > n \) is computed by following formula:

\[
f(l_1/l_2/\cdots/l_m) = f(l_1/l_2/\cdots/l_n) \times \prod_{i=1}^{m-n} \frac{f(l_{i+1}/l_{i+2}/\cdots/l_{n+i})}{f(l_{i+1}/l_{i+2}/\cdots/l_{n+i-1})}
\]

, where \( l_i, 1 \leq i \leq m \) are tags in XML document and \( f(l_1/l_2/\cdots/l_m) \) means
the frequency of the path consisting of tags. Independence assumption is
held here that the occurrence of a tag only depends on preceding \( n \) tags.
This approach is regarded as Markov process of order \( n-1 \) that current state
only depends on preceding \( n-1 \) states. In spite of that the independence
assumption is not really true, the estimate error is still relatively small as
shown in [1].

In our case, edge labels are considered as tags in XML document and hence
paths are specified as a sequence of edge labels. Taking into account short
loading time, small memory consumption and large frequency of path of
length 1 and 2, we construct the table of path of length at most 2. To com-
pute this table, we traverse the vertex set, compute the frequency of com-
binations of each vertex’s incoming and outgoing edges, and sum up the
frequency of identical path that shares the same ordered sequence of labels.
The time complexity required to compute the table is \( O(|E|^2) \). Employing
the formula above, we are capable of estimating the frequency of a given
path, namely the size of reachability graph.

### 5.3 Reachability Graph Construction

The procedure of building a reachability graph with respect to a particu-
lar Kleene star clause is in fact to externalize a customized relationship be-
tween pairs of vertex in the graph. For instance, in Listing 5.3, users try
to discover a co-authorship network embedded in the author-publication
network. From the perspective of solving the problem, it can be viewed
as a completely independent PGQL query that performs pattern matching.
Again, considering the query in Listing 5.3, we can express the reachability
graph construction as following PGQL query:

```sql
1 SELECT r1, r2
2 FROM AuthorPublicationNetwork
3 WHERE (r1)<-[[:publishBy]]-(paper) -[:publishBy]->(r2)
```

```
LISTING 5.3: Reachability Graph Query
```

However, due to the fact that internal use of PGX query engine is currently
not supported, we have to build a simpler query engine that launch a thread
that perform a step-by-step pattern matching for each vertex in the graph.
Note that we are running on top of a multicore machine that supports 72 or
more threads. Hence, it is important to leverage such power everywhere including graph construction. The query is viewed as segments including one query edge and one query vertex, such as \(<-[:publishBy]-(paper)\) and \([-[:publishBy]->(r2)\), except the first segment that only contains the first or last query vertex, such as \((r1)\). In each thread, we perform a BFS with predicate starting from one data vertex matching the first segment. Let us assume there are \(k\) segments \(\{S_1, S_2, \ldots, S_k\}\), the result of \(i\)-level in BFS is a set of vertices, to which there is a path from the starting vertex that satisfies \(\{S_1, S_2, \ldots, S_i\}\), \(1 \leq i \leq k\) in order. Consequently, the result of \(k\)-level contains the set of vertices reachable from the starting data vertex via a valid path. We can obtain the evaluation result of the query by iterating over all valid starting data vertices. Since the reachability graph construction is equivalent to the evaluation of the query, we established the graph by employing above approach.

### 5.4 Transitive Closure Implementation

In this section, we present the choice and implementation of transitive closure algorithm. It is pointed out by [57] that *naive TC*, *semi-naive TC*, and *SMART TC* are not suitable because of large memory consumption caused by redundant result duplicates and poor scalability in multicore environment compared with their single-source counterpart. In addition, most of methods in [19] do not fit our scenarios due to emphasizing on disk-based optimization, such as I/O reduction. The Warshall [54], Warren [35] algorithms as well as its follower [3] employs a \(|V| \times |V|\) Boolean matrix to compute the entire transitive closure in \(O(|V|^3)\) using Boolean operations. Again, the \(|V| \times |V|\) Boolean matrix could easily trigger an out-of-memory exception when dealing with large sparse graphs. Compared with all solutions above, single-source TC clearly has a great advantage in memory consumption and easily to be parallelized such that no interference among threads achieving better scalability.

#### 5.4.1 Single-source TC on Reachability Graph

The single-source transitive closure described in Listing 3.4 basically starts from a given source vertex (initially visited) and traverse the graph by keep following outgoing edges of each visited vertex til no new vertex discovered. Of course, since the purpose of this traversal is to identify all reachable vertices of the source vertex, same vertex will not be visited twice. Here we assume that the reachability graph has established, each connection between a pair of vertex represent a path conforming the path pattern in the Kleene star clause. Thus, any vertex that reachable from the source vertex in the reachability graph can be reached from the source vertex in the original property graph via a path that satisfies the Kleene star clause. To realize this procedure, we employed adapted BFS algorithm [57] using *bitmap* for visited vertex checking shown in Algorithm 2.

To obtain the complete transitive closure of the reachability graph, one has to iterate over all vertices. As indicated that the operator is executed in a


**Algorithm 2: BFS using bitmap**

**Input:** Reachability Graph $G$, source vertex $s$

**Output:** A Bitmap indicating reachable vertices - $seen$

1. $seen ← [false, false, \ldots, false]$;
2. $visit ← \{s\}$;
3. $visit\_Next ← \emptyset$;
4. **while** $visit ≠ \emptyset$ **do**
   5. **for each** $v ∈ visit$ **do**
      6. **for each** $n ∈ neighbor\_v$ **do**
         7. **if** $seen[n] = false$ **then**
            8. $seen[n] ← true$;
            9. $visit\_Next ← visit\_Next ∪ n$
         **end**
      **end**
   10. **end**
   11. $visit ← visit\_Next$;
   12. $visit\_Next ← \emptyset$;

multicore machine, thus more than one BFS instances are running simultaneously increasing the throughput of the operator. The maximal memory consumption is $n × |V|$ bits, assuming there are $n$ threads and each thread maintains a bitmap of size $|V|$.

## 5.4.2 Multi-source TC on Reachability Graph

In last section, we implement single-source TC on reachability graph using a bitmap-based BFS. Since one of the major application area of graph database is social network analytic, it is worth to investigate a bit domain-specific optimization. Graphs in this area are a special type of graph, namely small-world graph [4], characterized by large clustering coefficient and small average path length. In such graphs, vertices tend to form local groups of high density and the typical distance between pairs of vertex is a logarithm function of the graph size. More specifically, the diameter of the graph (i.e. longest shortest path) grows proportionally to the logarithm of the number of vertex.

