Subgraph Pattern Matching For Graph Databases

by

Sumana Venkatesh

(Under the Direction of John A. Miller)

Abstract

With the rapid increase of data in social and biological networks, ranging from a few dozen terabytes to many petabytes, managing the data with traditional databases has become very difficult. New data storage platforms have arisen to overcome lag in performance and capability from conventional approaches, which are built on traditional database technologies. Graph Databases have gained increased popularity when dealing with storage and processing of huge data with relationships. With the need to query these graph databases, fast and efficient graph pattern matching algorithms are required, to find exact and inexact matches. This thesis presents a new graph database that allows user to easily construct queries and run against huge vertex and edge labeled data graph. The database has a rich user interface, which is implemented using JavaFX and it uses fast pattern matching algorithms for subgraph isomorphism problem to get desired matches. It has an added functionality to perform pattern matching using regular expressions.

Index words: Graph Database, Query Builder, Subgraph Pattern Matching, Regular Expressions

Subgraph Pattern Matching For Graph Databases

by

Sumana Venkatesh

B.E, Visvesvaraya Technological University, India, 2009

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

Master of Science

Athens, Georgia

2014

©2014

Sumana Venkatesh

Subgraph Pattern Matching For Graph Databases

by

Sumana Venkatesh

Approved:

Major Professor: John A. Miller

Committee: Lakshmish Ramaswamy
Liming Cai

Electronic Version Approved:

Julie Coffield
Dean of the Graduate School
The University of Georgia
December 2014

Subgraph Pattern Matching For Graph Databases

Sumana Venkatesh

December 15, 2014

Dedication

This thesis is dedicated to my husband Prashanth, for his constant support throughout my study. I would also like to dedicate to my family and friends for their love and support.

Acknowledgments

I would like to thank my major professor Dr. Miller for his constant support throughout my thesis work. He has helped me to understand concepts and has given guidance whenever required. I would like to thank him for all the database concepts that he has taught that helped me in doing this research work. I would like to extend my gratitude to Dr. Ramaswamy for the knowledge that he has provided through his lectures on Distributed Systems and would like to thank Dr. Cai for teaching algorithms in the best way possible. I would like to thank all the professors for their guidance and support towards my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Literature Survey

Graph Databases have gained increased popularity in recent times due to the way they stores data and relationships between them. They are built with the underlying concept of graph theory. Information about an entity is stored within nodes and the edges that connect the nodes, defines a relationship between them. Graph databases are widely used for data storage in social networks, recommendation systems, networking, protein-protein interaction, transportation and many more. Data needs to be easily accessed, with all the required information such as details about a person as well as relationship with other friends or information about a product along with other products that come in the same category. Storing and retrieving such data can be critical and working with traditional databases to get such data with relationships can be challenging.

With huge amounts of data being stored in graph databases, we are dealing with complex data analysis and there is a need to find exact and inexact matches of small patterns called query graph in larger graphs called data graphs. This is termed as graph pattern matching. However, the problem of Subgraph Isomorphism where we need to find the exact match for the query in the data graph, i.e., structurally and semantically, is NP-Complete. Many algorithms have been proposed such as Graph Simulation [19], Dual Simulation [1], Strong

Simulation [6], Strict Simulation [7] and the most recently Tight Simulation [7] to find graph patterns in large labeled graphs.

1.1 Graph and Its Terminology

Given below are some of the concepts that define Graph and its related terminologies.

1.1.1 Graph Data Structures

A graph data structure comprises of a finite set of nodes or vertices along with set of edges. Based on how they are connected, it can be categorized into 3 types:

Simple Graph

A simple graph is a graph that has an ordered pair of vertices and edges denoted by $G = (V, E)$. A simple graph is an undirected graph without any loops or multiple edges. This graph can be connected or disconnected.

Directed Graph

A simple graph is a graph that has an ordered pair of vertices and edges denoted by $G = (V, E)$ where $E \subseteq VXV$ and each edge has a direction associated with it. Each edge starts from one V and points towards the other V in the pair.

Hyper Graph

It is a graph where the edges can connect any number of nodes or vertices. It is often denoted as $H = (V, E)$ where V is a set of vertices and E is a set of edges called as hyperedges.

Nested graph

A nested graph is the one where graph is contained by another graph and so on, i.e., it can be nested. Each node can itself expand into a graph.

1.1.2 Traversal and Indexing

A Traversal is visiting all the nodes in a graph, navigating from a start node to other nodes that have a relationship based on an algorithm or visiting all the nodes in a graph in a particular order, along with obtaining the value of each node or updating some information. It can identify the path along the way.

An Index maps from properties to either nodes or relationships. It helps to look up vertices. Often, we want to find a specific node or relationship according to a property it has.

1.1.3 Graph Query Languages

Graph Databases have their own often SQL like query language to access data from the database. Some of the popular, highly used, graph query languages are the following:

Cypher

Cypher[1] is the query language used by Neo4j database. It is expressive and clear while updating or retrieving data from the database. It is a powerful language and complex queries can be executed with ease. Cypher uses an SQL like structure where certain keywords are being reused and expressions for pattern matching are inspired by SPARQL, which came earlier. Cypher allows creation, deletion and updating of a database, which is applicable to nodes and relationships. An example is given below:

[1]http://neo4j.com/docs/stable/cypher-query-lang.html

$$\text{MATCH } (a) - [r] \rightarrow (b)$$

$$\text{RETURN } a;$$

where a is the start node, b is the end node and r is the relationship defined on them.

