ABSTRACT

BHATTACHARYYA, ABHISHA. Evaluating Generalized Path Queries by Integrating Algebraic Path Problem Solving with Graph Pattern Matching. (Under the direction of Dr. Kemafor Anyanwu Ogan.)

Path querying on Semantic Networks is gaining increased focus because of its broad applicability. Some graph databases offer support for variants of path queries e.g. shortest path. However, many applications have the need for the set version of various path problem i.e. finding paths between multiple source and multiple destination nodes (subject to different kinds of constraints). Further, the sets of source and destination nodes may be described declaratively as patterns, rather than given explicitly. Such queries lead to the requirement of integrating graph pattern matching with path problem solving. There are currently existing limitations in support of such queries (either inability to express some classes, incomplete results, inability to complete query evaluation unless graph patterns are extremely selective, etc).

In this thesis, we propose a framework for evaluating generalized path queries- gpqs that integrates an algebraic technique for solving path problems with SPARQL graph pattern matching. The integrated algebraic querying technique enables more scalable and efficient processing of gpqs, including the possibility of support for a broader range of path constraints. We present the approach and implementation strategy and compare performance and query expressiveness with a popular graph engine.
Evaluating Generalized Path Queries by Integrating Algebraic Path Problem Solving with Graph Pattern Matching

by
Abhisha Bhattacharyya

A thesis submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Master of Science

Computer Science

Raleigh, North Carolina
2019

APPROVED BY:

________________________________________  __________________________________________
Dr. Rada Chirkova              Dr. Nagiza Samatova

________________________________________
Dr. Kemafor Anyanwu Ogan
Chair of Advisory Committee
DEDICATION

To my parents.
BIOGRAPHY

Abhisha Bhattacharyya was born in Kolkata, India. She received her Bachelor's degree in Electrical Engineering from Techno India, Salt Lake, West Bengal University of Technology, Kolkata, India in 2011. She joined Tata Consultancy Services in Bangalore, India as a Systems Engineer and worked there for three years. She joined the masters program in Computer Science department at North Carolina State University in 2017. She started working as a summer intern with Dr. Kemafor Anyanwu Ogan in Summer 2018 and continued as a Research Assistant in Fall 2018 and Spring 2019.
ACKNOWLEDGEMENTS

I am indebted to my advisor, my committee members, my peers and my family without whom I would never have finished my thesis. First and foremost, I would like to express my sincerest gratitude to my advisor, Dr. Kemafor Anyanwu Ogan who gave me the opportunity to join her lab even before I was a student and then gave me a recommendation letter which helped me immensely in getting accepted in the graduate program at NC State. Without her continuous help, support and guidance I would not have not have gotten my masters degree. I also thank Dr. Rada Chirkova and Dr. Nagiza Samatova for serving on my thesis committee.

I would like to acknowledge the support of my past and present lab members, HyeongSik Kim, Ruth Okoilu, Shalki Shrivastava, Sidan Gao, Akanksha Mohan and Arunkumar Krishnamoorthy. I would like to specially thank HyeongSik Kim for taking time to help me out and giving me valuable knowledge which helped me immensely in my research work.

Last but not the least, I would like to extend my deepest gratitude to my family, particularly, my parents, Sujit Bhattacharyya and Kaberi Bhattacharyya and my brother, Arkajyoti Bhattacharyya for their constant support and words of encouragement which helped me tide over the difficult perids of my life and career. I would also, like to wholeheartedly thank my husband, Punnag Chatterjee, for encouraging me to get into graduate studies and supporting me immensely during the stressful periods of graduate school. Without my family I would not be where I am today.

This research was supported in part by the National Science Foundation under Grant No. IIS-1218277 and CNS-1526113.
## TABLE OF CONTENTS

**LIST OF FIGURES** .................................................................................................................. vii

**Chapter 1  INTRODUCTION** ................................................................................................. 1
  1.1 Motivation .......................................................................................................................... 3
  1.2 Observation ......................................................................................................................... 5
  1.3 Structure of the Thesis ......................................................................................................... 6

**Chapter 2  RELATED WORK** ................................................................................................ 7
  2.1 Graph Theoretic Approach ................................................................................................. 7
  2.2 Algebraic Approach ........................................................................................................... 8
    2.2.1 Relational Algebra (Database Approach) .................................................................. 8
    2.2.2 Hybrid Approach (Partially Relational + Graph Approach) ................................. 8
    2.2.3 Total Algebraic Approach (Relational Algebra* + Path Algebra) ....................... 10

**Chapter 3  BACKGROUND** .................................................................................................... 12
  3.1 Algebraic Path Problem Solving ......................................................................................... 12
  3.2 RDF .................................................................................................................................. 15
    3.2.1 RDF Data Model ........................................................................................................ 15
  3.3 Algebraic Query Evaluation of Graph Pattern Matching .................................................. 17
    3.3.1 Apache Jena ............................................................................................................... 18
  3.4 Semstorm .......................................................................................................................... 19
    3.4.1 Apache Tez ................................................................................................................... 20
    3.4.2 Loops and Cycles: ...................................................................................................... 20
  3.5 Serpent ............................................................................................................................... 20

**Chapter 4  APPROACH** ......................................................................................................... 21
  4.1 Query Expression ............................................................................................................... 22
    4.1.1 Identifying GPQ Sub-Query Components in SPARQL* Queries: ......................... 23
    4.1.2 Implementation Strategy: ......................................................................................... 23
  4.2 Query Compilation ............................................................................................................. 25
    4.2.1 Logical Query Plan Transformation: ........................................................................ 26
  4.3 Query Execution ................................................................................................................ 27
    4.3.1 Implementation Strategy ......................................................................................... 27
  4.4 Path Constraints ................................................................................................................ 29
  4.5 User Interface Updations: ............................................................................................... 29

**Chapter 5  EVALUATION** ....................................................................................................... 31
  5.1 Test setup ........................................................................................................................... 31
    5.1.1 Dataset and Queries: ................................................................................................. 31
    5.1.2 Hardware Configuration: .......................................................................................... 32
  5.2 Evaluation Results .............................................................................................................. 32
    5.2.1 Performance Evaluation: ......................................................................................... 32
    5.2.2 Completeness of results: ........................................................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.3 Expressiveness:</td>
<td>34</td>
</tr>
<tr>
<td>5.2.4 Query Compilation Time Comparison:</td>
<td>36</td>
</tr>
<tr>
<td>5.3 List of Queries</td>
<td>36</td>
</tr>
<tr>
<td>5.3.1 Small Queries</td>
<td>37</td>
</tr>
<tr>
<td>5.3.2 Large Queries</td>
<td>40</td>
</tr>
<tr>
<td>Chapter 6 CONCLUSION AND FUTURE WORK</td>
<td>45</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>46</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>49</td>
</tr>
<tr>
<td>Appendix A CODE</td>
<td>50</td>
</tr>
<tr>
<td>A.1 New Code</td>
<td>50</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>QueryPlan</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Categories of Existing Platforms</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Example graph, Path Expression and Path Sequence</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Example explaining Path Sequence</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Example showing a few steps of the SOLVE algorithm for source $s = 1$</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>RDF Data Model: Table showing example RDF triples</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>RDF Data Model: Triples represented in the form of graph with directed edges</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Example SPARQL query and its transformation after compilation by Jena's parser (a) Example query (b) Corresponding SSE</td>
<td>19</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>(a) An example path query in our implementation of the integrated platform (b) The SSE produced by Jena's parser and compiler</td>
<td>22</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Mapping of the parts of motivating example to components of gpq</td>
<td>25</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Query plan transformation from graph pattern matching query to gpq</td>
<td>26</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>The Tez DAG representing the physical plan for the query shown in Figure 4.1(a)</td>
<td>28</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Size of source and destination sets for each query</td>
<td>32</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Number of paths comparison</td>
<td>33</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Explanation of self-loops (a) Example graph (b) Comparison of the paths found by Sem-Ser vs Stardog</td>
<td>34</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Time taken per path (a) For Small Queries (b) For Large Queries</td>
<td>34</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Execution Time of constrained queries vs unconstrained queries</td>
<td>35</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Table showing comparison of the level of expressiveness of our platform with Neo4j and Stardog</td>
<td>35</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Chart showing compilation time comparison</td>
<td>36</td>
</tr>
</tbody>
</table>
Many applications have the need to find connections between entities in data sets. In graph theoretic terms, this amounts to querying for paths in graphs, between sources and destinations. For most real world applications in such scenarios, the connection sought is not usually between a single source and single destination but multiple source and destination nodes. Often the sets of sources and destinations cannot be easily given explicitly but these can rather be described declaratively in terms of patterns to be matched in graphs.

For example, to assess security risks such as for flights, security officials may want to know about relationships between \( p_1 = \text{passengers on any flights to Washington DC between June 30 and July 6 who purchased one-way tickets by cash} \), and \( p_2 = \text{countries on the CIA watchlist with at least one financial link through a bank} \). Here \( p_1 \) and \( p_2 \) are patterns describing the set of sources and destinations of interest and this declarative query also has a declarative constraint \( \text{at least one financial link through a bank} \). These sorts of inquiries also commonly occur when dealing with biological networks as well as in several non-traditional emerging applications, for examples, networking. For the latter example, suppose there is a network composed of SDN ASs (Autonomous Systems) that participate in some routing protocol. An AS controller may want to compute a domain-level path from one node to another for an application and the query may include constraints related to business relationships with potential transit domains. Another feature of path queries as demonstrated by the networking example is that, there can be constraints on paths e.g. avoid domains of type \( T \) or constraints on path disjointedness (link- or node-disjoint for specific resilience...
level) or other structural constraints. Such constraints are more expressive than the property path queries which requires a regular expression of the properties in path being searched for. In a sense, these queries are traditional path queries generalized to include graph patterns and path constraints. We refer to such queries as Generalized Path Queries - gpqs.

With the standardization of property path expressions in SPARQL, there is increased focus on graph traversal queries. However, property path queries have a fundamental difference besides their expressiveness when compared to path queries - the result of a property path expression is not the paths connecting the source and destination nodes but rather sets of source and destination nodes connected by paths that match the property path pattern. G-Core [Ang18] presents a good discussion of classes of these queries. For graph-based query engines such as Neo4j [Neo], StarDog [Sta], Allegrograph [Aas06], AnzoGraph [Anz], Virtuoso [EM09] that support the RDF model, there is varied support for path querying. Some other earlier platform such as [GA13a] [Gao18] [Gao10] [Fio12] [PZ11] [GN] has focused exclusively on the path querying.

A common thread across existing path querying evaluation strategies is that they are built on traditional graph algorithms. The challenge with graph theoretic interpretations of such queries is that the different constraints in gpqs may translate to different classes of graph problems, requiring different algorithms. For example, shortest path algorithms vs. subgraph isomorphism algorithms vs. subgraph homeomorphism, and so on. From the point of view of query processing, this is a limited approach because of the limited opportunity for decomposition and reusability. On the other hand, adopting an algebraic perspective allows problems to be interpreted in a more generalized form. This also allows for more natural integration with algebraic graph pattern query engines.

Considering such a strategy makes sense once one observes that gpqs are essentially comprised of four elements: graph pattern matching, joining/filtering of graph patterns, path computation, path filtering. There are only a few platforms that do allow the sources and destinations to be expressed in a more generalized manner (for example using a graph pattern to describe the source and destination nodes). But these usually employ a graph theoretical approach of computing the paths. Some existing platforms like [Sta] do partially interpret gpq-like queries algebraically. However, the absence of a complete algebraic query interpretation framework results on falling back on traditional graph algorithms in many situations.

Another major issue with this approach appears when trying to include semantic reasoning in the path queries. There are two ways in which semantic reasoning can be included in this approach: i. prematerialize the whole graph with reasoning and then apply path computation, or ii. integrate reasoning with path computation. The former approach tends to exponentially increase the size of the graph dataset and updates to the dataset become tricky while the latter is not a trivial task. On the other hand, with an algebraic perspective, you interpret the problem in terms of an algebraic expression of multiple operators. This also makes it much easier to integrate with existing SPARQL query engines, many of which use algebraic querying methods for graph pattern matching.
In this thesis, such an algebraic approach has been proposed to evaluate Generalized Path Queries (gpqs). Since determining the exact sources and destinations for a particular path query involves simple graph pattern matching, graph computation is not required for it. We propose to use existing pattern matching approaches, for example Union of Conjunctive Queries to compute the exact sources and destinations and then add the path computation operator at the root of the query plan. This approach works because we possess the knowledge that path computation would always be the last step of query computation, i.e., the output of the graph pattern matching would feed into the path computation operator. This thesis also describes two different implementation models for the algebraic approach: the deep approach and the lightweight approach. An example integration using existing graph pattern matching platform and path computation platform has been shown. A comparative study has also been done between the integrated platform and other platforms that can handle such generalized path queries.