It is readily deduced that given a graph traversal, such as BFS, most vertices are visited within few hops because of small graph diameter. This inference is leveraged in [52, 30] facilitating a BFS from different aspects. A well known problem in BFS study is that random neighbor vertex visit leads to bad memory locality and hence ineffective use of cache. Clearly, one can solve this issue by always visiting vertex in order. However, without any assumption of graph type of graph traversal, the performance gain is uncertain. Bearing such inference in mind, [30] proposes to visit neighbors in order only in those intermediate BFS levels where majority of vertices are visited, while adopts optimization such as bitmap and direction-optimization [9] for other BFS levels such as beginning and end levels.
In contrast with that, [52] considers to use this inference to solve the problem of multi-source BFS - a problem of efficient execution of large amount of BFSs on the same graph. Since within few steps most vertices in the graph are visited, it is highly likely that multiple independent BFS starting from different source vertices discover same vertices in same BFS level. It is proposed that instead of launching independent BFS in each thread, grouping them into one batch and executing in one thread such that each discovered new vertex can be shared by multiple BFS, thus improving cache utility and reducing impact of memory latency.

Algorithm 3: Multi-Source BFS

\textbf{Input:} Reachability Graph $G$, \\
\hspace{1em} a set of source vertices $S$, \\
\hspace{1em} a set of BFS instances $B$

\begin{algorithmic}
\State \For {each $b_i \in B$}
\State \hspace{1em} $seen[s_i] \leftarrow 1 \ll b_i$;
\State \hspace{1em} $visit[s_i] \leftarrow 1 \ll b_i$;
\EndFor
\State \textbf{reset} $visitNext$;
\While {$visit \neq \emptyset$}
\State \hspace{1em} \For {$i = 1 \cdots N$}
\State \hspace{2em} \If {$visit[v_i] \neq B_0$}
\State \hspace{3em} \For {each $n \in \text{neighbors}[v_i]$}
\State \hspace{4em} $U \leftarrow visit[v_i] \& \sim seen[n]$;
\State \hspace{4em} \If {$U \neq B_0$}
\State \hspace{5em} $visitNext[n] \leftarrow visiNext[n] \| U$;
\State \hspace{5em} $seen[n] \leftarrow seen[n] \| U$
\EndIf
\EndFor
\EndIf
\EndFor
\State \hspace{1em} $visit \leftarrow visitNext$;
\State \hspace{1em} \textbf{reset} $visitNext$;
\EndWhile
\end{algorithmic}

The version of MS-BFS presented in Algorithm 3 is heavily optimized by bitmap and bit operations, where the set of BFSs visiting a same vertex is represented by a bitmap. Note that the set of BFSs $\{b_1, b_2, \cdots, b_{|B|}\}$ can be represented by a bitmap of length $|B|$, where $i$-th bit field corresponds to $b_i$. Each vertex in the graph has a bitmap that $i$-th bit field is true if $i$-th BFS is visiting it and is false if not. Below, we briefly explain how this algorithm works. The Line 1 to 4 initializes the bitmaps of all given source vertices by setting bits of visiting BFS to true. Then, if the $visit$ array is not empty indicating there are still unterminated BFSs, we check BFS bitmaps of all vertices sequentially (reducing random memory access). For each vertex is being visited in this level, we obtain those BFSs that visits its neighbors in next level in Line 10, $U \leftarrow visit[v_i] \& \sim seen[n]$, where $n$ is the neighbor of $v_i$. $U$ is the bitmap indicating those BFSs, $visit_v$ represents BFSs visiting current vertex $v_i$, and $seen[n]$ represents all BFSs already visited $n$. The bitwise NOT operation $\sim seen[n]$ turns bit fields of unvisited BFSs to trun and thus
the bitwise AND operation $U \leftarrow \text{visit}_i \land \sim \text{seen}[n]$ identifies valid BFSs visiting the neighbor in next level. After that, updating visitNext and seen arrays with discovered BFS bitmap using bitwise OR operation. After visit in each level, visitNext is assigned to visit for next level and being reseted.

The main performance improvements of MS-BFS lie on following points: 

a) Each thread can run $k$ BFSs concurrently without paying synchronization cost, increasing throughput by a factor $k$; 
b) Sequential visiting vertex in Line 7 greatly enhances cache utility while dealing with small-world graph; 
c) Using bitmap representation and bitwise operation effectively improve traversal speed. Practically, we can run multiple MS-BFS instances independently on a multicore system. Nevertheless, the memory consumption increases up to $m \times |V| \times |B|$ bits, where $m$ is the number of used threads, $|V|$ is the number of vertex, and $|B|$ is the number of BFSs in MS-BFS.

In addition to that, as we mentioned that MS-BFS mainly benefits from small-world graphs, for normal graph its performance actually becomes worse compared with bitmap-based BFS, verified in experiments. To solve this issue, we need to determine whether a graph is a small-world graph or not. One way to determine the property of small world is to match the average path length with the logarithm value of the graph size. However, computing the precise average path length would be too expensive in our situation. Therefore, we adopt a sampling approach that we pick up $m$ random source vertices and run bitmap-based BFSs as trial, then average the path length in those BFSs to do the match. We switch to MS-BFS if matched and keeping running bitmap-based BFS if not.

5.4.3 TC without Reachability Graph

Both Algorithm 2 and 3 are discussed in the context of given reachability graph. There are still chances that the cost of materialization of reachability graph is unaffordable. For instance, the reachability graph could be a complete graph that contains $|V| \times |V|$ edges. To alleviate the memory pressure, we delay the materialization and only compute the one-hop vertices for a given source vertex when necessary. The term “one-hop” here means a hop in the reachability graph and a path complying with the path pattern in the original property graph. In other words, instead of returning the neighbors in constant time in the Line 6 in Algorithm 2 and Line 9 in Algorithm 3, the neighbor set is computed on the fly using the same method in Section 5.3. Evidently, the major drawback is that predicates on edge are substantially re-evaluated, and it becomes worse in case of handling supernodes that have large in/out-degree.

5.5 Query Plan Optimization

So far, we have discussed about how to parse a (2)RPQ, how to construct a reachability graph, how to evaluate non-Kleene star, non-nested Kleene star and nested Kleene star clause, and how to pick up the best algorithm
given a limited amount of memory. The last missing puzzle in the operator
design is that how to determine the query execution plan of a given (2)RPQ
(i.e. functionality of NextClause method). Internally, PGX employs a dy-
namic programming strategy [44] to identify an optimal join tree formed
by clauses. There is a dynamic programming (DP) table holding all partial
plans. At the beginning, the table contains query plans for each clause.
Then, new plans are generated by extending old plans with one neighbor
clause and added to DP table till no new plan discovered. During the plan
construction, the selectivity of each plan is tracked and computed based
on corresponding result cardinality. It is easy to estimate the cardinality of
non-Kleene star clause based on basic graph statistic information, such as
frequency of property, number of edge, number of vertex. Such information
can be cheaply collected during graph loading. The cardinality estimate of
transitive closure (i.e. the result of non-nested Kleene star clause), however,
is difficult because precise estimate relies on global reachability information
that is expensive to compute. At this moment, the cardinality estimate of
nested Kleene star clause is not included. Below, we explore several state-
of-the-art techniques for TC size estimate.