Gremlin

Gremlin[2] is a popular graph traversal language. TinkerPop maintains Gremlin. Gremlin is used in graph queries and is extremely useful when there are high levels of traversals. It supports Java and Groovy and all other JVM languages. Gremlin will work for any framework or graph database that implements the Blueprints data model. Blueprints model is something similar to JDBC but particularly intended for graph databases. An example is given below:

$$g.V('name','Tom').out('father').age$$

where g is the graph on which we are querying, V('name','Tom') get all the vertices with the name 'Tom', out('father') get all the outgoing edges with father property from Tom and age property get the age for Tom's father.

SPARQL

SPARQL[3] stands for SPARQL Protocol and RDF Query Language. It is a graph query language that is used to query data that is stored in RDF format. It queries RDF graph using pattern matching. The query is in the form of a triple, i.e., Subject-Predicate-Object format. The data are often stored in a TripleStore, which is a database to store data in triple format. An example[4] is given below:

[2]https://github.com/thinkaurelius/titan/wiki/Gremlin-Query-Language
[3]http://www.w3.org/2009/sparql/
[4]http://www.w3.org/2009/Talks/0615-qbe/

@prefix card: <http://SomedatasetURL >

PREFIX foaf: <http://xmlns.com/foaf/0.1/ >

SELECT ?name

WHERE { ?person foaf:name ?name .}

The variables that start with a ? can match any node in the given dataset. The SELECT clause gives the result in the form of a table with variables and values that match the query. Any part of the triple can be replaced by variables.

1.1.4 Comparison of Graph Database and Relational Database

Relational databases have been around for several decades and are been used extensively for data storage and retrieval. They use the Structured Query Language (SQL) as a query language for data retrieval. In the recent years, an alternate called the NoSQL has come to light. Examples are Google's BigTable, Facebook's Cassandra, and Amazon's Dynamo, etc. Neo4j is also one such NoSQL graph database.

A NoSQL database may be more suitable versus a relational database if

- Tables have many columns, each of which is used by only few of the rows
- There are several many to many relationships.
- Tables require frequent schema changes.

1.1.5 Graph Database and Subgraph Pattern Matching Models

Graph Databases and subgraph pattern matching algorithms has been a focus of study in recent times. This is mainly due to the huge amount of data that has been available on the Web. The current trend in the field of big data is NoSQL databases. This trend is deviating the focus from relational databases to graph databases that have more flexibility

when it comes to storing data with relationships. Many graph databases are available and each of them are built catering to a specific set of tasks. Since graph database stores data in the form of graph, many pattern matching models have arisen to come up with the best solution possible. Ullmann[5] is the first among the existing algorithms to tackle subgraph isomorphism problem. However, since this problem is NP-Complete, several algorithms have been proposed to reduce the computation time by relaxing some of the conditions. Some of these polynomial time algorithms are Graph Simulation [19], Dual Simulation [1], Strong Simulation [6], Strict Simulation [7] and most recently Tight Simulation [7]. Some of the libraries that are available that perform subgraph matching are igraph [15], nauty [16], vflib [17] etc.

Four of the existing graph databases Neo4j[5], Titan[6], OrientDB[7] and DEX[8], were compared in [18] for performance evaluation. For traversal workloads, Neo4j outperformed other graph databases. For read only intensive workloads, all four databases performed the same. For read-write workloads DEX and Titan-Cassandra outperform the other databases, Neo4j, Titan-BerkleyDB and OrientDB. Basically, every other database that is present out there has some good features and some flaws.

However, the existing graph databases do not have a system which has a user friendly query builder that has a simple interface to quickly build queries. Graphite [13] works on the same lines but it does not handle edge labels. An existing graph pattern matching model Dual Simulation has been used, to include edge label functionality that previously considered node labels only. Neo4j has some good features for viewing graph data and has its own query language to match and retrieve queries. Work has been done on regular expression on pattern queries [12], but none of the existing systems focus on all three of our research interests, an

[5]http://neo4j.com
[6]http://thinkaurelius.github.io/titan/
[7]http://www.orientechnologies.com/orientdb/
[8]http://www.sparsity-technologies.com

interface to build queries easily, focus on edge labeled graphs and perform pattern matching using regular expressions.

Chapter 2

An Interactive Graph Database To Perform Fast Subgraph Pattern Matching

[1]

[1]Sumana Venkatesh, Aravind Kalimurthy, John A. Miller, Matthew Saltz, to be submitted to 2015 IEEE International Congress on Big Data

Abstract

With the rapid increase of data in social and biological networks, ranging from a few dozen terabytes to many petabytes, managing the data with traditional databases has become very difficult. New data storage platforms have arisen to overcome lag in performance and capability from conventional approaches, which are built on traditional database technologies. Graph Databases have gained increased popularity when dealing with storage and processing of huge data with relationships. With the need to query these graph databases, fast and efficient graph pattern matching algorithms are required, to find exact and inexact matches. This paper presents a new graph database that allows users to easily construct queries and run against huge vertex and edge labeled data graphs. The database has a rich user interface, which is implemented using JavaFX and it uses fast pattern matching algorithms for subgraph isomorphism problem to get desired matches. It has an added functionality to perform pattern matching using regular expressions.