1.1 Motivation

There are a lot of applications that require path query computation for multiple sources and multiple destinations and in most cases the sources and destinations will not be explicitly stated but described declaratively. One possible application of this type of path query is Flight and Airport Risk Assessment (which has been described in [Any07]). To determine potential threats to flight and airport safety, security officials would query for and investigate all high risk passengers scheduled for the flight. An example query could be find the relationships between passengers scheduled for flights to Washington DC, who purchased their tickets by cash or purchased their tickets less than a day before departure, and have links to flight training. Another example query in the same scenario would be finding connections with financial links between passengers who fall within a certain demographic on a particular flight and known groups that endorse extremism or countries that support such groups.

Another possible application is Local Threat Assessment. To determine threats to local safety, for example, to prevent an attack like the San Bernardino Shooting incident [San], the local security officials would want to regularly query and investigate high risk residents in the locality. An example query for this purpose would be finding the relationships between residents who recently purchased weapons or are frequent visitors at a shooting range, and have been in touch with any person suspected of terrorism. An important point to be noted for this example query is that the source and destination sets are the same and we are trying to find connections between the members of the set. Here, security officials may also be interested in social media links that show if the person likes or follows posts from known extremist groups.

Both the applications described above aim to retrieve paths or subgraphs connecting specific nodes where the source and destination nodes are not explicitly known but can be described
declaratively (for example, *passengers scheduled for flights to Washington DC, who purchased their tickets by cash or groups that endorse extremism*). The resulting paths are also subject to some constraints (for example, *associated with flight training or financial links or social media links to extremist groups*). Such queries can have very important applications where network analysis plays a significant role. Thus, an integrated system that can take declarative descriptions of the sources, destinations and required constraints (if any) and produce qualifying paths and connections between the sources and destinations would have a wide range of real-world applications.

There are several different ways to try to approach this problem, and the first one is a graph theoretical approach. The problems we are trying to solve are homeomorphic in nature for which graph traversal based algorithm can be developed, but such algorithms are NP hard. Also such graph traversal algorithms usually require the whole graph to be in memory to be able to produce correct and consistent results. Pure graph traversal algorithms also have a few other disadvantages. Firstly, such algorithms assume explicit definitions of the source and destination nodes. Such explicit definitions are not always available for a lot of practical problems as seen in the examples described earlier. Secondly, the problems we are dealing with are a class or set of problems where addition of a new constraint makes it a slightly different problem in the same set. Pure graph algorithms would have to approach each problem separately and have different solutions to each problem in the set. Thirdly, graph algorithms are notoriously difficult to parallelize, for example, depth-first-search algorithm is inherently sequential.

The alternative to pure graph algorithms is an algebraic interpretation of the problem where query evaluation can be represented by an expression that is composed of multiple operators in a defined order. In each of the scenarios described earlier four distinct components can be distinguished clearly.

1. Pattern matching, to interpret the sets of source and destination nodes which have been described declaratively.
2. Pattern filtering, to filter out unnecessary triples.
3. Path problem solving, to find the actual links between the set of sources and destinations.
4. Path constraint filtering, to filter out irrelevant paths according to the given conditions.

An algebraic interpretation would allow us to represent each of these components as a query operator. Thus, the whole problem translates to a query planning problem, where we determine the most optimized sequence of operators to produce the correct results in the most efficient manner.


1.2 Observation

The first component which is the graph pattern matching can be solved using any existing pattern matching platforms. For the second and third components, [Tar79b] [Tar79a] discusses the basic idea behind an algebraic technique for solving path problems, where a variety of path problems can be interpreted in terms of sum and product operations. There is also some recent work on this in [Any07] which shows the overall architecture of such an integrated system and also the query evaluation plan. Figure 1.1 shows the implementation plan proposed in [Any07]. Here the Pattern matching and Pattern Filter steps can be taken care of by any existing graph pattern matching platform. For the Path Finder and Path Filter steps we would need a path operator for the algebraic approach.

Usually in such scenarios, where different queries on a dataset differ only slightly from each other, there is a lot of overlap between the answers to those queries, many of them tending to sharing sub-expressions and sub paths. The algebraic approach makes query evaluation much more amenable to multi-query optimization since this approach allows reuse of the common sub-expressions. The most commonly occurring sub-expressions and sub-paths may be computed only once and then reused for each subsequent query as necessary.

In this thesis, we propose an algebraic query evaluation technique for gpqs that delineates the four gpqs subquery elements and their mapping to algebraic query operations so that gpqs query planning translates to composition and ordering of query operations. More specifically, the thesis presents

• a conceptual query evaluation model that integrates algebraic graph pattern matching with algebraic path problem solving.

• an implementation model that perturbs the plan for graph pattern matching query generated by a SPARQL query compiler by splicing in algebraic path querying operators to produce a gpqs query plan. Another advantage of this strategy is that current SPARQL parsers and existing graph pattern matching compiler can be adopted without modification.

• an example implementation strategy using Apache Jena's query compiler and Apache Tez' DAG for physical execution is presented.

![Query Plan](image)

**Figure 1.1 Query Plan**
• comparison of the performance and expressiveness of the integrated platform with a popular engine.

1.3 Structure of the Thesis

The thesis has been subdivided into five chapters, where the first Chapter gives a brief introduction to the topic, motivation behind the research and the contributions of the thesis.

The background of the research has been discussed in the second Chapter. This chapter contains information about the RDF data model, SPARQL querying language, graph pattern matching, path querying etc. The second chapter also introduces the open source Apache Jena [McB02] framework for RDF storage and SPARQL query processing, the Apache Tez [Tez] project which is used to build application frameworks that allows for a complex directed-acyclic-graph of tasks for processing data. Apart from the tools and models, this chapter also includes an introduction to SemStorm [Kim17b] and the PrefixSolve algorithm [GA13b]. SemStorm is a query processing and storage framework for large-scale knowledge graph. The PrefixSolve algorithm is an implementation framework that allows efficient querying of graphs on disk. The code changes involved in the thesis topic has been built on the existing code base these two projects. More explanation regarding these two platforms can be found in the second Chapter of this thesis.

The third chapter includes the literature study required for the research like existing integrated platforms and their disadvantages. The fourth chapter discusses the approach used for the integration of Semstorm’s processing followed by path query computation on the input dataset using a single query. This chapter also includes implementation details.

Chapter 5 discusses the experiments conducted, their evaluation and results. It also provides information regarding the datasets and queries we used to perform the experiments. This chapter lists the SPARQL and stardog paths queries that were run to perform the experiments. The conclusion is included in Chapter 6. The new and changed code is given in Appendix A.
[AG08][Woo12] provides a good survey of graph query languages. Figure 2.1 shows the categories that existing platforms can be divided into. Existing approaches can be divided into two main categories: Graph Theoretic Approach and Algebraic Approach. The Algebraic Approach can be further sub-divided into Relational Algebra (Database Approach), Hybrid Approach (Partially Relational + Graph Approach) and Total Algebraic Approach (Relational Algebra* + Path Algebra). Each of these categories are explained in detail in the next few sections.

2.1 Graph Theoretic Approach

Query evaluation using traditional graph traversal approach suffers from a few issues. Firstly, the problem becomes a sub-graph homeomorphism problem, which is NP hard. Also, slightly different constraints in gpqs may translate to completely different classes of problems, requiring different algorithms. For example, Dijkstra's algorithm is for shortest paths. Traditional graph traversal algorithms assume that they will be provided with explicit definitions of the source and destination nodes and have no provision for accepting declarative definitions. Additionally, the algorithms in its entirety are difficult to parallelize.
2.2 Algebraic Approach

The second method of query evaluation is an algebraic approach where query operators act as the building blocks of query evaluation and are composed by an optimizer.

2.2.1 Relational Algebra (Database Approach)

In this approach introduced by [GN] the nodes on the paths are interpreted in terms of joins. The Database approach Hence, every edge with start and end nodes is joined with the next edge with start and end nodes thus forming paths. For queries using this approach the length of the resulting path must be bound since the query needs to specify the number of joins expected in the path. If the dataset is partitioned into different tables based on any parameter, the query would need to specify the tables that have to be joined. Since the number of tables mentioned in the query must be bound, this approach cannot find paths of arbitrary length. Also, the approach is mostly focused on efficiently finding shortest path.

2.2.2 Hybrid Approach (Partially Relational + Graph Approach)

There are a few platforms that follow a hybrid approach where the pattern matching is evaluated algebraically and the path part of the problem is evaluated in the traditional graph traversal way. Platforms in this category include Neo4j [Neo], which has a very non-user friendly syntax which is declarative but not SQL-like. Yet another platform using a similar approach is Stardog [Sta], which
provides a limited range of path constraints. Similar to the previous category these platforms also focus on shortest path queries, and cannot efficiently run all-paths queries.

Neo4j [Neo], AgensGraph [Age] uses Cypher [Fra18][Cyp] as its query language. Cypher uses a fast bidirectional breadth-first search algorithm for optimizing path queries. However, this algorithm can be used only in certain scenarios. Firstly, this algorithm can only be used for finding shortest path, when finding all paths Neo4j uses a much slower exhaustive depth-first search algorithm. Even when finding shortest paths the fast algorithm is used only if the predicates in the path query can be evaluated on the fly while searching for the path. In case of predicates that need to look at the full path before deciding whether to keep or discard that particular path Neo4j's query evaluation must fall back to exhaustive searching.

Triples that have a constant as the object are considered as properties of the subject node. For example after loading the triple `acm:person-297809 akt:full-name "Wendy E. Mackay"` into Neo4j the node identified by `acm:person-297809` would have a property `full-name` with the value "Wendy E. Mackay". Now suppose we have a shortest path query with the predicate that says `ALL nodes in the path must have a full-name property`. This predicate can be evaluated on the fly while running the query and thus this query will be evaluated using the fast bidirectional breadth-first search algorithm. However, suppose the the predicate in the above query was `At least one node in the path must have a full-name property`. To evaluate this query Neo4j needs to inspect the whole path before it can decide whether the path meets the requirements. Thus, for this query exhaustive depth-first search algorithm will be used. Neo4j's Cypher also has another drawback where it's shortest path algorithm does not work when the start and end nodes are the same. Such a scenario might occur when performing a shortestPath search where the sources and destinations are overlapping sets of nodes. This error can be avoided by changing certain configuration parameters, but, in that case there might be some missing results. Also this issue does not occur when running all paths queries.

Stardog follows a more traditional SPARQL operators for query evaluation. For any path query with start and end variable patterns Stardog uses the below evaluation strategy

\[
\text{eval}(\text{PQ}(s, PS, e, PE, PQ)) = \text{Join}(PS, \text{Join}(PE, \text{eval}(\text{PQ}(s, e, PQ), DG)))
\]

where \(s\) and \(e\) are start and end variables respectively, \(PS\) and \(PE\) are the start and end graph patterns with given dataset \(D\), graph \(G\) and Path Query \(PQ\). For any path query with start and end variable patterns Stardog first finds all possible paths that match \(PQ\) which is similar to property paths. The set of paths found is then joined with the end or destination graph pattern. This in turn is joined with the start or source graph pattern. Before Stardog allowed full-fledged path queries it used property paths and hence, Stardog takes this approach towards solving path queries. This approach of applying filter first and then joining with source and destination patterns might be useful when the filter is highly restrictive. However, if the path query filter is not restrictive it will produce a large
resultset which will then need to be joined with the start and end patterns. Figure ?? shows some of
the existing platforms and the algorithms used by these platforms.

There is another hybrid approach that is followed by platforms such as Virtuoso [EM09], RDFPath
[PZ11], Blazegraph [Bla], Gremlin [Rod15] which is the query language for JanusGraph [Jan] and Neptune [Nep], Oracle's PGQL [Res16][Sev16][Ora][Pgql] which use Path expression queries. Their focus is
on patterns for path constructs and they do not have any provision for patterns for sources and destinations. These platforms require users to write a regular expression of the properties in the path and
the result of the query is the set of sources and destinations and not necessarily the paths between
them. To write a regular expression the users need to possess the knowledge of the properties in the
path and also a general ordering of the properties. \( ?a \text{ akt:has-author/akt:has-affiliation}^* ?b \).

is an example query which is looking for paths that has one "has-author" property and zero or
more "has-affiliation" properties.

Virtuoso, RDFPath and Blazegraph use property paths which is a route through a graph between
two graph nodes. SPARQL 1.1 [Pro] supports these property paths [Pro] which allows the extension
of matching triple patterns to arbitrary length paths. However, using property paths it is only
possible to know the specific sources and destination, but not the exact paths. Gremlin also requires
the predicates of the path to be specified in the query. Oracle's PGQL finds paths using general
expressions over vertices and edges of the graph. In this case, like in the case of property paths the
user needs to have the knowledge of the sequence of edges of the paths required.

2.2.3 Total Algebraic Approach (Relational Algebra* + Path Algebra)

In this approach the problem is decomposed in a way that makes it amenable for efficient evaluation.
Suppose we have the motivating example query:

Query: Find relationships between passengers on any flights to Washington DC between June 30
and July 6 who purchased one-way tickets by cash, and countries on the CIA watchlist where there
is at least one financial link through any bank.