5.5.1 Overview of TC Size Estimate

Considering a set $S = \{s_i | v_i \in V\}$, where $V$ is the set of vertex and $s_i$ is
the cardinality of transitive closure of vertex $v_i$. We want to estimate the
value of $T = \sum_{i=1}^{\mid V \mid} s_i$. [41] presents a vertex-based sampling algorithm that
samples repeatedly with replacement from $S$ until $S$ (the sum of samples)
is greater than $2 \times |V|$. Assuming the sample time is $m$, then the estimate
is computed based the following formula $\hat{T} = \frac{|V| \times S}{m}$. For $0 < \epsilon < 0.5,$
this algorithm can estimate $T$ within a factor of $1/\epsilon$ with a probability larger
than $1 - 2 \times \epsilon$. The factor $1/\epsilon$ implies undesired high variance. Since $s_i$ is
the cardinality of a set of vertices that reachable from $v_i$, for $1 \leq i \leq |V|$, we
can employ min-hash sketches [14] - an more stable estimate technique that
utilizes the power of order statistics to solve general set cardinality estimate
problem. Below, we present the investigation involving different flavors of
min-hash sketches and adapt to TC size estimate.

5.5.2 Min-hash Sketch

Given a multiset $S = \{e_1, e_2, \cdots, e_N\}$, we assume that there exists a hash
function $h$ that can map elements in $S$ uniformly distributed over the interval
$[0, 1]$. $\{h(e_1), h(e_2), \cdots, h(e_N)\}$ consisting of hashed value of $S$ is called
ideal multiset considered as built by randomly sampling real value $N$ times
from the interval $[0, 1]$. Without assumption on ordering and data value,
the ideal multiset contains information regarding the number of distinct el-
ements (use $n$ for shorthand) in $S$. Roughly speaking, the larger $n$ is, the
smaller the expectation of the minimum of $n$ uniformly distributed values
will be. Precisely, the expectation is $\frac{1}{n+1}$. In other words, the interval $[0, 1]$
is partitioned into $n+1$ sub-interval of equal-width by $n$ real values hashed
from the multiset $S$. This method is called $k$-min sketch, where $k$ is a param-
eter indicating trade-off between performance and accuracy.
In [15], authors discuss this problem in a context of set and refer the hash mapping as rank assignment, \( r_i = h(e_i) \) for \( 1 \leq i \leq N \), such that the probability of an element \( x \in S \) has the minimum rank is \( \frac{1}{n} \). There is a difference of 1 between the probability here and the expectation in last paragraph. The reason is the difference between rank assignment and hash function. In the ideal multiset, it is assumed that \( n \) real values are uniformly distributed over the interval \( [0, 1] \), hence there are \( n + 1 \) sub-interval. However, the rank assignment is actually assigning integer rank to set elements first and then scale down to the interval \( [0, 1] \) by dividing by \( n \), thus there are only \( n \) sub-interval (the maximum rank is always \( n \)). Given \( m \) rank assignments, one can obtain a set of \( m \) minimum ranks \( \{r^1, r^2, \ldots, r^m\} \). Two estimators are built based on those sample ranks:

1. Averaging the \( m \) samples - \( m/\sum_{1 \leq i \leq m} r^i \).

2. Selecting a representative from the samples. Assume that \( \hat{r} \) the \( [n \times (1 - 1/e)] \)-smallest value in the \( m \) samples - \( 1/\hat{r} \).

Nevertheless, it is pointed out in [26] that the inverse of the minimum has an infinite expectation. To overcome this problem, the author proposes the bottom-k sketch [16] that instead of using the minimum of an ideal multiset, the k-th minimum is adopted resulting in following estimator \((k - 1)/\hat{r}\), where \( \hat{r} \) is the k-th minimum. Note that bottom-k sketch is extracted from one rank assignment and hence requires no \( k \) times of experiments. Besides, repeating experiment \( k \) times with \( k \) different hash functions is a time-consuming work. To avoid this issue, stochastic averaging process is employed in [23], also known as k-partition sketch. The core idea is to simulate the effect of \( k \) independent experiment by subdividing the interval \([0, 1]\) into \( k \) buckets and uniformly distributing elements of the multiset into those buckets, within each of which a k-min or bottom-k sketch is adopted. This approach has been proved to efficiently increase the accuracy by a factor of \( \sqrt{k} \).

### 5.5.3 Empirical Study

In this subsection, an empirical study is conducted regarding the accuracy of those randomized estimators. For the indicator of estimate accuracy, we use Normalized Root Mean Square Error (NRMSE) - \( \sqrt{|n - \hat{n}|^2/n} \), where \( n \) is the cardinality and \( \hat{n} \) is the obtained estimate. Since the content of the test set is irrelevant, a test set of cardinality \( n \) is simply regarded as \( \{1, 2, \ldots, n\} \). The three set cardinalities are 1000, 10000, 100000. Additionally, five algorithms are involved in this study: k-min averaging, k-min selection, HyperLogLog, k-partition, and bottom-k. k-min averaging is performed inside each bucket of k-partition. HyperLogLog is a state-of-art estimator based on k-partition that uses harmonic mean to replace arithmetic mean reducing the effect of outlier. In all approaches, we adopt uniform distribution as hash function. The parameter \( k \) varies from 16 to 1024 to study the effect of different \( k \) on each algorithm. At the end, we compute the average NRMSE of all algorithms based on the results of 100 runs.

Figure 5.1 shows the results of the experiment. Few interesting things are observed: 1) when \( k \ll n \), NRMSE of five algorithms gradually reduce as
$k$ increases and asymptotically become close to each other. Especially, the difference interval of $NRMSE$ is smaller than 0.05 when $k \geq 128$. 2) when $n$ is close to $k$, large estimate error appears in $k$-partition-based estimators,
i.e. HyperLogLog and k-partition, as shown at the top of Figure 5.1. The reason is that not enough elements are placed in each bucket when the number of buckets is close to the number of elements. In such situation, many buckets only contains few or even zero elements leading to unrealistic estimates. Therefore, k-partition-based estimator is also called biased estimator [24] that requires correction mechanism. 3) The curves of k-min-based estimators, i.e. k-min selection and k-min averaging, indeed verify the upper bound of relative error, $1/\sqrt{k}$, presented in [13].

5.5.4 Min-hash Sketch for TC Size Estimate

Bearing the study above, we would say that k-partition-based estimator is not a good option for transitive closure size estimate. The reason is that it is hard to determine a suitable value $k$ because of potential estimate error caused by small reachable vertex sets (TC set for shorthand). A k-min-based TC estimator is presented in [15] that given a random rank assignment of all vertices, one can identify the minimum rank in each TC set of each vertex in $O(|E|)$ using Algorithm 4.

Algorithm 4: TC Size Estimate with K-min

Input: Digraph $G = \{V, E\}$,
A vertex list sorted according to a given rank assignment $\hat{V} = \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_{|V|}\}, \hat{v}_i \in V$

1 while $\hat{V} \neq \emptyset$ do
2 \hspace{1em} $v \leftarrow$ first element in $\hat{V}$;
3 \hspace{1em} Identify $\hat{V}_i \subset \hat{V}$ including all vertices can reach $v$;
4 \hspace{1em} for each $\hat{v} \in \hat{V}_i$ do
5 \hspace{2em} Mark the minimum as $v$;
6 \hspace{1em} end
7 \hspace{1em} $\hat{V} \leftarrow \hat{V} \setminus \hat{V}_i$;
8 end

The basic idea of Algorithm 4 is that 1) taking the vertex with minimum rank as the source vertex $v$ in unvisited vertex set $\hat{V}$; 2) then traverse the graph reversely marking the minimum rank of all visited vertex $\hat{V}_i$ as $v$, as shown in Line 3 to 6; 3) remove the set of visited vertices in this iteration; 4) terminate if the unvisited vertex set $\hat{V}$ is empty; go to 1) if not. Clearly, Algorithm 4 has to be executed $k$ times in order to obtain $k$ sample values and produce an estimate for the cardinality of the set of reachable vertices. Then, by summing up those estimates, we obtain the final estimate of TC size $\hat{T}$. The statement is held that $E(\frac{T-\hat{T}}{T}) = O(1/\sqrt{k})$, where $T$ is the real size and $\hat{T}$ is the estimate, proved in [15]. Additionally, one of our concerns is the time complexity of the estimate algorithm since it is employed in query plan stage that should be as fast as possible. Let us do a bit analysis. The sorted vertex list (random permutation) can easily be generated by Fisher–Yates shuffle in $O(|V|)$ and Algorithm 4 has a time

\[6\text{https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle}\]
complexity of $O(|E|)$. Hence, the entire time complexity of TC estimate is $O(k \times (|V| + |E|))$.

However, with bottom-k estimator, we can avoid to generate $k$ rank assignment such that turn the time complexity to $O(|V| + k \times |E|)$. We present the adapted bottom-k-based TC size estimate in Algorithm 5.

**Algorithm 5: TC Size Estimate with bottom-k**

**Input:** Digraph $G = \{V, E\}$,
A vertex list sorted according to a given rank assignment $\hat{V} = \hat{v}_1, \hat{v}_2, \cdots, \hat{v}_{|V|}, \hat{v}_i \in V$,
parameter $k$

1. Let count be an array of integer initialized as zero;
2. $i \leftarrow 0$;
3. while $i \leq |V|$ do
   4. $v \leftarrow \hat{v}_i$;
   5. Identify $V_i = \{v_i | v_i \in V \cap \text{count}[v_i] < k\}$ including all vertices that can reach $v$ and have been updated less than $k$ times;
   6. for each $v' \in V_i$ do
      7. Update the k-th minimal rank of $v'$ to $v$;
      8. $\text{count}[v'] \leftarrow \text{count}[v'] + 1$;
   9. end
10. end

Below, we prove the correctness of Algorithm 5. Suppose that the list of reachable vertices of vertex $v_i$ is represented as $\hat{V}^i = v^i_1, v^i_2, \cdots, v^i_m$, for $m$ is the size of the list, and is sorted according the given rank assignment. Clearly, $\hat{V}^i$ is a sub-list of $\hat{V}$ and preserves the same order of vertex. Since the start vertex of each iteration ($v$ in Line 4) is strictly picked up according to the order in $\hat{V}$, $v_i$ is strictly visited by vertices of $\hat{V}^i$ in order. Therefore, the k-th minimal rank of $v_i$ is obtained if it is updated $k$ times. Then, by summing up the result of naive bottom-k estimator ($(k-1)/r^k$, $r^k$ is the k-th minimum rank) for each vertex, we obtain the estimate $\hat{T}$ for TC size.

### 5.5.5 Summary

In this Section, we explore min-hash sketch as a means to estimate the result set of non-nested Kleene star clause. Indeed, it gives a precise estimate with little variance compared with sampling approach in [41]. Thus, by using such technique, the optimizer is able to generate effective query plan for RPQs with no nested Kleene star clause. However, one drawback of min-hash sketch for TC estimate is that the underlying graph has to be materialized in order to compute the estimate. When dealing with nested Kleene star clause, to fully compute the reachability graph of its inner RPQ is impractical in query plan stage. Since nested Kleene star clause occurs seldom, we resolve this issue by always delaying its computation to the end of evaluation.
Chapter 6

Experiment

6.1 Objective

It is obvious that the most expensive part of (2)RPQ processing lies on Kleene star clause evaluation, to which we provide two particular implementations of TC that can work with or without reachability graph in 5 for a specific context. Through the conducted experiments, we would like to verify following statements:

- The effectiveness of materializing reachability graph. It is expected that the performance should be improved by order of magnitudes after materialization of a reachability graph.

- The memory consumption of reachability graph. Since reachability graph only captures topology information regardless of properties, the memory consumption should be negligible compared with the property graph.

- The performance impact of time required for reachability graph construction. The ratio of reachability graph over the total evaluation time could indicate the performance bottleneck.

- The impact of reachability graph type (i.e. small-world or not). Clearly, MS-BFS should benefit from small-world network, whereas bitmap-based BFS should be faster when processing non-small-world graphs.

- The impact of path length. The longer the path length is, the more complex the path pattern is, leading to higher evaluation cost.

6.2 Datasets

Two synthetic datasets are engaged in the experiments - the Lehigh University Benchmark (LUBM) dataset [28] and the Linked Data Benchmark Council (LDBC) dataset [22].