In the total algebraic approach this above mentioned query will be decomposed into the following
components:

• two set queries called graph pattern queries,

• a connection query connecting the two sets,

• a constraint specified on the nature of the connection.

Hence, the query now becomes a constrained path query. This approach provides a few advantages
over the other approaches.

Advantages of an Algebraic Approach: Using this approach evaluation can be generalized to handle
all possible constraints in gpqs. Also, an algebraic approach allows a more natural integration
with graph pattern query engines which usually evaluate queries algebraically. Additionally, since evaluation is being broken down into multiple components some of these may be parallelizable.
3.1 Algebraic Path Problem Solving

Let us review an existing approach of algebraic path problem solving introduced by [Tar79b]. An edge \( e \) in a directed labeled graph \( G = (V, E) \) is denoted as \( e = (v_1, v_2) \) with label \( \lambda(e) = l_e \) where \( v_1, v_2 \in V \) and \( e \in E \). A path \( p \) in such a graph \( G = (V, E) \) is defined as an alternating sequence of nodes and labeled edges terminating in a node \( p = \{v_1, l_{e_1}, v_2, l_{e_2}, ..., v_n, l_{e_n}, v_{n+1}\} \) where \( v_1, v_2, ..., v_n, v_{n+1} \in V \) and \( e_1, e_2, ..., e_n \in E \). A path expression of type \((s, d)\), \( PE(s, d) \), as defined in [Tar79b], is a 3-tuple \( \langle s, d, R \rangle \), where \( R \) is a regular expression over the set of edges defined using the standard operators union(\( \cup \)), concatenation(\( \cdot \)) and closure(\( \ast \)) such that the language \( L(R) \) of \( R \) represents paths from \( s \) to \( d \) where the RDF graph contains nodes \( s \) and \( d \). Let us consider the example graph shown in Figure 3.1(a) which has been borrowed from [GA13a]. Also, \( \lambda \) and \( \phi \) denote empty string and empty set respectively. Now the path expression of type \((2, 7)\) in the example graph would be \( PE(2, 7) = \langle 2, 7, ((b \cdot c \cdot f) \cup (i \cdot f)) \rangle \)

Path expressions may be only partially complete and its regular expression may represent only a subset of all possible paths between the source and destination nodes. The example graph has several paths between nodes 2 and 7 but only two of these paths are represented in the path expression mentioned above. In the path expression string (or regular expression) the nodes are omitted for brevity and only the edges are included in the string. The path expression string is actually stored as an abstract syntax tree (AST) where the operators like union(\( \cup \)) and concatenation(\( \cdot \)) form the
If a graph is ordered using any numbering scheme, information about paths in the graph can be represented using a particular sequence of path expressions called a path sequence (PS) [Tar79b] [GA13a]. A Path Sequence (PS) is a unique ordering of path expressions that represent all path information in a graph, such that for any path \( p \), there is a unique partition of \( p \) into non empty subpaths, and a unique sequences of indices \( I \) of PS, such that the \( i \)th subpath of \( p \) is represented by the path expression in PS at the \( i \)th index in \( I \). Figure 3.1(b) shows the path sequences that represents the example graph in Figure 3.1(a). Figure 3.2 explains the property of Path Sequence that allows the algorithm that uses this construct to solve all paths problem by scanning the path sequence from left to right. Suppose path \( p \) can be divided into 5 subpaths \( p_1, p_2, p_3, p_4, p_5 \) and indices \( I = \{2, 3, 100, 390, 1000\} \). Then the paths \( p_1 \) to \( p_5 \) must be represented by the path expressions at these indices of the path sequence. However, \( p_5 \) cannot be represented by say \( PE_{155} \).

Given a path sequence, many path problems can be solved using a simple propagation SOLVE algorithm [Tar79b] [GA13a] that assembles path information as it scans the path sequence from left to right. At every iteration of the SOLVE algorithm the following step is performed

\[
PE(s, w_i) \cup (PE(s, v_i) \bullet PE(v_i, w_i)) \rightarrow SA[w_i].
\]

Here we take an existing path expression and extend it using conjunction and possibly union with other path expressions. At every iteration \( i \) the algorithm retrieves the \( i \)th path expression \( PE(v_i, w_i) \) with source \( v_i \) and destination \( w_i \) from the path sequence PS. Then, it extends the
path expression currently in $SA[v_i]$, i.e., $PE(s, v_i)$ by concatenating it with $PE(v_i, w_i)$ resulting in $PE(s, w_i)$. This represents paths from $s$ to $w_i$ via $v_i$. This is then used to extend the path expression currently in $SA[w_i]$ using union operation resulting in updated path expression $PE(s, w_i)$ which is stored in $SA[w_i]$. This step is performed over and over till the algorithm reaches the end of the path sequence and at the end of the scan, we are guaranteed completeness of the source node used to drive the propagation phase. Figure 3.3 shows a few steps of the SOLVE algorithm with source $s = 1$. Multiple path problems can be solved using the same algorithm by interpreting Union($\cup$) and concatenation($\cdot$) operators appropriately. The same algorithm can be performed for multiple sources to solve path queries with multiple sources and multiple destinations. [GA13a] shows how this SOLVE algorithm can be optimized when running for multiple sources such that common paths can be reused instead of being recomputed.

One of the main issues with graph computation is that every problem requires a different algorithm. For example Shortest path requires a certain algorithm (Dijkstra's, Floyd-Warshall's etc), all paths would require a different algorithm. Each of these algorithms solve a very specific problem. We propose to use a different approach to solve path computation. [GA13a], [Any07] and [Gao18] describe a query evaluation framework where all paths an algebraic method is followed. In this method all paths are always computed as the first step, hence, solving the general problem first and then applying the necessary filters (constraints) to narrow down the results. This method eliminates the need to have a different algorithm for each problem. To implement this type of query evaluation a representation method is required. However, the details of this representation method are not in the scope of this thesis.
Solving \( s = 1, d \): Initialize: \( PE(s, s) = \lambda \rightarrow SA[s], \) \( PE(s, d) = \emptyset \) for \( d \neq s \rightarrow SA[d] \)

Step i (iteration i): \( PE(s, w_i) \cup (PE(s,v_i) \bullet PE(v_i,w_i)) \rightarrow SA[w_i] \)

Step 1 (s=1, \( v_1 = 1, w_1 = 4 \)): \( PE(1, 4) \cup \emptyset \rightarrow SA[4] \)

......

Step 5 (s=1, \( v_5 = 4, w_5 = 5 \)): \( PE(1, 5) \cup \emptyset \rightarrow SA[5] \)

......

Step 7 (s=1, \( v_7 = 4, w_7 = 5 \)): \( PE(1, 6) \cup \emptyset \rightarrow SA[6] \)

\[ \text{Figure 3.3 Example showing a few steps of the SOLVE algorithm for source } s = 1 \]

### 3.2 RDF

The World Wide Web [Www], also known as WWW or simply the Web is a network of information where each entity, known as resources, is identified by a Uniform Resource Identifiers (URI). This unique identification of each resource facilitates referencing that resource in further building the information network larger by linking each resource to other resources using meaningful connections.

The World Wide Web Consortium (W3C) [W3c] is the international standards organization and community that develops and maintains standards for the Web. The Resource Description Framework (RDF) [Rdfb] is a W3C standard model for data on the Web. In the RDF data model two resources identified by their URIs are linked using a predicate. The next subsection discusses the RDF data model in detail.

#### 3.2.1 RDF Data Model

The RDF [Rdfb] data model structures data into triples consisting of a subject, predicate and object. Now RDF data can also be viewed as graphs where the subject and object are nodes and the predicate is a directed edge from the subject node to the object node. Both these data models are used in this thesis as the pattern matching platform uses the triple data model while the graph processing platform uses the graph data model.

Figure 3.4 shows a set of RDF triples that describe a subset of University data taken from a sample LUBM dataset [Guo05]. The table in Figure 3.4 shows data triples in black and schema
<table>
<thead>
<tr>
<th>RDF Data Triples</th>
<th>RDF Data Triples (Cont.)</th>
<th>RDF Data Triples (Cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>:prs1 :affWith :u1</td>
<td>:prs3 :affWith :u1</td>
<td>:FP1 :researchInterest &quot;Research1&quot;</td>
</tr>
<tr>
<td>:prs1 :affWith :u3</td>
<td>:prs3 :affWith :u4</td>
<td>:Dept1 :name &quot;Dept1&quot;</td>
</tr>
<tr>
<td>:prs1 :affWith :u1</td>
<td>:prs3 :affWith :u1</td>
<td>:FP1 :researchInterest &quot;Research1&quot;</td>
</tr>
<tr>
<td>:prs1 :affWith :u3</td>
<td>:prs3 :affWith :u4</td>
<td>:Dept1 :name &quot;Dept1&quot;</td>
</tr>
</tbody>
</table>

**RDF Data Triples RDF Data Model: Table showing example RDF triples**

RDF triples in bold and blue. The former capture instance data (assertions about instances of relationships or instances of class memberships). The latter describes domain concepts, relationships and the relationships between them. All the entities that start with ":" have a leading prefix such as <http://swat.cse.lehigh.edu/onto/univ-bench.owl#> which has been omitted from the table for brevity. Hence, each entity that starts with ":" is a unique identifiers or URI while each entity that is enclosed in quotes is a string literal.

The data triple (:prs1 :affWith :u1) denotes that the entity identified by :prs1 is affiliated with (:affWith) the entity :u1 which is a type of University as can be understood from the triple (:u1 rdf:type :University). The type triples are a special type of data triple and these have been marked with patterned boxes in the Figure 3.4. These rdf:type triples identify entities like :u1, :prs1 to be instances of class :University and :Person respectively using the triples (:u1 rdf:type :University) and (:prs1 rdf:type :Person). The schema triples include domain concepts definitions such as Faculty and Person and RDF Schema RDF(S) [Rdfa] metarelations such as rdfs:subclassOf which is used to assert that the class Employee "is a subclass of" the class Person. The schema triples also define the domain and range of each property, for example, :lectures and :affWith. The triple (:lectures rdfs:domain :Faculty) denotes that the property :lectures will always have entities of type (rdf:type) :Faculty as the subject, for example (:prs1 :lectures :crs1) where (:prs1 rdf:type :Faculty). Similarly, the
Figure 3.5 RDF Data Model: Triples represented in the form of graph with directed edges

triple (?:affWith rdfs:range :University) denotes that the property :affWith will always have entities of type :University as the object, for example (:prs1 :affWith :u1) where (:u1 rdf:type :University).

Figure 3.5 shows the part of the RDF data shown in Figure 3.4 represented as a graph where the subjects and objects like :prs1 and :u1 respectively are nodes while the predicates like :affWith are directed and labeled edges between the nodes where the lable of the edge is the predicate itself. Nodes denoted using ovals are resources identified by unique identifiers or URIs and the nodes denoted by rectangular boxes are string literals. Also the rdf:type edges are shown with bold arrows. Although this graph only shows a sample of the data triples, the schema triples can also be denoted in a similar manner. For path computation RDF data is modelled in this way as a graph with directed and labeled edges.

3.3 Algebraic Query Evaluation of Graph Pattern Matching

SPARQL [PS08] is the standard query language for RDF and its main primitive is a graph pattern, which is a set of triple patterns. A triple pattern is a triple which has variable components, that denoted with a leading ? i.e. are not bound to constants. A triple pattern is matched against the data triples in the dataset based on its bound components. Then, parts of the matching triples that correspond to the triple variables (with leading ?) are called "bindings" for that variable. Using basic SPARQL to query RDF data is called graph pattern matching. For example, suppose there is a SPARQL query with the triple pattern (?s :affWith :u2 .). When this pattern used to query on the data shown in Figure 3.4, the variable ?s will bind to {:prs1, :prs2, :prs3}. Each triple
pattern in a SPARQL query is joined to the next triple pattern using a "." at the end of the triple pattern and the join is performed on the common variable between two triple patterns. For example, if the query contains the graph pattern \( ?s \text{ :affWith :u1 . } ?s \text{ :affWith :u3 .} \) then \( ?s \) will bind to only \{ :prs1, :prs2 \}. This graph pattern contains a "subject-subject" join since both the triple patterns have the join variable in their subject position. Joins in SPARQL can also be "subject-object" join where one triple pattern has the join variable in the subject position while the other triple pattern. An example of such a join would be the graph pattern \( \langle :prs1 \text{ :lectures } ?o . \ ?o \text{ rdf:type :Course .} \rangle \). Here the variable \( ?o \) will bind to \{ :crs1 \}. The last type of join in SPARQL is the object-object join, for example, the graph pattern \( \langle :prs1 \text{ :affWith } ?o . \ ?o \text{ :affWith :prs3 .} \rangle \) where the variable \( ?o \) would bind to \{ :u1 \}.

Evaluation of graph pattern matching query is usually performed using operators with an algebraic query plan where we typically use relational-like query operators. The graph patterns are compiled into an algebraic logical plan representation, which is generally a sequence of query operators with an implied execution ordering. For example, Jena ARQ [McB02][Jen] is a popular query engine that supports SPARQL queries and it creates a SPARQL Syntax-Expression(SSE) as an algebraic logical query plan. The last step in query evaluation is transforming the logical plan to a physical plan which depends on the physical execution environment.