6.2.1 LDBC-SNB

The LDBC Social Network Benchmark (LDBC-SNB) aim at to become a fair benchmark specialized for graph-related techniques, such that researchers can use it for testing invented algorithms against the state-of-art algorithms,
and companies can use it to identify system's performance bottleneck. Basically, LDBC-SNB simulates a realistic social network generating people information, such as name, gender, email, birth day etc, realistic social connections (i.e. friendship, co-worker, schoolmate), and human interaction (i.e. post-reply relationship) over a tunable period of time. Notably, both social connections and human interactions conforms to the long tail effect in order to be realistic. LDBC-SNB has a tunable workload parameter called scale factors summarized in table 6.1. Taking the scale factor 1 as an example, the generated synthetic dataset contains 11,000 persons' information, their social connections and interaction over a period of 3 years starting from 2010. Additionally, LDBC-SNB is a data schema-fixed data generator that all entities, relationships and interactions are generated according to the schema in Figure 6.1.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>1</th>
<th>3</th>
<th>10</th>
<th>30</th>
<th>100</th>
<th>300</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Person</td>
<td>11K</td>
<td>27K</td>
<td>73K</td>
<td>182K</td>
<td>499K</td>
<td>1.25M</td>
<td>3.6M</td>
</tr>
<tr>
<td>Number of Years</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**TABLE 6.1: LDBC-SNB Scale Factor Characteristics**

![Diagram](image)

**Figure 6.1: LDBC Social Network Benchmark Data Schema**

Clearly, there are 15 distinct relationships engaged that we considered as edge labels. Note that we ignore all properties on vertice and edges because the purpose of the experiments is to illustrate the performance of RPQ over edge labels. Due to the fact that LDBC-SNB does not yet support generating queries involving regular path query, based on the provided data schema, we design eight RPQs that the length of path pattern ranges...
from 1 to 4 and different graph types are engaged. The focus of query design lies on identifying circular relationships formed by one or more edge labels in the schema. For instance, person-know-person, person-forum creator-forum member etc. In addition to that, no predicates over vertex properties and edge properties are engaged since they are not interested in this experiment. Those RPQs are presented in Appendix A.1.1, and we characterized them by path pattern length in Table 6.2 respectively.

<table>
<thead>
<tr>
<th>Path Length</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.2: LDBC Query Path Length

6.2.2 LUBM

The fast advance of Semantic Web Knowledge based System poses a urgent need for a benchmark that allows fair comparison between different systems and facilitate the development systematically. With such strong need, Lehigh University Benchmark (LUBM) is developed in [28] that is able to benchmark various systems, such as RDF-based, partial OWL Lite-based, complete OWL Lite-based. Like LDBC, it also has a fixed schema describing a school domain, on which data instance is generated based. The data ontology is expressed in OWL Lite (a sublanguage of OWL[1]). Fourteen test queries are designed according to several basic metrics: selectivity, complexity, input size etc. Since none of them contains a Kleene star clause, again we design 8 RPQs following same design focus in LDBC queries, based on the LUBM data schema. The test queries are presented in Appendix A.2.1. Table 6.3 illustrates the characteristics of all LUBM queries in terms of path length. In this paper, we use a scale factor of 1k that means 1000 synthetic universities are involved in the experiment.

<table>
<thead>
<tr>
<th>Path Length</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.3: LUBM Query Path Length

6.2.3 Summary of Datasets

In this section, we summarize the final data set generated from two benchmarks. We use scale factor of 1 and 1k for LDBC-SNB and LUBM respectively. The characteristics of resulting data sets are presented in table 6.4.

[1]https://www.w3.org/OWL/
It is worth to mention that both data sets are generated and stored in form of RDF. However, current version of PGX cannot directly load PDF and convert it properly into internal property graph model. The way PGX use is that loading RDF data set into Oracle database and then using RDF loader for Oracle database to load the graph into memory. Because in RDF format, edges with properties are presented as entities, leading to increase on the number of vertex and edge without proper conversion. The same thing happens to LUBM data set. Therefore, few edges in both data sets are actually represented in forms of two edges. Taking the second LDBC query as example, shown below. Clearly, the query involving knows connection between people are expressed by concatenating two edges with label knows and hasPerson. Despite that, we claim that from the perspective of performance comparison, such queries are still valid to present the difference and even increase the variety of query sets.

```
1 PATH knows := () - [:‘knows’] -> () - [:‘hasPerson’] -> ()
2 SELECT COUNT(*)
3 WHERE () -> knows* / -> ()
```

Listing 6.1: LDBC Query 2

### 6.3 System Specification

All experiments are conducted on compute cluster infrastructure at Oracle Labs. The main cluster consists of 64 nodes that are X3-2 servers – formerly known as x4170 M3, equipped with two Intel "Sandy Bridge" Xeon E5-2660 @ 2.20GHz 8-core processors, 256 GB DDR3-1600 memory, LSI MegaRaid 9261-8i RAID adapters and Mellanox Connect-IB Infiniband cards. Each experiment is submitted as a task and assigned to a single node that will only be released after the assigned task is finished. The operating system is Oracle Linux Server release 6.5.

### 6.4 Result Analysis

In this section, we demonstrate obtained experiment results and elaborate experiment setup in more detail. For both LDBC-SNB and LUBM dataset, we conduct a set of tests to reveal the effectiveness of implemented transitive closure algorithms - bitmap-based BFS and MS-BFS. For TC without reachability graph, we only test the version using bitmap-based BFS. Also, time for reachability graph construction as well as memory consumption...
is recorded, on which a fine-grained performance analysis could be performed. The complete results of LDBC and LUBM performance test is presented in Table A.1 and A.2.

6.4.1 LDBC

In Table 6.5 presents the performance of bitmap-based bfs with and without materialized reachability graph as well as the materialization time. Two indicators Speedup$^1$ and Speedup$^2$ are computed based on those time measured in milliseconds. The difference between two metrics is that Speedup$^2$ taking into account the time for reachability graph materialization. We use RG as the shorthand for reachability graph. Practically, same path pattern frequently appear in a sequence query workload and hence it make sense to cache the materialized reachability graph, which makes Speedup$^1$ a suitable performance indicator.

According to the indicator Speedup$^1$, the first thing observed is that materializing RG brings a huge performance improvement that could lead to approximately 1000 times faster processing. The speedup for query 6 simply cannot be computed because the one without materialized RG cannot finish within 11 hours and hence was forcefully stopped. As shown in Table 6.2, the path length is increased from query 1 to 8. Here, we regard path length as the indicator of query complexity, hence a rational inference is that there is a positive correlation between query complexity and performance improvement of materializing RG. Same trend is observed in Speedup$^2$ where RG construction time is considered, confirming the inference of existence of correlation. By comparing two indicator Speedup$^1$ and Speedup$^2$, we do see that materializing RG every time could potentially harm performance. On average a 50% drop on speedup is noticed (see query 1,2,5,7, 8).

<table>
<thead>
<tr>
<th>Query ID</th>
<th>BFSb (ms)</th>
<th>BFSbRG (ms)</th>
<th>RGCons (ms)</th>
<th>Speedup$^1$</th>
<th>Speedup$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>238,380</td>
<td>2,428</td>
<td>2,773</td>
<td>89x</td>
<td>45x</td>
</tr>
<tr>
<td>2</td>
<td>275,000</td>
<td>497</td>
<td>830</td>
<td>553x</td>
<td>207x</td>
</tr>
<tr>
<td>3</td>
<td>843,000</td>
<td>1,727</td>
<td>382</td>
<td>488x</td>
<td>399x</td>
</tr>
<tr>
<td>4</td>
<td>717,700</td>
<td>1,054</td>
<td>331</td>
<td>680x</td>
<td>518x</td>
</tr>
<tr>
<td>5</td>
<td>232,766</td>
<td>198</td>
<td>281</td>
<td>1175x</td>
<td>492x</td>
</tr>
<tr>
<td>6</td>
<td>&gt; 11 hours</td>
<td>408,166</td>
<td>2,619</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>232,449</td>
<td>206</td>
<td>265</td>
<td>1128x</td>
<td>493x</td>
</tr>
<tr>
<td>8</td>
<td>241,440</td>
<td>227</td>
<td>456</td>
<td>1063x</td>
<td>353x</td>
</tr>
</tbody>
</table>

Table 6.5: LDBC-SNB: Ratios between the performance of bitmap-based BFS with and without reachability graph (i.e. BFSbRG, BFSb respectively). Reachability graph construction time (RGCons) is not included in Speedup$^1$ but included in Speedup$^2$.