3.3.1 Apache Jena

The popular query engine Jena ARQ [McB02][Jen] supports RDF SPARQL queries. [Kim17c] describes an optimized platform for graph pattern matching that uses Jena's parser and compiler. In the parsing step the query string is provided to the Jena parser which converts the string to a Query object. This Query object is a representation of the abstract syntax tree (AST) which is then compiled and transformed into the corresponding SPARQL Syntax-Expression (S-Expression). Figure 3.6(a) shows an example SPARQL query with three triple patterns. The first two triple patterns have a subject-subject join between them and the third triple pattern has a subject-object join with the previous two triple patterns. Here, deciding the order of joins is important. Figure 3.6(b) shows the corresponding transformation to an SSE. As per this SSE the subject-subject joins are performed first and that becomes a Basic Graph Pattern (BGP). The subject-object join is denoted by the "join" keyword between the two BGPs. At the top of the SSE is the projection operator that projects out the two variables \( ?s1 \) and \( ?s \). Since the query has a SELECT \( * \) WHERE all the variables are projected to the output. In case only certain variables were selected in the query only those would be projected out by the SSE.
3.4 Semstorm

[Kim17d][Kim17a] introduces a Hadoop-based file organization that supports efficient storage of data collections. It also supports optimized graph pattern matching queries which are executed in Apache Tez [Tez] environment. Prior to using Semstorm for executing SPARQL queries, a pre-processing step is necessary. This step prepares the dataset for querying by grouping triples in triplegroups. Triples are grouped together based on their subject URI and then these triplegroups are typed based on the specific predicates in each triplegroup. These pre-processed triples are stored in files and these files are later used for querying instead of the actual dataset. This pre-processing step helps in eliminating subject-subject joins during query execution since the data is stored and indexed accordingly, thus, improving the efficiency of Semstorm’s query execution.

Semstorm compiles the query string using Apache Jena which produces the logical query plan in the form of an SSE. Jena’s SSE is quite useful for Semstorm since Jena also groups triples in a Basic Graph Pattern based on the subject. Semstorm then takes this SSE and converts it into a physical query execution plan which in this case is nothing but a Tez DAG.
3.4.1 Apache Tez

Apache Tez is a framework that can be used by data processing applications in Hadoop. Tez improves upon the old MapReduce framework by dramatically improving its speed, while maintaining MapReduce's ability to scale. Tez provides a simple API to represent data processing steps in the form of a Directed Acyclic Graph (DAG). Each vertex in the DAG defines the logic for data manipulation (for example, filter conditions) and also states the resources and environment required for the logic to be executed (for example, input file or output file location etc). Every vertex is a composition of Logical Input, Logical Output, Processor and an optional Configuration payload. Each edge in the DAG defines the flow of data between producer and consumer vertices. During query execution in Tez, the intermediate results are stored in memory and only spills to disk when there is not enough memory to hold the data. Another factor that adds to the efficiency of Apache Tez is the fact that it processes a batch of rows instead of one row at a time.

In the DAG created by Semstorm the leaves are usually the nodes that reads data from HDFS while the root vertex writes output to HDFS. The intermediate vertices are involved in different types of processing steps such as joining, annotating, filtering etc.

3.4.2 Loops and Cycles:

A simple path is a path that does not contain any repeated nodes while a cycle is an alternating set of nodes and edges (a path) where a node is reachable starting from itself. Now cycles can be of multiple types such as a simple cycle where only the starting and ending node can be repeated. A loop [Gui] [BM76] (also called self-loop) can be thought of as a type of cycle with one node and one edge connects the node to itself.

3.5 Serpent

Serpent is a path query computation platform that computes all paths between a given set of sources and destinations (multi source, multi destination path computation). Serpent uses path-sequence and the SOLVE algorithm to execute path computation queries.
Our approach can be divided into three main phases.

1. **Query Expression**
   
   *Goal:* While expressing the gpq our goal was to minimize disruption to existing infrastructure, e.g. parser.
   
   *Solution:* To achieve this goal we introduced a syntactic sugar to represent the path operator.

2. **Query Compilation**
   
   *Goal:* For being able to compile the gpq we need to integrate graph pattern matching with path problem solving.
   
   *Solution:* We modified the graph pattern matching logical query plan generated by the pattern matching engine by splicing in algebraic path querying operators to produce a gpqs query plan.

3. **Query Execution**
   
   *Goal:* The final phase includes executing gpqs.
   
   *Solution:* For actually executing gpqs we must translate the gpq logical query plan created in the previous phase into physical query plan by introducing required physical query operators.
For implementing our approach, we built a prototype by integrating the graph pattern matching platform **SemStorm** with path computation platform **Serpent**. Semstorm is a Hadoop-based file organization storage system that supports efficient graph pattern matching query execution using an algebraic query evaluation technique. Semstorm uses Apache Tez as the execution environment. Serpent is a platform for finding all paths between a set of sources and destinations. It builds on the path algebraic technique using path-sequences to compute paths.

### 4.1 Query Expression

Introducing a new query class would typically require the extension of query language and processing framework. However, we adopted an approach of introducing a syntactic sugar that avoided the need for changing SPARQL’s query syntax. This approach also enables us to minimize disruption of existing infrastructure, for example, the parser of the graph pattern matching engine.

Although this approach has a lot of positives, it suffers from two main issues. Firstly, since we are not modifying the parser of the pattern matching platform, it would not recognize a new path operator in the query syntax, and such a query would fail during compilation. Secondly, the pattern matching platform would produce a graph pattern query plan which we must migrate to a gpq query plan. Also, we would need to identify the sub-query components, i.e., source variable, destination variable and constraint variable and then project the bindings to these variables while filtering out the bindings to the rest of the variables.

<table>
<thead>
<tr>
<th>Original query</th>
<th>SSE produced by Jena after parsing and compiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX rdf:<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a>)</td>
<td>(rdf:<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a>))</td>
</tr>
<tr>
<td>SELECT * WHERE {</td>
<td>(project (?s1 ?s)</td>
</tr>
<tr>
<td>?s1 rdf:type akt:Affiliated-Person .</td>
<td>(product)</td>
</tr>
<tr>
<td>?s1 akt:full-name &quot;Wendy E. Mackay&quot; .</td>
<td>(join</td>
</tr>
<tr>
<td>?s akt:has-author ?s1 .</td>
<td>[triple ?s1 rdf:type akt:Affiliated-Person]</td>
</tr>
<tr>
<td>?s2 akt:full-name &quot;Irene Greif&quot; .</td>
<td>[triple ?s1 akt:full-name &quot;Wendy E. Mackay&quot;]</td>
</tr>
<tr>
<td>?s2 akt:has-affiliation ?d .</td>
<td>(BGP</td>
</tr>
<tr>
<td>?s ?pathVar ?d .</td>
<td>[triple ?s2 akt:full-name &quot;Irene Greif&quot;]</td>
</tr>
<tr>
<td>}</td>
<td>[triple ?s2 akt:has-affiliation ?d])}}</td>
</tr>
</tbody>
</table>

**Figure 4.1** (a) An example path query in our implementation of the integrated platform (b) The SSE produced by Jena’s parser and compiler
4.1.1 Identifying GPQ Sub-Query Components in SPARQL* Queries:

Our syntactic sugar is based on adopting a pre-defined variable name ?pathVar as the path operator. We acknowledge the risk of other users using this variable in their queries, but assume this risk to be small. However, this operator should also be a legal variable that is recognized by the graph pattern matching platform's parser. This will ensure that the unaltered parser can parse and compile path queries without failing due to syntax issues. Here, we refer to SPARQL with our pre-defined variable ?pathVar as SPARQL*.

4.1.2 Implementation Strategy:

In this section, we describe the approach followed to identify the source and destination variables using the pre-defined path variable ?pathVar and then project them out from the graph patterns. Figure 4.1 shows a typical query in our integrated platform and the corresponding SSE produced by Jena's parser and compiler. The last triple pattern in example query in Figure 4.1(a), \(<?s \ ?pathVar \ ?d>\) denotes the path computation between all bindings to the variable ?s and the variable ?d. Presence of the ?pathVar variable in the predicate position implies that it is a path query. Now, we must keep track of the position of the source and destination variables in the graph patterns and finally, after all the joins have taken place we must project out only the bindings of the source and destination variables. These bindings would then go into the path operator. To do this, we create the required datastructures to hold the position information of the source and destination variables in the query. This information will be later required when we create the final physical plan of the query.

For our proof-of-concept prototype, we implemented the approach by integrating Semstorm [Kim17c] as the graph pattern matching platform and Serpent [GA13a][Gao18] as the path query computation platform. Semstorm uses the below two main datastructures as query plan representation to hold the position information of the different triple patterns in the submitted query.

- **subjObjListMap** holds the mapping between the subjects and the corresponding objects in the query. The subjObjListMap for the query in Figure 4.1(a) would be

  subjObjListMap: {?s=[[?s1]], ?s1=["Wendy E. Mackay"],
  ?s2=[["Irene Greif"], [?d]]}

- **subjPropListMap** holds the mapping between the subjects and the properties or predicates in the triple patterns in the query. The subjPropListMap of the query in Figure 4.1(a) would be

  subjPropListMap: {?s=[has-author], ?s1=[full-name],
  ?s2=[full-name, has-affiliation]}
In addition to these, the following datastructures have been added to facilitate path computation and provide required location information to the path operator.

- **pathSrcDst** is a map that shows the mapping between the source variable and its corresponding destination variable. For the query in Figure 4.1(a), the pathSrcDst would be

  \[ \text{pathSrcDst: } \{?s=[?d]\}\]

  The current structure of this map allows one source variable to have multiple destination variables, since the value in this map is a list of destination variables. However, we have not tested queries with multiple destinations and path operator variables. Some more development work would be required before our platform can handle multiple path operator variable and we leave that for future work.

- **srcMap** contains the source variable in the key position and a list of integers in the value position. The list of integers denote the exact position of the source variable in the subjObjListMap datastructure. The srcMap of the query in Figure 4.1(a) would be

  \[ \text{srcMap: } \{?s=[[0, -1]]\}\]

- **dstMap** is similar to the srcMap, except that its key contains the destination variable and the list of integers in its value position denote the position of the destination variable. The dstMap of the query in Figure 4.1(a) would be

  \[ \text{dstMap: } \{?d=[[2, 1]]\}\]

- **cndMap** is also same as the srcMap and dstMap except that it hold the constraints information. For example, some query might want to restrict paths to the ones which contain at least one \texttt{akt:has-affiliation} property or predicate. Then, this triple will be a part of the constraints and the position of this triple would be captured in the cndMap. The query in Figure 4.1(a) is not a constrained query and hence, its cndMap would be empty.

The list of integers in the value position of the **srcMap**, **dstMap** and **cndMap** all denote the position of the respective variables in subjObjListMap. For example, \{?s=[[0, -1]]\} means the variable \(?s\) is in the first BGP of subjObjListMap (indexing starts at 0) and -1 denotes that it is the subject of the BGP. \{?d=[[2, 1]]\} means that the variable \(?d\) is in the third BGP and it is the second object of that BGP. Sometimes these variables might also be the join variable between two graph patterns and so,
they can exist in multiple BGPs and the value of the respective maps will have a list of integer pairs, identifying the position of the variable in subjObjListMap. The information stored in these new datastructures are used during query evaluation phase later to identify the source, destination and constraint variables and then project only the bindings to these variables while filtering out the rest.

4.2 Query Compilation

Since we are interpreting the problem algebraically it translates to query evaluation being represented by an expression composed of multiple query operators in a specified order. It is easy to observe that gpqs are essentially comprised of four elements:

- Filtering of graph patterns,
- Graph pattern matching,
- Path finding,
- Path filtering

Figure 4.2 shows these four components and the pieces from the motivating example that correspond to each of these components. An algebraic evaluation of gpqs would involve translating each of these components into one or more query operator and then deciding the most efficient ordering of these operators. Since there are efficient algebraic pattern matching engines available, we can leverage these engines for the first two components. Then we would need to create logical operators to represent the path finder and path filter components.

![Figure 4.2 Mapping of the parts of motivating example to components of gpq](image)

**Figure 4.2** Mapping of the parts of motivating example to components of gpq
4.2.1 Logical Query Plan Transformation:

A simplifying but reasonable strategy to develop the logical query plan is the use of a fixed order between the graph pattern matching phase and the path computation phase. The rationale here is that in gpqs, pattern matching serves to compute the set of sources, destinations and/or intermediate nodes in constraints. In other words, the output of graph pattern matching can be seen as input to the path problem phase. Interpreting this in terms of query plans implies that the path computation and path filter operators will always be at the root of the tree for any gpqs query plans. In the sequel, we elaborate our realization of the above implied strategy.

Our query planning approach is based on transforming the query plan produced by graph pattern matching engine to a generalized path query evaluation plan. The intuition is that the subqueries which are the graph patterns defining the sets of sources, destinations and constraints for path computation can be translated to query plans in the usual manner. However, the semantics of such queries will usually imply a cross-product of intermediate results (since the subgraph patterns will be disconnected). We illustrate this idea with the example query in Figure 4.1(a) (but ignoring the last triple pattern \( \langle ?s \text{ pathVar} ?d \rangle \) which is our syntactic sugar for the path variable triple pattern). 4.1(b) shows the SSE created by Jena's parser and compiler and Figure 4.3(a) shows the SSE as a tree. Figure 4.3(b) shows the transformed query plan suited for the corresponding gpq.

To achieve the correct query semantics, the cross-product and projection operators have to be removed and query operators associated with path computation introduced. We would also need to identify the source variable and the Basic Graph Pattern (BGP) which has the source variable \( BGP_s \), identify the destination variable and the BGP which has the destination variable \( BGP_d \). Now we would need to add the path operator and send \( BGP_s \) and \( BGP_d \) as inputs to the path operator. After path computation the last step is filtering of paths based on the constraints provided in the query.
The new components of the query plan are highlighted in orange in Figure 4.3(b). The final operator in the plan is a path filter operator (if path filtering constraints are specified - absent in example). The newly introduced components of the query plan are enclosed in a dotted box in Figure 4.3(b).

4.3 Query Execution

Our graph pattern matching platform, Semstorm [Kim17c] is an RDF processing platform that is targeted for Cloud-processing and uses Apache Hadoop/Tez execution environment. Semstorm's compiler builds on Jena's parser, using Jena's SSE to create a Tez [Tez] DAG as the physical query plan based on Semstorm's query algebra. To achieve an equivalent physical query plan transformation, similar to the logical plan transformation in Figure 4.3, new physical query operators have to be introduced. Since our physical execution environment is Tez, the new physical operators are nothing but new Tez Vertices.

4.3.1 Implementation Strategy

We introduced four new Tez vertex types were added that act as the physical query operators.

- **Annotator Vertex for source, destination and constraint variables.** Semstorm is meant to run SELECT * WHERE queries and so, it propagates the data for all of the variables in the query. Hence, the output of the TypeScanner or Packager vertex contains subjects, predicates and objects that have bound to all the variables in the query. However, the path computation platform Serpent expects three lists of nodes that denote sources, destinations and constraints respectively. Hence, annotators were required to identify the source, destination or constraint variables and then, allow only the bindings for that variable to pass through, discarding the rest of the bindings. While this might seem to be a less optimized method, it must be noted that bindings to other variables cannot be discarded before all joins have completed since the source, destination or constraint variable may not always be the join variable.

- **PathComputer Vertex.** This is the path operator which performs the path computation. It takes the sources, destinations and constraints as input, converts these into three String arrays as is required by Serpent and then calls the appropriate method in the Serpent platform. For every path query DAG this vertex will always be at the root.

Figure 4.4 shows the final Tez DAG that needs to be generated for the example query in Figure 4.1(a). The TypeScanner::?s vertex in the DAG identifies and reads all triples that match the pattern \{?s akt:has-author ?s1\} from the pre-processed triplegroups file. Similarly, the other TypeScanner vertices read the respective matching triples. TypeScanner::?s1 vertex reads all graph patterns
that match the pattern {?s1 rdf:type akt:Affiliated-Person . ?s1 akt:full-name "Wendy E. Mackay"}. The \textit{TypeScanner::?s2} vertex reads all graph patterns that match the pattern {?s2 akt: full-name "Irene Greif" . ?s2 akt:has-affiliation ?d}. The output of the \textit{TypeScanner::?s} and \textit{TypeScanner::?s1} vertices go to \textit{Annotator(?s1):0} and \textit{Annotator(?s1):1} vertices respectively. These annotator vertices identify the join variable and its position in the graph pattern. For example, \textit{Annotator(?s1):0} identifies ?s1 as the join variable and it is at the first position (index starts at 0) in the \texttt{subjObjListMap} described earlier. The \textit{Packager::?s,?s1} vertex performs the actual join operation between the two graph patterns and provides the joined output of the two input graph patterns.

The \textit{TypeScanner}, \textit{Annotator} and \textit{Packager} vertices remain as it is in the Semstorm graph pattern matching platform without any modifications. If a simple pattern matching query (i.e., not a path computation query) is submitted to Semstorm, it would add a \textit{Producter} vertex that would take inputs from the \textit{Packager::?s,?s1} and \textit{TypeScanner::?s2} vertices and the output of the \textit{Producter} vertex would go into a \textit{Flattener} vertex which would write out the final query output to an output file.
on disk. In our integrated version, we created a fork at this point, where for a path query, we do not add the Producter and Flattener vertices. After joins, we add the Annotator-Source: [[0,-1]] and Annotator-Destination: [[2,1]] to annotate the source and destination respectively. We also add the value from the srcMap and dstMap to the respective vertex name. Since the destination vertex in our example is not involved in any joins the destination annotator vertex gets its input directly from the typeScanner vertex that has the destination variable. The PathComputer::Src:?s, Dst:?d vertex comes at the root of the DAG since this will be executed last. While creating this vertex, information about the source variable ?s and destination variable ?d are added to it using the configuration payload. This DAG is submitted to the query execution framework of Semstorm, which executes the DAG and produces the final path output.

### 4.4 Path Constraints

Some path constraints can be evaluated by reinterpreting the union and concatenation operations during the propagation algorithm (SOLVE) e.g. for shortest paths. Others will be defined as manipulations over the path expression produced by unconstrained version of the problem e.g. finding paths that contain a given set of nodes (no order specified). Those manipulations will be encapsulated in operators that are parent nodes of the pathComputer node in operator plan tree. The efficiency of the such operations will depend on the nature of path expression representation e.g. a binary encoded representation. However, a detailed discussion path constraints is outside scope of this thesis.

### 4.5 User Interface Updations:

Semstorm uses a settings file to capture configurations such as input file location, ontology file location and so on. Also, Semstorm requires the user to write the full SPARQL query and create a query file that must be provided as an argument to the Main method when running any queries. To remove this requirement of the user needing to write the full query, we also added a few new fields in the settings.ini file of the Semstorm platform. The new settings file is shown in below.

```ini
[semstorm]
localMode = true
numPartitions = 1
inputPath = src/test/resources/data/btc500M.nt
interimPath = triplegroups
ontologyPath = src/test/resources/ontology/univ-bench.nt
urlmapPath = src/main/resources/urlmap.json
```
outputPath = queryOutput
yarnResourceMangerPort = 8050
hdfsNamenodePort = 8020
tezCompressFilePath = /hdp/apps/current/tez/tez.tar.gz
SemStormFilePath = /hdp/apps/current/tez/aux-jars/SemStorm.jar
usePropTypeCache = true
triplePatterns = ?s1 rdf:type akt:Affiliated-Person , ?s1 akt:full-name "Wendy E. Mackay" , ?s akt:has-author ?s1 , ?s2 akt:full-name "Irene Greif" , ?s2 akt:has-affiliation ?d

[serpent]
preprocess = false
inputPath = src/test/resources/data/btc500M.nt
interimPath = pathseqs
nodeTypeMapPath = src/test/resources/nodetypemap.ser
isPathQuery = yes
direction = d
srcVar = ?s
dstVar = ?d
cndVar =
cndTriples =
constOption =

The generateSSE method in BGPQueryCompiler class in Semstorm was edited to enable it to read from the new settings file and construct the query by looking at the values of the prefix, triplePatterns, isPathQuery, srcVar, dstVar and cndTriples fields
CHAPTER

5

EVALUATION

5.1 Test setup

The primary goal of our evaluation was to compare our integrated system with existing platform that can perform multi-source multi-destination path querying where the source and destination nodes are described as graph patterns on the following parameters.

1. Performance, i.e., time taken to run the same queries.

2. Completeness of results, i.e., whether the platform returns all paths expected.

3. Expressiveness, i.e., what level of queries can be expressed in each platform.

4. Query compilation time comparison on our platform with and without path operator.

5.1.1 Dataset and Queries:

We ran our queries on the BTC500M dataset [Har11] (size 0.5GB, 2.5 million triples). While formulating queries we tried to focus on making sure that the final paths would be at least more than two hops long. Also we varied the number of source and destination nodes. The queries we ran varies from very small number of sources and destinations to very large set of sources and destinations. We ran five small queries and five large queries where small and large indicate the size of the set of sources and destinations. Table 5.1 shows the final sizes of the sources and destinations for the ten
queries we used. In the charts Small Queries and Large Queries have been abbreviated to SQ and LQ respectively due to shortage of space. The same queries were also modified to add constraints to run constrained query experiments.

All the comparisons were done with Stardog. We also considered comparing our platform with Neo4j. However, while trying to run the queries using Cypher we found that the resources available on our server was not enough to run all-paths queries on the BTC dataset on Neo4j. All-paths queries on this dataset were running indefinitely and then causing the Neo4j server to crash. We were able to run shortest path queries on Neo4j but that result is not included in this thesis as finding shortest path is not an evaluation goal for this thesis.

5.1.2 Hardware Configuration:

Evaluation was conducted on single node running Hadoop in a privately owned RedHat Enterprise Server server housed in the University’s server lab. The server is equipped with Xeon octa core x86_64 CPU (2.33 GHz), 40GB RAM, and two HDDs (3.6TB and 445GB). All results have been averaged over five trials.

5.2 Evaluation Results

In all the charts our platform has been labelled as "Sem-Ser".

5.2.1 Performance Evaluation:

When comparing absolute time taken by our platform with that of Stardog, we found that Stardog performed better in all queries except for SmallQuery 1, LargeQuery 1 and LargeQuery 2. The LargeQuery 2 timed out and produced only partial results when run on Stardog. This query took the longest time (5.5 minutes) and produced the largest number of paths (0.8 million paths) in our platform. This is mainly because the graph patterns provided for the source and destination nodes

<table>
<thead>
<tr>
<th>Queries</th>
<th>Sources</th>
<th>Destinations</th>
<th>Queries</th>
<th>Sources</th>
<th>Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallQuery 1</td>
<td>25</td>
<td>2</td>
<td>LargeQuery 1</td>
<td>13641</td>
<td>907</td>
</tr>
<tr>
<td>SmallQuery 2</td>
<td>4</td>
<td>6</td>
<td>LargeQuery 2</td>
<td>29974</td>
<td>32583</td>
</tr>
<tr>
<td>SmallQuery 3</td>
<td>4</td>
<td>3</td>
<td>LargeQuery 3</td>
<td>11793</td>
<td>6</td>
</tr>
<tr>
<td>SmallQuery 4</td>
<td>29</td>
<td>7</td>
<td>LargeQuery 4</td>
<td>29974</td>
<td>2290</td>
</tr>
<tr>
<td>SmallQuery 5</td>
<td>26</td>
<td>31</td>
<td>LargeQuery 5</td>
<td>2290</td>
<td>32582</td>
</tr>
</tbody>
</table>

*Figure 5.1* Size of source and destination sets for each query
is quite general, thus leading to a large number of matches. As can be seen from Table 5.1 this query resulted in almost thirty thousand source nodes and over thirty two thousand destination nodes. Consequently there were a large number of paths connecting these nodes.

5.2.2 Completeness of results:

Figure 5.2(a) shows the number of paths identified by small queries and Figure 5.2(b) shows the same for larger queries executed. \( LQ_2 \) has been marked with an asterisk since it did not finish in Stardog and hence, all the charts have only one value for this query. For all the queries Stardog produced incomplete results and also duplicate paths. This dataset has a lot of triples such as \( \langle \text{acm:58567 akt:has-publication-reference acm:58567} \rangle \). In this triple the subject and the object is the same uri \( \text{acm:58567} \) and hence this is called a loop or self-loop. The BTC dataset has a lot of such triples and Stardog does not consider the self loops in the paths it identifies. For example, suppose we have the RDF graph shown in Figure 5.3 consisting of the following triples \( \langle A \ p1 \ A \rangle \langle A \ p2 \ B \rangle \langle B \ p3 \ B \rangle \). Now suppose the source node given to the query is \( A \) and destination node is \( B \). On path query execution Stardog will ignore the self-loops \( \langle A \ p1 \ A \rangle \) and \( \langle B \ p3 \ B \rangle \) and will output only one path \( \langle A \ p2 \ B \rangle \). However, our platform will include the self-loops and hence find four paths \( \langle A \ p2 \ B \rangle, \langle A \ p1 \ A \ p2 \ B \rangle, \langle A \ p2 \ B \ p3 \ B \rangle \) and \( \langle A \ p1 \ A \ p2 \ B \ p3 \ B \rangle \). Figure 5.3(b) shows the paths returned by each platform. Using the "CYCLIC" keyword, while running the queries in Stardog, to detect loops also did not work since this keyword is looking for "cycles" and not "loops". Rather than giving the correct number of paths, use of the "CYCLIC" keyword reduced the number of paths to zero (most likely because the source and destination sets were disjoint - hence there were no cycles). This is the reason behind Stardog mostly finding less paths as compared to our platform.