To have a better understanding of how RG construction affects the final performance, we compute two ratios Share$^1_1$ and Share$^2_2$ in Table 6.6. Table 6.6 shows three more values: MSBFSRG - the time of MS-BFS on RG measured in milliseconds, Share$^1_1$ - RGCons/(BFSbRG + RGCons), Share$^2_2$ - RGCons/(MSBFSRG + RGCons). Both ratios indicate the percentage of
RG construction time in the complete evaluation time. Although the ratios range from 0% to 70%, most time it is above 20% or even stable at 50%. It seems beneficial to develop a cache mechanism for RG eliminating the amount of time required for materialization. However, query 6 indicates that such phenomenon might only happens in case of relatively simple RG. From the measured time, we can say that query 6 has a relatively larger and more complex RG. The performance of executing query 6 benefits from materializing RG probably by order of magnitude as shown in Table 6.5, whereas the RG construction time is almost negligible in bitmap-based BFS solution (0.63%) and only 10% in MS-BFS solution. Thus, we would rather say that the performance impact of materializing RG is gradually diminished as the RG become increasingly complex.

<table>
<thead>
<tr>
<th>Query ID</th>
<th>BFSbRG (ms)</th>
<th>MSBFSRG (ms)</th>
<th>RGCons (ms)</th>
<th>Share(_1)</th>
<th>Share(_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,428</td>
<td>4,138,652</td>
<td>2,773</td>
<td>53.31%</td>
<td>≈ 0%</td>
</tr>
<tr>
<td>2</td>
<td>497</td>
<td>241</td>
<td>830</td>
<td>62.54%</td>
<td>77.49%</td>
</tr>
<tr>
<td>3</td>
<td>1,727</td>
<td>289</td>
<td>382</td>
<td>18.11%</td>
<td>56.92%</td>
</tr>
<tr>
<td>4</td>
<td>1,054</td>
<td>277</td>
<td>331</td>
<td>23.90%</td>
<td>54.44%</td>
</tr>
<tr>
<td>5</td>
<td>198</td>
<td>203</td>
<td>281</td>
<td>58.66%</td>
<td>58.07%</td>
</tr>
<tr>
<td>6</td>
<td>408,166</td>
<td>21,336</td>
<td>2,619</td>
<td>0.63%</td>
<td>10.93%</td>
</tr>
<tr>
<td>7</td>
<td>206</td>
<td>208</td>
<td>265</td>
<td>56.26%</td>
<td>56.02%</td>
</tr>
<tr>
<td>8</td>
<td>227</td>
<td>274</td>
<td>456</td>
<td>66.76%</td>
<td>62.46%</td>
</tr>
</tbody>
</table>

Table 6.6: LDBC-SNB: Share of reachability graph construction in total query processing. Share\(_1\) indicates the graph construction share in bitmap-based BFS solution. Share\(_2\) indicates the graph construction share in MS-BFS solution.

It is also noticed in Table 6.6, MS-BFS approach is faster than bitmap-based BFS approach for certain queries, 6x, 4x, 19x for query 3, 4, 6 respectively, while is terribly worse in query 1. By carefully inspecting those queries, we find out that LDBC query 1 is trying to identify a network of reply relationship that essentially is a graph consisting of disconnected components. For such graphs, MS-BFS only introduce large overhead because no performance gain from sharing vertex discovery process. In contrast to query 1, RGs of query 3,4,6 representing interactions between people are likely to be small world graphs, benefiting MS-BFS. Therefore, an efficient and effective small-world graph identification is crucial for the performance.

### 6.4.2 LUBM

Accordingly, we also compute the Speedup\(_1\) and Speedup\(_2\) indicators for LUBM test, shown in Table 6.7. Again, the materialization of RG indeed improves the evaluation performance by ten times to thousands times. In LDBC test, we use path length as an indicator of query complexity and infer the existence of correlation between query complexity and performance improvement of materializing RG. Speedup\(_1\) in Table 6.7 seems to disagree on this inference. Although LUBM query 8 is of path length 4, the speedup is in fact less than query 4,5,6,7. By investigating the RGs of query 4,5,6,7 and comparing them with that of query 8, we find that the performance
improvement is also related to another factor \textit{graph connectivity}. Since connections of RG represent real paths in the loaded property graph, the more connections between pairs of vertices, the more expensive to check them in the loaded property graph. For instance, the metric $\text{BFSbRG}$ of query 3 is close to $\text{RGCons}$, indicating simple structure and less connectivity.

<table>
<thead>
<tr>
<th>Query ID</th>
<th>BFSb (ms)</th>
<th>BFSbRG (ms)</th>
<th>RGCons (ms)</th>
<th>Speedup$_1$</th>
<th>Speedup$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,226,113</td>
<td>705</td>
<td>3,171</td>
<td>3157x</td>
<td>574x</td>
</tr>
<tr>
<td>2</td>
<td>2,361,853</td>
<td>16,426</td>
<td>5,349</td>
<td>143x</td>
<td>108x</td>
</tr>
<tr>
<td>3</td>
<td>3,568,323</td>
<td>170,621</td>
<td>189,127</td>
<td>21x</td>
<td>10x</td>
</tr>
<tr>
<td>4</td>
<td>2,253,685</td>
<td>744</td>
<td>7,996</td>
<td>3029x</td>
<td>574x</td>
</tr>
<tr>
<td>5</td>
<td>2,266,147</td>
<td>673</td>
<td>782</td>
<td>3367x</td>
<td>1558x</td>
</tr>
<tr>
<td>6</td>
<td>2,262,733</td>
<td>664</td>
<td>842</td>
<td>3407x</td>
<td>1502x</td>
</tr>
<tr>
<td>7</td>
<td>2,217,803</td>
<td>653</td>
<td>870</td>
<td>3396x</td>
<td>1456x</td>
</tr>
<tr>
<td>8</td>
<td>2,310,418</td>
<td>1,618</td>
<td>2,031</td>
<td>1427x</td>
<td>633x</td>
</tr>
</tbody>
</table>

Table 6.7: LUBM: Ratios between the performance of BFSb with and without reachability graph materialization. Reachability graph construction time is not included in Speedup$_1$ but included in Speedup$_2$.