In some queries, for example, SmallQuery_1 and LargeQuery_1, Stardog does find more number of paths. However, these results contain duplicate paths. For example, although Stardog
produces 40 paths for SmallQuery, the number of unique paths is 6. Since there was a big mismatch between the number of paths found by our platform and Stardog we compared the time taken per path identified rather than the absolute time taken for executing each query. Figure 5.4(a) shows the time per path comparison for the small queries and Figure 5.4(b) shows the same for large queries. For some queries, which our platform is better optimized for, the difference in time taken per path was big, while for others the difference is quite small. However, in all of the queries our platform takes less time per path as compared to Stardog.

5.2.3 Expressiveness:

All types of graph patterns can be expressed in Neo4j, Stardog as well as our platform. However, Stardog does not support constraints such as ALL, ANY, NONE which can be used to filter out certain paths as required. Figure 5.6 shows the comparison of the expressiveness of our platform with that of Stardog and Neo4j. The "ALL" keyword in Stardog is used to denote all paths. Neo4j
has predicate functions (all, any, exists, none, single) which can be used for the same purpose of filtering. However, since we were not able to run all paths queries on Neo4j it was not possible to compare constrained queries on our platform with that of Neo4j. Figure 5.5 shows the time taken for constrained queries as compared to unconstrained queries on our platform. All of the constrained queries understandably taken longer time to complete query execution, since these queries include an extra filtering step. For all of the small queries the increase in execution time is minimal mainly because the size of the resulting set of paths before filtering is also small. For the large queries the increase in execution time is more noticeable. This is again due to the larger size of the resultset of paths before filtering causing a longer filtering time.

<table>
<thead>
<tr>
<th>Query</th>
<th>Sem-Ser</th>
<th>Stardog</th>
<th>Neo4j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern matching Query</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Unconstrained Path Query</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Constrained Path Query</td>
<td>✔️</td>
<td>❌</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Figure 5.5 Execution Time of constrained queries vs unconstrained queries

Figure 5.6 Table showing comparison of the level of expressiveness of our platform with Neo4j and Stardog
5.2.4 Query Compilation Time Comparison:

Figure 5.7 shows the time taken for query compilation on our platform for queries which have the path operator compared with the same queries without the path operator. As can be seen in the figure the path operator does not affect query compilation time by a lot. For most of the queries the compilation time increased by less than one second.

5.3 List of Queries

To formulate path queries that will work on the btc500 dataset we loaded the dataset first in Neo4j to visualize the whole graph. This visualization helps in identifying suitable paths that can be used to perform experiments. Due to the large size of the dataset Neo4j was running out of memory when trying to visualize the graph of the whole dataset. Hence, we started visualizing parts of the graph to identify potential paths and consequently write required queries. One of the main criteria that we used to filter out paths is the number of hops. We aimed at finding paths that have more than two hops. Once the paths have been identified we examined the source and destination nodes of the paths and tried to formulate SPARQL queries that describe those nodes. The next section lists the final ten Sem-Ser path queries that we used to run our experiments. Each Sem-Ser query was also translated to a Stardog path query with start and end patterns. In the next section every Sem-Ser query is followed by the corresponding Stardog path query.
5.3.1 Small Queries

We ran two sets of queries to test the full range of capabilities of our platform. The small queries set has a small number of source and destination nodes and consequently resulted in a smaller number of paths. This section lists the small Sem-Ser queries and the corresponding Stardog queries along with the number of sources and destinations for each query.

5.3.1.1 SmallQuery1 - Source nodes: 25, Destination nodes: 2

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>
PREFIX acmper:<http://acm.rkbexplorer.com/id/>

SELECT * WHERE {
    ?s akt:has-author ?o2 .
    acmper:person-240119 akt:has-affiliation ?d .
    ?s ?pathVar ?d .
}

Stardog Query:
stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>
PREFIX acmper:<http://acm.rkbexplorer.com/id/>

PATHS START ?s {
    ?s akt:has-author ?o2 .
}
END ?d {
    acmper:person-240119 akt:has-affiliation ?d .
} VIA ?p "}
5.3.1.2 SmallQuery₂ - Source nodes: 4, Destination nodes: 6

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {
    ?s1 rdf:type akt:Affiliated-Person .
    ?s1 akt:full-name "Wendy E. Mackay" .
    ?s akt:has-author ?s1 .
    ?s2 akt:full-name "Irene Greif" .
    ?s2 akt:has-affiliation ?d .
    ?s ?pathVar ?d .
}

Stardog Query:
stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PATHS START ?s {
    ?s1 rdf:type akt:Affiliated-Person .
    ?s1 akt:full-name 'Wendy E. Mackay' .
    ?s akt:has-author ?s1 .}
END ?d {
    ?s2 akt:full-name 'Irene Greif' .
    ?s2 akt:has-affiliation ?d .}
VIA ?p "

5.3.1.3 SmallQuery₃ - Source nodes: 4, Destination nodes: 3

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acmper:<http://acm.rkbexplorer.com/id/>

SELECT * WHERE {
    ?s akt:has-author acmper:person-117520 .
    ?s1 akt:has-affiliation ?d .
}
?s1 akt:full-name "Violet R. Syrotiuk" .
?s ?pathVar ?d .
}

Stardog Query:

```
stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acmper:<http://acm.rkbexplorer.com/id/>
PATHS START ?s {
  ?s akt:has-author acmper:person-117520 . } END ?d {
  ?s1 akt:has-affiliation ?d .
  ?s1 akt:full-name 'Violet R. Syrotiuk' . } VIA ?p "
```

5.3.1.4 SmallQuery4 - Source nodes: 29, Destination nodes: 7

Sem-Ser Query:

```
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>
SELECT * WHERE {
  ?s1 rdf:type akt:Affiliated-Person .
  ?s1 akt:full-name "I. Bolsens" .
  ?s1 akt:has-affiliation ?d .
  ?s ?pathVar ?d .
}
```

Stardog Query:

```
stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>
PATHS START ?s {
```
5.3.1.5 SmallQuery5 - Source nodes: 26, Destination nodes: 31

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>

SELECT * WHERE {
    ?b akt:has-affiliation ?d .
    ?s ?pathVar ?d .
}

Stardog Query:
stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>
PATHS START ?s {
    ?s akt:addresses-generic-area-of-interest acm:H.3.1 .}
END ?d {
    ?b akt:full-name 'Andrei Z. Broder' .
    ?b akt:has-affiliation ?d .}
VIA ?p "

5.3.2 Large Queries

All of the large queries set have large number of source nodes and some of them have large number of destination nodes, while others have a smaller number of destination nodes. Since the number of sources and destinations is larger these queries have resulted in a large number of paths. We have
tried to run a wide range of source and destination sizes and hence the size of the source nodes set in
the queries in this section varies from a little over two thousand to almost thirty thousand. The size
of the destination nodes set has an even larger variation, varying from as little as six nodes to over
thirty two thousand nodes. This section lists the all large Sem-Ser queries and the corresponding
Stardog queries along with the number of sources and destinations for each query.

5.3.2.1 LargeQuery1 - Source nodes: 13641, Destination nodes: 907

Sem-Ser Query:
PREFIX semd: <http://www.semanticdesktop.org/ontologies/2007/01/19/nie#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX linkd: <http://data.linkedct.org/resource/linkedct/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>

SELECT * WHERE {
  ?s semd:mimeType "application/rdf+xml" .
  ?s rdfs:label ?o .
  ?s2 foaf:based_near ?d .
  ?s2 rdf:type linkd:location .
  ?s ?pathVar ?d .
}

Stardog Query:
stalk query btc500m
"PREFIX semd: <http://www.semanticdesktop.org/ontologies/2007/01/19/nie#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX linkd: <http://data.linkedct.org/resource/linkedct/>/
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PATHS START ?s {
  ?s semd:mimeType 'application/rdf+xml' .
  ?s rdfs:label ?o .
} END ?d {
  ?s2 foaf:based_near ?d .
  ?s2 rdf:type linkd:location .
}
5.3.2.2 LargeQuery₂ - Source nodes: 29974, Destination nodes: 32583

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {
    ?s akt:has-author ?o2 .
    ?s1 akt:full-name ?o1 .
    ?s1 akt:has-affiliation ?d .
    ?s ?pathVar ?d .
}

Stardog Query:
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PATHS START ?s {
    ?s akt:has-author ?o2 . }
END ?d {
    ?s1 akt:full-name ?o1 .
    ?s1 akt:has-affiliation ?d . }
VIA ?p "

5.3.2.3 LargeQuery₃ - Source nodes: 11793, Destination nodes: 6

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {
    ?s1 rdf:type akt:Affiliated-Person .
}
?s1 akt:full-name ?o1 .
?s akt:has-author ?s1 .
?s2 akt:full-name "Irene Greif" .
?s2 akt:has-affiliation ?d .
?s ?pathVar ?d .

Stardog Query:

stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PATHS START ?s {
    ?s1 rdf:type akt:Affiliated-Person .
    ?s1 akt:full-name ?o1 .
    ?s akt:has-author ?s1 . }
END ?d {
    ?s2 akt:full-name 'Irene Greif' .
    ?s2 akt:has-affiliation ?d . }
VIA ?p "

5.3.2.4 LargeQuery₄ - Source nodes: 29974, Destination nodes: 2290

Sem-Ser Query:

PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {
    ?s akt:has-author ?o2 .
    ?s ?pathVar ?d .
}

Stardog Query:

stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
PATHS START ?s {
    ?s akt:has-author ?o2 . }
END ?d {
    ?d akt:addresses-generic-area-of-interest acm:F.2.2 . }
VIA ?p "

5.3.2.5 LargeQuery5 - Source nodes: 2290, Destination nodes: 32582

Sem-Ser Query:
PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>

SELECT * WHERE {
    ?s akt:addresses-generic-area-of-interest acm:F.2.2 .
    ?s1 rdf:type akt:Affiliated-Person .
    ?s1 akt:full-name ?o1 .
    ?s1 akt:has-affiliation ?d .
    ?s ?pathVar ?d .
}

Stardog Query:

stardog query btc500m
"PREFIX akt:<http://www.aktors.org/ontology/portal#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX acm:<http://acm.rkbexplorer.com/ontologies/acm#>
PATHS START ?s {
    ?s akt:addresses-generic-area-of-interest acm:F.2.2 . }
END ?d {
    ?s1 rdf:type akt:Affiliated-Person .
    ?s1 akt:full-name ?o1 .
    ?s1 akt:has-affiliation ?d . }
VIA ?p "

44
This thesis presents an algebraic query evaluation strategy to evaluate generalized path queries with declaratively defined source and destination nodes. This thesis also presents a general framework and steps to integrate any existing graph pattern matching platform with a path computation platform. Lastly, this thesis describes an implementation of such an integrated platform and shows performance comparison with this integrated platform with that of popular platforms that can handle such generalized path queries.

Acknowledgment

The work presented in this thesis is partially funded by NSF grant IIS-1218277 and CNS-1526113.
BIBLIOGRAPHY


[Gui] Guichard, D. “An Introduction to Combinatorics and Graph Theory” ().


48
A.1 New Code

This section provides a part of the new code written to implement the proof-of-concept prototype. The whole code base is available at The entire code base is available at https://github.ncsu.edu/abhatt22/Semstorm-Serpent.

```java
package edu.ncsu.csc.coul.semstorm.query.planner;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.apache.hadoop.io.Text;
import org.apache.tez.common.TezUtils;
import org.apache.tez.dag.api.DAG;
import org.apache.tez.dag.api.DataSinkDescriptor;
```
import org.apache.tez.dag.api.Edge;
import org.apache.tez.dag.api.ProcessorDescriptor;
import org.apache.tez.dag.api.TezConfiguration;
import org.apache.tez.dag.api.UserPayload;
import org.apache.tez.dag.api.Vertex;
import org.apache.tez.runtime.library.conf.UnorderedKVEdgeConfig;
import org.apache.tez.runtime.library.conf.UnorderedPartitionedKVEdgeConfig;
import org.apache.tez.runtime.library.conf.HashPartitioner;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
import edu.ncsu.csc.coul.semstorm.query.operator.Operator;
import edu.ncsu.csc.coul.semstorm.query.operator.TGAnnotateCndProcessor;
import edu.ncsu.csc.coul.semstorm.query.operator.TGAnnotateDstProcessor;
import edu.ncsu.csc.coul.semstorm.query.operator.TGAnnotateProcessor;
import edu.ncsu.csc.coul.semstorm.query.operator.TGAnnotateSrcProcessor;
import edu.ncsu.csc.coul.semstorm.query.operator.TGPathCompProcessor;
import edu.ncsu.csc.coul.semstorm.query.operator.TGProductProcessor;
import edu.ncsu.csc.coul.semstorm.storage.writable.TGWritable;
import edu.ncsu.csc.coul.semstorm.util.ObjectSerializer;

/**
 * @author Abhisha Bhattacharyya (abhatt22@ncsu.edu)
 */

class PathOperatorSubPlanGenerator {
    private static final Logger log =
        LoggerFactory.getLogger(PathOperatorSubPlanGenerator.class);
    private Map<String, List<Set<String>>> subjObjListMap;
    private Map<String, Set<String>> pathSrcDst;
}
private Map<String, Set<List<Integer>>> srcMap;
private Map<String, Set<List<Integer>>> dstMap;
private Map<String, Set<List<Integer>>> cndMap;
private String cndVar;

public PathOperatorSubPlanGenerator(
    Map<String, List<Set<String>>> subjObjListMap,
    Map<String, Set<String>> pathSrcDst,
    Map<String, Set<List<Integer>>> srcMap,
    Map<String, Set<List<Integer>>> dstMap,
    Map<String, Set<List<Integer>>> cndMap,
    String cndVar)
{
    this.subjObjListMap = subjObjListMap;
    this.pathSrcDst = pathSrcDst;
    this.srcMap = srcMap;
    this.dstMap = dstMap;
    this.cndMap = cndMap;
    this.cndVar = cndVar;
}

public void generateSubDAG(TezConfiguration tezConf, DAG dag,
    List<Vertex> pathOpVertexList, DataSinkDescriptor
dataSinkPath,
    String direction, String inputPath, String interimPath,
    String constOption, String nodeTypeMap) {
    UnorderedPartitionedKVEdgeConfig
        unorderedPartitionedEdgeConf =
            UnorderedPartitionedKVEdgeConfig.newBuilder(
                Text.class.getName(), TGWritable.class.getName(),
                HashPartitioner.class.getName()).setFromConfiguration(
                tezConf).build();

    Vertex annoSrcVertex = null;
    Vertex annoDstVertex = null;
Vertex annoCndVertex = null;

/* Below code assumes there will be only one source and one destination
* Hence only one srcVertex and dstVertex is used.
* In case pathSrcDst has more than one source or destination
* only the last one will be saved, all others will be overwritten.
*/
for(String src : pathSrcDst.keySet()) {
    /* Creating Source annotator vertex
     * and then connecting an edge between Source vertex and Source annotator vertex
     */
    annoSrcVertex = createVertex(src,
                                pathOpVertexList, tezConf, "Src", dag);

    Set<String> dstSet = pathSrcDst.get(src);
    for(String dst : dstSet) {
        /* Creating Destination annotator vertex
         * and then connecting an edge between Destination vertex and Destination annotator vertex
         */
        annoDstVertex = createVertex(dst,
                                      pathOpVertexList, tezConf, "Dst", dag);

        /* Creating Condition annotator vertex
         * and then connecting an edge between Condition vertex and Condition annotator vertex
         */
        log.info("cndVar: "+cndVar);

        if(!cndVar.isEmpty()) {
            annoCndVertex = createVertex(cndVar,
                                          pathOpVertexList, tezConf, "Cnd", dag);
        }
/* Creating PathComputer vertex
* and then connecting edges between Source annotator and PathComputer
* Vertex
* and also Destination annotator and PathComputer vertex
*/

String srcDstTrackVar = new String();
srcDstTrackVar = "Src: " + src + ", Dst: " + dst;
log.info("srcDstTrackVar: " + srcDstTrackVar);

List<String> srcDstList =
    Lists.newArrayList();
srcDstList.add(src);
srcDstList.add(dst);
log.info("srcDstList: " + srcDstList);

UserPayload pathCompVertexUserPayload = null;
String serSrcDstList = null;
String serDirection = null;
String serInputPath = null;
String serInterimPath = null;
String serConstOption = null;
String serNodeTypeMap = null;

try {
    serSrcDstList =
        ObjectSerializer.serialize((Serializable) srcDstList);
    tezConf.set("SrcDstList", serSrcDstList);
}
serDirection =
  ObjectSerializer.serialize(
  (Serializable) direction);
  tezConf.set("Direction",
  serDirection);
serInputPath =
  ObjectSerializer.serialize(
  (Serializable) inputPath);
  tezConf.set("InputPath",
  serInputPath);
serInterimPath =
  ObjectSerializer.serialize(
  (Serializable) interimPath);
  tezConf.set("InterimPath",
  serInterimPath);
serConstOption =
  ObjectSerializer.serialize(
  (Serializable) constOption);
  tezConf.set("ConstOption",
  serConstOption);
serNodeTypeMap =
  ObjectSerializer.serialize(
  (Serializable) nodeTypeMap);
  tezConf.set("NodeTypeMap",
  serNodeTypeMap);

pathCompVertexUserPayload =
  TezUtils.createUserPayloadFromConf(
  tezConf);
}
} catch (Exception e) {
  log.error(e.getMessage(), e);
  return;
}
Vertex pathCompVertex = 
  ~ Vertex.create(Operator.PATHCOMPUTER + 
  ~ "::" + srcDstTrackVar,
  ~ ProcessorDescriptor.create( 
  ~ TGPathCompProcessor.class.getName() 
  ~ ).setUserPayload( 
  ~ pathCompVertexUserPayload), 1);

pathCompVertex.addDataSink( 
  ~ Operator.OUTPUT, dataSinkPath);

dag.addVertex(pathCompVertex);

Edge manyToOneEdgeSrc = 
  ~ Edge.create(annoSrcVertex, 
  ~ pathCompVertex, 
  ~ unorderedPartitionedEdgeConf 
  ~ .createDefaultEdgeProperty());

dag.addEdge(manyToOneEdgeSrc);

log.info("New Edge added - "+
  ~ annoSrcVertex.getName() + " to "+
  ~ pathCompVertex.getName());

Edge manyToOneEdgeDst = 
  ~ Edge.create(annoDstVertex, 
  ~ pathCompVertex, 
  ~ unorderedPartitionedEdgeConf 
  ~ .createDefaultEdgeProperty());

dag.addEdge(manyToOneEdgeDst);

log.info("New Edge added - "+
  ~ annoDstVertex.getName() + " to "+
  ~ pathCompVertex.getName());

56
if(!cndVar.isEmpty()) {
    Edge manyToOneEdgeCnd =
        Edge.create(annoCndVertex,
                  pathCompVertex,
                  unorderedPartitionedEdgeConf
                    .createDefaultEdgeProperty());
    dag.addEdge(manyToOneEdgeCnd);
    log.info("New Edge added - " +
              annoCndVertex.getName() + " to " +
              pathCompVertex.getName());
}

public Vertex getVertex(List<Vertex> pathOpVertexList, String var, Set<List<Integer>> position, List<String> trackVarList) {
    Vertex retVertex=null;
    log.info("pathOpVertexList: "+pathOpVertexList+ " var:"+var+" position:"+position+ " trackVarList:
              "+trackVarList);
    for(List<Integer> pos : position) {
        if(pos.get(1)==-1) {
            //if the given variable is in subject position of triple
            for(Vertex v : pathOpVertexList) {
                if(v.getName().contains(var)) {
                    retVertex = v;
                    break;
                }
            }
        }
    }
}
else {
    // if given variable is in object position
    // of triple
    for(String sub : subjObjListMap.keySet()) {
        List<String> objList = new ArrayList<String>();
        for(Set<String> objSet: subjObjListMap.get(sub)) {
            objList.addAll(objSet);
        }
        if(objList.contains(var)) {
            trackVarList.add(sub);
        }
    }
}

for(Vertex v : pathOpVertexList) {
    for(int i=0; i< trackVarList.size(); i++) {
        /* if the name of the current vertex does not contain any one of the strings in trackVarList
         * then break out of the inner for loop and move on to the next iteration of the outer for loop
         */
        if(!(v.getName().contains(trackVarList.get(i)))) {
            break;
        }
    }
}

/* if given variable is in object position
 // of triple
 for(String sub : subjObjListMap.keySet()) {
    List<String> objList = new ArrayList<String>();
    for(Set<String> objSet: subjObjListMap.get(sub)) {
        objList.addAll(objSet);
    }
    if(objList.contains(var)) {
        trackVarList.add(sub);
    }
 }

for(Vertex v : pathOpVertexList) {
    for(int i=0; i< trackVarList.size(); i++) {
        /* if the name of the current vertex does not contain any one of the strings in trackVarList
         * then break out of the inner for loop and move on to the next iteration of the outer for loop
         */
        if(!(v.getName().contains(trackVarList.get(i)))) {
            break;
        }
    }
*/

58
if (i == trackVarList.size() - 1) {
    retVertex = v;
}

return retVertex;

public Vertex createVertex(String var, List<Vertex> pathOpVertexList, TezConfiguration tezConf, String type, DAG dag) {
    UnorderedKVEdgeConfig unorderedKVEdgeConfig = UnorderedKVEdgeConfig.newBuilder(Text.class.getName(), TGWritable.class.getName()).setFromConfiguration(tezConf).build();

    Vertex ver = null;
    Vertex annoVertex = null;

    List<String> trackVarList = new ArrayList<String>();
    Set<List<Integer>> position = Sets.newHashSet();

    if (type.equals("Src")) {
        position = srcMap.get(var);
    } else if (type.equals("Dst")) {
        position = dstMap.get(var);
    } else if (type.equals("Cnd")) {
        position = cndMap.get(var);
    }

    return new Vertex(ver, annoVertex, unorderedKVEdgeConfig, trackVarList, pathOpVertexList, position);
ver = getVertex(pathOpVertexList, var, position, trackVarList);

//log.info("Destination Vertex : " + ver.getName());

UserPayload annoVertexUserPayload = null;
String serVariable= null;
String serPosition = null;
String serSubName = null;
String serSubjObjListMap = null;
String Variable = type+"Variable";
String Position = type+"Position";
String SubName = type+"SubName";

try {
    serVariable =
        ObjectSerializer.serialize((Serializable) var);
tezConf.set(Variable, serVariable);
    serPosition =
        ObjectSerializer.serialize((Serializable) position);
tezConf.set(Position, serPosition);
    serSubjObjListMap =
        ObjectSerializer.serialize((Serializable) subjObjListMap);
tezConf.set("SubjObjListMap", serSubjObjListMap);

    if(trackVarList.size()>0) {
        serSubName = ObjectSerializer.serialize(
                (Serializable) trackVarList.get(0));
tezConf.set(SubName, serSubName);
    }

    //log.info(type+"SubName: "+trackVarList);
}
annoVertexUserPayload =
→ TezUtils.createUserPayloadFromConf(tezConf);
} catch (Exception e) {
    log.error(e.getMessage(), e);
    return null;
}

if(type.equals("Src")) {
    annoVertex = Vertex.create(Operator.ANNOTATOR + 
        "- Source: " + position,
        ProcessorDescriptor.create(
            TGAnnotateSrcProcessor.class.getName())
        .setUserPayload(annoVertexUserPayload));
} else if(type.equals("Dst")) {
    annoVertex = Vertex.create(Operator.ANNOTATOR + 
        "- Destination: " + position,
        ProcessorDescriptor.create(
            TGAnnotateDstProcessor.class.getName())
        .setUserPayload(annoVertexUserPayload));
} else if(type.equals("Cnd")) {
    annoVertex = Vertex.create(Operator.ANNOTATOR + 
        "- Condition: " + position,
        ProcessorDescriptor.create(
            TGAnnotateCndProcessor.class.getName())
        .setUserPayload(annoVertexUserPayload));
}

//annoDstVertex.addDataSink(Operator.OUTPUT, dataSinkDst);

dag.addVertex(annoVertex);
Edge oneToOneEdgeDst = Edge.create(ver, annoVertex,
    unorderedKVEdgeConfig
    .createDefaultOneToOneEdgeProperty());
dag.addEdge(oneToOneEdgeDst);
log.info("New Edge added - " + ver.getName() + " to " + annoVertex.getName());

return annoVertex;
public class TGAnnotateSrcProcessor extends SimpleMRProcessor {
    private static final Logger log =
        LoggerFactory.getLogger(TGPathCompProcessor.class);
    private Configuration conf;
    private String srcVar;
    private String subName;
    private Set<List<Integer>> srcPosition;
    private Map<String, List<Set<String>>> subjObjListMap;
    private ObjectSerializer serializer;

    public TGAnnotateSrcProcessor(ProcessorContext context) {
        super(context);
        try {
            UserPayload userPayload =
                getContext().getUserPayload();
            this.conf = TezUtils.createConfFromUserPayload(
                userPayload);

            String serSrcVar = conf.get("SrcVariable");
            srcVar = (String)
                serializer.deserialize(serSrcVar);

            String serSrcPosition = conf.get("SrcPosition");
            srcPosition = (Set<List<Integer>>) 
                serializer.deserialize(serSrcPosition);

            String serSubjObjListMap =
                conf.get("SubjObjListMap");
            subjObjListMap = (Map<String, List<Set<String>>>)
                serializer.deserialize(serSubjObjListMap);

            String serSubName = conf.get("SrcSubName");
subName = (String)
    ~ serializer.deserialize(serSubName);

} catch (Exception e) {
    log.error(e.getMessage(), e);
}
}

@override
public void run() throws Exception {
    LogicalOutput output =
        ~ getOutputs().values().iterator().next();
    KeyValueWriter kvWriter = (KeyValueWriter)
        ~ output.getWriter();