The content of Table 6.8 is similar to Table 6.6 used to show the share of RG construction time. It is observed again that the overall percentage of RG construction time is above 50% indicating a potential performance bottleneck in current implementation. The impact of graph type is also observed. For instance, query 8 try to find very complicated connections between people, potentially leading to a separated graph where MS-BFS cannot compensate the introduced overhead by performance gain. Thus, MS-BFS approach is 30 times slower than bitmap-based BFS.

<table>
<thead>
<tr>
<th>Query ID</th>
<th>BFSbRG (ms)</th>
<th>MSBFSRG (ms)</th>
<th>RGCons (ms)</th>
<th>Share$_1$</th>
<th>Share$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>705</td>
<td>491</td>
<td>3,171</td>
<td>81.81%</td>
<td>86.59%</td>
</tr>
<tr>
<td>2</td>
<td>16,426</td>
<td>735,882</td>
<td>5,349</td>
<td>24.56%</td>
<td>0.72%</td>
</tr>
<tr>
<td>3</td>
<td>170,621</td>
<td>-</td>
<td>189,127</td>
<td>52.57%</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>744</td>
<td>501</td>
<td>7,996</td>
<td>91.48%</td>
<td>94.10%</td>
</tr>
<tr>
<td>5</td>
<td>673</td>
<td>498</td>
<td>782</td>
<td>53.74%</td>
<td>61.09%</td>
</tr>
<tr>
<td>6</td>
<td>664</td>
<td>499</td>
<td>842</td>
<td>55.90%</td>
<td>62.78%</td>
</tr>
<tr>
<td>7</td>
<td>653</td>
<td>487</td>
<td>870</td>
<td>57.12%</td>
<td>64.11%</td>
</tr>
<tr>
<td>8</td>
<td>1,618</td>
<td>30,760</td>
<td>2,031</td>
<td>55.65%</td>
<td>6.19%</td>
</tr>
</tbody>
</table>

Table 6.8: LUBM: Share of reachability graph construction in total query processing. Share$_1$ indicates the graph construction share in bitmap-based BFS solution. Share$_2$ indicates the graph construction share in MS-BFS solution.
Chapter 7

Conclusion

7.1 Overview

In this thesis, we design and implement a regular path query operator that is able to evaluate any reachability-based RPQ in the context of in-memory multicore system. In order to study this problem within such context, we identify three basic requirements - space requirement, performance requirement, and commitment of result delivery.

First, we examine several existing approaches for RPQ evaluation and filter out those that could result in substantial memory consumption and hence stop the execution. At the end, transitive closure algorithm is identified as the most suitable algorithms for in-memory evaluation of RPQ. The detailed investigation regarding literature review is presented in Chapter 2.

Due to the fact that TC algorithm cannot deal with labeled graphs, we come up with the RG-based TC algorithms that delegates predicate evaluation to a lower level function and query it as a black box. Such separation of functionality allows independent optimization without interference. Furthermore, in order to satisfy the requirement - commitment to deliver result, the operator should be able to decide whether the reachability graph should be materialized or not. Such ability necessitate the existence of RG size estimator. The complete design of RPQ operator is presented in Algorithm 1.

Bearing such design in mind, we elaborates two particular implementation of single-source TC - bitmap-based BFS and MS-BFS. For the sake of ease of understanding, we also briefly discuss about how PGX parse a given query, the internal graph representation, and non-Kleene star clause evaluation. At the end of Chapter 5, to prepare for query optimization regarding conjunctive RPQs including more Kleene star clauses, we conduct an empirical study of min-hash sketch technique used for TC size estimate, which is regarded as the basis of query optimization.

In Chapter 6, two sets of experiment regarding two synthetic datasets LDBC and LUBM are performed to demonstrate the effectiveness of the implemented solutions. Performance of three solutions are recorded as well as RG construction time and memory consumption. Based on obtained results, interesting questions, like how effective the materialized RG is and what is the performance impact of RG construction time etc., are answered in the result analysis part.
Chapter 7. Conclusion

7.2 Future Work

One interesting topic in this work is to study how to solve RPQ without RG. Clearly, we provide the TC-based solution without RG to satisfy the requirement of commitment of result delivery. However, redundant re-evaluation of predicates over edges make this approach less efficient for large-scale graph processing. One possible solution, inspired by reachability index is to build a summary graph structure capable of answering reachability query for certain set of vertices, while for those are not in the set a local search of small range is enough. Such hybrid solution seems to be promising in the situation where one has to constantly do trade-off between performance and space.

Another topic could be optimization of TC algorithms. Given a graph and multiple source vertices, how to efficiently traverse the graph. The essential idea is to share discovered information inter- and intra- each run of BFS/MS-BFS. Information could be shared via synchronization for TC instances running at the same time or shared via cache for instances having a temporal order. For the later, caching every bit of discovered information is unreasonable. Therefore, how to effectively utilize cache is of interest. One idea we have is that since MS-BFS benefits from small-world graph, is it possible to construct a small-world graph out of a normal graph during evaluation via cache? If we consider the cached information as adding connections to the graph, then the information discovery can also be viewed a graph construction. The problem of how to turn a normal graph to a small-world graph by effectively adding few edges is well-studied [36, 46, 17]. Combining graph construction with processing already appeared in network research area [50] and is proved to be effective. It would be interesting to know how to adapt the approach to graph query processing.

In addition, there are many estimate techniques employed in our operator including reachability graph size estimate and transitive closure cardinality estimate. Apparently, the precise estimate are crucial to performance since the query plan is determined based on them. We have already discussed in Section 5.5.5 that cardinality estimate regarding nested Kleene star clause is very expensive in query plan stage. Therefore, an affordable estimate method in query plan stage that capable of giving precise estimate is desired. One promising direction will be generating a relatively smaller summary graph that captures enough statistical information including correlation between edges, frequency of edge labels, reachability between vertices etc. Then, applying special estimate methods to the summary graph. The small size of the summary graph could significantly reduces the time required in query plan optimization stage while still yielding effective and efficient query plan.
Bibliography