    LogicalInput input =
        ~ getInputs().values().iterator().next();

    String inputVerName =
        ~ getInputs().keySet().iterator().next();
    //log.info("getInputs().keySet().iterator().next():
    ~ "+inputVerName);

    KeyValueReader kvReader = (KeyValueReader)
        ~ input.getReader();

    //List<Text> srcList = Lists.newArrayList();

    if(inputVerName.contains(Operator.TYPESCANNER)) { //src
        ~ vertex is not involved in any joins
        //log.info("Src var not involved in any joins");
        findOutput(kvReader, kvWriter);
    }
    else {
        ~ When source variable is involved in joins
    }
if(srcPosition.size()==1) {
   //log.info("Src var involved in join but not the join var");
   for(List<Integer> srcPos : srcPosition) {
      if(srcPos.get(1)==-1) {
         //source variable is in the
         // subject position of the
         // triplegroup
         getSubjSrc(inputVerName,
                      kvReader, kvWriter);
      }
      else {
         //source variable is in
         // the object position
         // of the triplegroup
         getObjSrc(inputVerName,
                      kvReader, kvWriter);
      }
   }
   else {
      //source variable is the join parameter.
      //log.info("Src var is the join parameter");
      boolean flag = false;
      for(List<Integer> srcPos : srcPosition) {
         if(srcPos.get(1)==-1) {
            if(flag) {
               //source variable is in the subject
               // position of the triplegroup
               break;
            }
         }
      }
      if(flag) {
         //source variable is the join parameter
      }
   }
}
//log.info("Src var in the subject position of triplegroup");
   getSubjSrc(inputVerName, kvReader, kvWriter);
}
else {
   //source variable is in the object position of the triplegroup
   //log.info("Src var in object position of triplegroup");
   getObjSrc(inputVerName, kvReader, kvWriter);
   
   }

void findOutput(KeyValueReader kvReader, KeyValueWriter kvWriter)
   {
      for(List<Integer> srcPos : srcPosition) {
         if(srcPos.get(1)==-1) {
            //src is in subject position
            try {
               while (kvReader.next()) {
                  TGWritable tgWritable = (TGWritable) kvReader .getCurrentValue();
                  List<Text> srcList = Lists.newArrayList();
                  Text tgSubj = null;
                  tgSubj = tgWritable.subjList .get(0);
                  srcList.add(tgSubj);
               }
            }
         }
      }
   }
TGWritable newTGWritable = new TGWritable(
    Lists.newArrayList(),
    srcList,
    Lists.newArrayList());

kvWriter.write(new Text(),
    newTGWritable);

//log.info("srcList: "+srcList);
}

} catch (Exception e) {
    log.error(e.getMessage(), e);
}

} else {
    //src is in object position
    try {
        while (kvReader.next()) {
            TGWritable tgWritable =
                (TGWritable) kvReader.getCurrentValue();
            List<Text> srcList =
                Lists.newArrayList();

            List<List/Set<Text>>>
                tgObjsList =
                tgWritable.objsList;
            List/Set<Text>> tgObjs =
                tgObjsList.get(0);
            Set<Text> objSet =
                tgObjs.get(
                    srcPos.get(1));

            srcList.addAll(objSet);
void getSubjSrc(String inputVerName, KeyValueReader kvReader, 
       KeyValueWriter kvWriter) {

    /* First we need to find out the position of the source variable 
    * in the subject list of the input. We do this by looking at the 
    * order 
    * of variables in the name of the incoming Packager vertex. 
    * eg: If the vertex has name Packager::?k,?d,?z then the subject 
    * list 
    * will contain the variables in the ?k,?d,?z. We pick the 
    * corresponding index. 
    */ 
    String[] inpVars = 
        inputVerName.split("::")[1].split(",");
    int index = 0;
    for(; index <inpVars.length; index++) {
        //log.info("srcList: " +srcList);
    }
    } catch (Exception e) {
        log.error(e.getMessage(), e);
    }
    }
}

TGWritable newTGWritable 
    = new 
    TGWritable(Lists 
            .newArrayList(), srcList,
            Lists.newArrayList());

    kvWriter.write(new
            Text(),
            newTGWritable);

    //log.info("srcList: " +srcList);
}
if(inpVars[index].equals(srcVar.trim())) {
    break;
}

try {
    while (kvReader.next()) {
        List<Text> srcList =
            Lists.newArrayList();
        TGWritable tgWritable = (TGWritable)
            kvReader.getCurrentValue();
        List<Text> tgSubjsList =
            tgWritable.subjList;
        List<List<Set<Text>>> tgObjsList =
            tgWritable.objsList;

        srcList.add(tgSubjsList.get(index));

        TGWritable newTGWritable = new
            TGWritable(Lists.newArrayList(),
                srcList, Lists.newArrayList());

        kvWriter.write(new Text(),
            newTGWritable);

        //log.info("srcList: "+srcList);
        //log.info("tgSubjsList: "+tgSubjsList);
        //log.info("tgObjList: "+tgObjList);
    }
}
}

try {
    while (kvReader.next()) {
        List<Text> srcList =
            Lists.newArrayList();
        TGWritable tgWritable = (TGWritable)
            kvReader.getCurrentValue();
        List<Text> tgSubjsList =
            tgWritable.subjList;
        List<List<Set<Text>>> tgObjsList =
            tgWritable.objsList;

        srcList.add(tgSubjsList.get(index));

        TGWritable newTGWritable = new
            TGWritable(Lists.newArrayList(),
                srcList, Lists.newArrayList());

        kvWriter.write(new Text(),
            newTGWritable);

        //log.info("srcList: "+srcList);
        //log.info("tgSubjsList: "+tgSubjsList);
        //log.info("tgObjList: "+tgObjList);
    }
}

} catch (Exception e) {
    log.error(e.getMessage(), e);
}
}
void getObjSrc(String inputVerName, KeyValueReader kvReader, 
   KeyValueWriter kvWriter) {

    String[] inpVars =
        inputVerName.split("::")[1].split(",");
    int index1 = 0;
    int index2 = 0;
    int index3 = 0;
    for(; index1 <inpVars.length; index1++) {
        if(inpVars[index1].equals(subName.trim())) {
            break;
        }
    }
    //log.info("Value of index: "+index1);

    for(String sub : subjObjListMap.keySet()) {
        if(sub.equals(subName.trim())) { //found the subject
            List<Set<String>> objList =
                subjObjListMap.get(sub);
            for(Set<String> objs : objList) {
                if(objs.contains(srcVar)) {
                    for(String obj : objs) {
                        if(obj.equals(
                            srcVar)) {
                            break;
                        }
                        index3++;
                    }
                    index2++;
                }
                break;
            }
            break;
        }
        index2++;
    }
    break;
}
try {
while (kvReader.next()) {
    List<Text> srcList = Lists.newArrayList();
    TGWritable tgWritable = (TGWritable) kvReader.getCurrentValue();
    List<List<Set<Text>>> tgObjsList = tgWritable.objsList;
    Set<Text> tgObjSet = tgObjsList.get(index1).get(index2);

    int counter = 0;
    for(Text obj : tgObjSet) {
        if(counter == index3) {
            srcList.add(obj);
        }
        counter++;
    }

    TGWritable newTGWritable = new TGWritable(Lists.newArrayList(), srcList, Lists.newArrayList());
    kvWriter.write(new Text(), newTGWritable);
} catch (Exception e) {
    log.error(e.getMessage(), e);
} }
}

package edu.ncsu.csc.coul.semstorm.query.operator;
import java.nio.file.Paths;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.Text;
import org.apache.log4j.Level;
import org.apache.tez.common.TezUtils;
import org.apache.tez.dag.api.UserPayload;
import org.apache.tez.runtime.api.LogicalInput;
import org.apache.tez.runtime.api.LogicalOutput;
import org.apache.tez.runtime.api.ProcessorContext;
import org.apache.tez.runtime.library.api.KeyValueReader;
import org.apache.tez.runtime.library.api.KeyValueWriter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
import edu.ncsu.csc.coul.pathquery.PSLoader;
import edu.ncsu.csc.coul.pathquery.QueryFilter;
import edu.ncsu.csc.coul.pathquery.QueryFilter.Constraint;
import edu.ncsu.csc.coul.pathquery.PathQueryProcessor;
import edu.ncsu.csc.coul.pathquery.util.Global;
import edu.ncsu.csc.coul.semstorm.storage.writable.TGWritable;
import edu.ncsu.csc.coul.semstorm.util.ObjectSerializer;

/**
 * @author Abhisha Bhattacharyya (abhatt22@ncsu.edu)
 */
public class TGPathCompProcessor extends SimpleMRProcessor {
    private static final Logger log =
        LoggerFactory.getLogger(TGPathCompProcessor.class);
    private Configuration conf;
    private ObjectSerializer serializer;
    private List<Text> srcDstList;
    String direction;
    String inputPath;
    String interimPath;
    String constOption;
    String nodeTypeMap;

    @SuppressWarnings({ "unchecked", "static-access" })
    public TGPathCompProcessor(ProcessorContext context) {
        super(context);
        try {
            UserPayload userPayload =
                getContext().getUserPayload();
            this.conf = TezUtils.createConfFromUserPayload(
                userPayload);
            String serSrcDstList = conf.get("SrcDstList");
            srcDstList = Lists.newArrayList();
            List<String> srcDstListTemp = (List<String>)
                serializer.deserialize(serSrcDstList);
            for (String srcDst: srcDstListTemp)
                srcDstList.add(new Text(srcDst));
            String serDirection = conf.get("Direction");
            direction = (String)
                serializer.deserialize(serDirection);
            String serInputPath = conf.get("InputPath");
        } catch (Exception e) {
            log.error("Error occurred in TGPathCompProcessor", e);

    // More code...
}
inputPath = (String)
→ serializer.deserialize(serInputPath);
String serInterimPath = conf.get("InterimPath");
interimPath = (String)
→ serializer.deserialize(serInterimPath);
String serConstOption = conf.get("ConstOption");
constOption = (String)
→ serializer.deserialize(serConstOption);
String serNodeTypeMap = conf.get("NodeTypeMap");
nodeTypeMap = (String)
→ serializer.deserialize(serNodeTypeMap);

} catch (Exception e) {
    log.error(e.getMessage(), e);
}
}

@Override
public void run() throws Exception {
    Map<String, LogicalInput> inputs = getInputs();
    LogicalOutput output =
→ getOutputs().values().iterator().next();
    KeyValueWriter kvWriter = (KeyValueWriter)
→ output.getWriter();
    Set<String> srcList = Sets.newHashSet();
    Set<String> dstList = Sets.newHashSet();
    Set<String> cndList = Sets.newHashSet();
    try {
        //log.info("srcDstList: "+srcDstList);
        kvWriter.write(null, srcList);
    } finally {
        //log.info("srcDstList: "+srcDstList);
    }
for (Map.Entry<String, LogicalInput> entry :
inputs.entrySet()) {
    String inputVerName = entry.getKey();

    LogicalInput input = entry.getValue();
    KeyValueReader kvReader =
    (KeyValueReader)input.getReader();

    while(kvReader.next()) {
        List<Text> srcdstList =
        Lists.newArrayList();
        TGWritable tgWritable =
        (TGWritable)kvReader.getCurrentValue();
        srcdstList.addAll(tgWritable.subjList);
        kvWriter.write(null, srcdstList);

        if(inputVerName.contains("Source")) {
            for(Text src: srcdstList)
                srcList.add(src.toString());
        } else if(inputVerName.contains("Destination")) {
            for(Text dst: srcdstList)
                dstList.add(dst.toString());
        } else if(inputVerName.contains("Condition")) {

for (Text cnd: srcdstList)
    {
        cndList.add(cnd.toString());
    }

log.info("Final srcList: "+srcList);
log.info("Final dstList: "+dstList);
log.info("Final cndList: "+cndList);

String[] srcs = new String[srcList.size()];
String[] dsts = new String[dstList.size()];
String[] cnds = new String[cndList.size()];

srcs = srcList.toArray(srcs);
dsts = dstList.toArray(dsts);
cnds = cndList.toArray(cnds);

Constraint constOptionParam = Constraint.ALL;
switch (constOption) {
    case "all": constOptionParam =
        ~ Constraint.ALL; break;
    case "any": constOptionParam =
        ~ Constraint.ANY; break;
    case "none": constOptionParam =
        ~ Constraint.NONE; break;
    default:
        log.error("Cannot recognize the
        ~ constraint option: " +
        ~ constOption);
        return;
QueryFilter propFilter = null;

if (cnds.length > 0) {
    propFilter = new QueryFilter(constOptionParam, cnds);
}

log.info(nodeTypeMap);
Global.logger().setLevel(Level.DEBUG);

QueryProcessor qp = new QueryProcessor();
qp.getAllPathsInMemory(direction, srcs, dsts, propFilter, null, PSLoader.getPS(),
⇔ PSLoader.getContextPathSequence(), PSLoader.getNodeTypeMap(), "F");

} catch (Exception e) {
    log.error(e.getMessage(), e);
}

}