Appendix A

Benchmark Queries

A.1 LDBC

A.1.1 RPQs Designed For LDBC Dataset

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
</tr>
</thead>
</table>
| A.1 | PATH post_comment := ( ) -[:’replyOf’] -()  
2 SELECT COUNT(*)  
3 WHERE ( ) -/post_comment*/ -() |
| A.2 | PATH knows := ( ) -[:’knows’] -() -[:’hasPerson’] -()  
2 SELECT COUNT(*)  
3 WHERE ( ) -/knows*/ -() |
| A.3 | PATH people_forum_people := ( ) <-> [:’hasModerator’] -()  
2 SELECT COUNT(*)  
3 WHERE ( ) -/people_forum_people*/ -() |
| A.4 | PATH people_like_post := ( ) -[:’likes’] -() -[:’hasPost’] -()  
2 SELECT COUNT(*)  
3 WHERE ( ) -/people_like_post*/ -() |
| A.5 | PATH people_create_post := ( ) <-> [:’hasModerator’] -()  
2 SELECT COUNT(*)  
3 WHERE ( ) -/people_create_post*/ -() |
| A.6 | PATH forum_people_forum := ( ) <-> [:’hasModerator’] -()  
2 SELECT COUNT(*)  
3 WHERE ( ) -/forum_people_forum*/ -() |
Appendix A. Benchmark Queries

Listing A.6: LDBC Query 6

```sql
PATH person_forum_post := () <- [:hasModerator] - [:containerOf] - [:hasCreator] - ()
SELECT COUNT(*)
WHERE /person_forum_post
```

Listing A.7: LDBC Query 7

```sql
PATH path := () <- [:studyAt] - [:hasOrganisation] - [:isLocatedIn] - ()
SELECT COUNT(*)
WHERE /path
```

Listing A.8: LDBC Query 8

### A.1.2 LDBC Complete Result

<table>
<thead>
<tr>
<th>Query ID</th>
<th>BFSb (ms)</th>
<th>BFSbRG (ms)</th>
<th>MSBFSRG (ms)</th>
<th>RGCons (ms)</th>
<th>RGMem (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>238,380</td>
<td>2,428</td>
<td>4,138,652</td>
<td>2,773</td>
<td>114,577</td>
</tr>
<tr>
<td>2</td>
<td>275,000</td>
<td>497</td>
<td>241</td>
<td>830</td>
<td>3,183</td>
</tr>
<tr>
<td>3</td>
<td>843,000</td>
<td>1,727</td>
<td>289</td>
<td>382</td>
<td>6,305</td>
</tr>
<tr>
<td>4</td>
<td>717,700</td>
<td>1,054</td>
<td>277</td>
<td>331</td>
<td>4,183</td>
</tr>
<tr>
<td>5</td>
<td>232,766</td>
<td>198</td>
<td>203</td>
<td>281</td>
<td>604</td>
</tr>
<tr>
<td>6</td>
<td>11 hours</td>
<td>408,166</td>
<td>21,336</td>
<td>2,619</td>
<td>263,689</td>
</tr>
<tr>
<td>7</td>
<td>232,449</td>
<td>206</td>
<td>208</td>
<td>265</td>
<td>604</td>
</tr>
<tr>
<td>8</td>
<td>241,440</td>
<td>227</td>
<td>274</td>
<td>456</td>
<td>2,502</td>
</tr>
</tbody>
</table>

Table A.1: LDBC-SNB query evaluation time in milliseconds. Reachability graph construction time and memory consumption

### A.2 LUBM

#### A.2.1 RPQs Designed For LUBM Dataset

```sql
PATH master_2_doctoral := () <- [:masterDegreeFrom] - [:doctoralDegreeFrom] - ()
SELECT COUNT(*)
WHERE /master_2_doctoral*
```

Listing A.9: LUBM Query 1

```sql
PATH co_author := () <- [:publicationAuthor] - [:publicationAuthor] - ()
SELECT COUNT(*)
WHERE /co_author*
```

Listing A.10: LUBM Query 2
A.2. LUBM

1 PATH classmate := () -> ['takesCourse'] -> () <-> ['takesCourse'] ->()
2 SELECT COUNT(*)
3 WHERE () -> classmate*/ ->()

Listing A.11: LUBM Query 3

1 PATH undergraduate_2_master := () <-> ['undergraduateDegreeFrom'] -> () <-> ['mastersDegreeFrom'] -> ()
2 SELECT COUNT(*)
3 WHERE () -> undergraduate_2_master*/ ->()

Listing A.12: LUBM Query 4

1 PATH work_2_master := () <-> ['subOrganizationOf'] -> () <-> ['worksFor'] -> () <-> ['mastersDegreeFrom'] -> ()
2 SELECT COUNT(*)
3 WHERE () -> work_2_master*/ ->()

Listing A.13: LUBM Query 5

1 PATH work_2_doctoral := () <-> ['subOrganizationOf'] -> () <-> ['worksFor'] -> () <-> ['doctoralDegreeFrom'] -> ()
2 SELECT COUNT(*)
3 WHERE () -> work_2_doctoral*/ ->()

Listing A.14: LUBM Query 6

1 PATH work_2_undergraduate := () <-> ['subOrganizationOf'] -> () <-> ['undergraduateDegreeFrom'] -> ()
2 SELECT COUNT(*)
3 WHERE () -> work_2_undergraduate*/ ->()

Listing A.15: LUBM Query 7

1 PATH work_2_undergraduate := () <-> ['advisor'] -> () <-> ['worksFor'] -> () <-> ['takesCourse'] -> () <-> ['teacherOf'] -> ()
2 SELECT COUNT(*)
3 WHERE () -> work_2_undergraduate*/ ->()

Listing A.16: LUBM Query 8

A.2.2 LUBM Complete Result
## Appendix A. Benchmark Queries

<table>
<thead>
<tr>
<th>Query ID</th>
<th>BFSb (ms)</th>
<th>BFSbRG (ms)</th>
<th>MSBFSRG (ms)</th>
<th>RGCons (ms)</th>
<th>RGMem (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,226,113</td>
<td>705</td>
<td>491</td>
<td>3,171</td>
<td>8,061</td>
</tr>
<tr>
<td>2</td>
<td>2,361,853</td>
<td>16,426</td>
<td>735,882</td>
<td>5,349</td>
<td>407,642</td>
</tr>
<tr>
<td>3</td>
<td>3,568,323</td>
<td>170,621</td>
<td>-</td>
<td>189,127</td>
<td>8,880,562</td>
</tr>
<tr>
<td>4</td>
<td>2,253,685</td>
<td>744</td>
<td>501</td>
<td>7,996</td>
<td>8,059</td>
</tr>
<tr>
<td>5</td>
<td>2,266,147</td>
<td>673</td>
<td>498</td>
<td>782</td>
<td>8,012</td>
</tr>
<tr>
<td>6</td>
<td>2,262,733</td>
<td>664</td>
<td>499</td>
<td>842</td>
<td>8,014</td>
</tr>
<tr>
<td>7</td>
<td>2,217,803</td>
<td>653</td>
<td>487</td>
<td>870</td>
<td>8,019</td>
</tr>
<tr>
<td>8</td>
<td>2,310,418</td>
<td>1,618</td>
<td>30,760</td>
<td>2,031</td>
<td>145,500</td>
</tr>
</tbody>
</table>

Table A.2: LUBM query evaluation time in milliseconds. Reachability graph construction time and memory consumption.