INDEX WORDS: Graph Database, Query Builder, Subgraph Pattern Matching, Regular Expressions

2.1 Introduction

Graph Databases are growing and are increasingly used for data storage and analysis. It models data as vertices, edges and properties. Information about an entity is stored within vertices. Vertices are connected by edges defining a relationship between them. Every node has an edge to its adjacent node which help to retrieve related data easily. Graph databases have acquired renewed interest and have been widely used for storage of large data in the world of social networks, recommendation systems, networking, bio-informatics [11], transportation and many more. Since graph databases allow us to analyze data and relationships in a better way, it is gaining much interest in the world of social media, which

deal with data from millions of users every single day. Some of the graph databases that have emerged in the recent times are Neo4j[2], Titan[3], OrientDB[4], InfiniteGraph[5], SparkSee[6], FlockDB[7], HyperGraphDB[8], AllegroGraph[9] and GraphBase[10]. Neo4j is a state of the art graph database and is commercially used in large scale.

With huge data being stored in graph databases, there is a need to query data frequently. This is known as query processing on large graphs. Several algorithms have been proposed to perform graph pattern matching where small patterns called query graphs are matched for subgraphs within large data graphs. However, the problem of finding exact matches for a query, structurally and semantically, known as Subgraph Isomorphism is NP-Hard. So in the recent years, many have developed graph simulation models that have polynomial time complexity. Some of the algorithms under this category are Graph Simulation [14], Dual Simulation [1], Strong Simulation [6], Strict Simulation [7] and most recently Tight Simulation [7].

While variants of graph pattern-matching algorithms have been proposed, there is a lack of systems, which lets a user to easily build query graphs, run any of the high performance algorithms to find matches and analyze the results with minimal effort. The existing systems also lack the functionality and use of regular expressions in query graphs which can be highly useful. Previously, work has been done in this area by adding regular expressions to edge labels in reachability queries and pattern graphs [12].

In our research, we have aimed to overcome the above challenges with our graph database that can store graph data as well as provide an interface to construct queries and run against a

[2]http://neo4j.com
[3]http://thinkaurelius.github.io/titan/
[4]http://www.orientechnologies.com/orientdb/
[5]http://www.objectivity.com/infinitegraph
[6]http://www.sparsity-technologies.com
[7]https://github.com/twitter/flockdb
[8]http://www.hypergraphdb.org/index
[9]http://franz.com/agraph/allegrograph/
[10]http://graphbase.net

data graph and know the possible matches in the data graph using efficient pattern matching algorithms. This helps the users to analyze patterns and relationships better. We have also introduced the concept of graph pattern matching using regular expression that facilitates ease of use while building queries. The search pattern or regex specified in the query graph is matched with a label in the data graph to get results. Edge label functionality has been added to the existing Dual Simulation [1] algorithm, which previously worked only on node labeled graphs. Using the database, the user can:

1. Construct a query pattern by dragging nodes
2. Connect nodes by edges on mouse click
3. Assign a label to each node via colors
4. Define a regular expression pattern as a label that matches an entire string or label in the data graph
5. Assign an edge label
6. Run the query using one of the pattern matching algorithms to get matches
7. Export the constructed query graph to CSV format.

This paper is organized as follows. In Section II, we discuss the background stating how query processing takes place in general and also about existing pattern matching algorithms. Section III will be an introduction to our graph database, with details about how the user interface works. Section IV gives a detailed overview of the edge label functionality that is introduced in this paper. We discuss how the Dual Simulation algorithm works when edge label matching is added to it. We have also explained with an example on how the algorithm works to give results. Section V gives details about graph pattern matching using regular expressions, some details on the algorithm that Java uses for pattern matching and

advantages of using it. Section VI shows experimental results of performance of DualIso, with and without the edge label functionality. In the last sections, we talk about future work and conclusion.

2.2 Background

In this section, we discuss graph terminology, query processing and several of the existing pattern matching algorithms. Throughout this paper, we have considered a directed graph, with vertex label and edge label. A vertex and edge labeled directed graph may be defined as $G(V, E, L, l)$, where

V = set of vertices

$E \subseteq V \times V \times L$ (set of labeled edges)

L = set of labels

$l : V \rightarrow L$ (vertex labeling function)

Note, the base type of set L can be set at configuration time. The system has been tested with label types of integer and string.

We denote outgoing edges for a vertex v as $adj(v)$ where for some vertex $v \in V$, $adj(v) = \{v' : (v, v') \in E\}$. We sometimes refer the vertices in $adj(v)$ as children of v and also refer to v as the parent of all the vertices in $adj(v)$.We have assumed that all the vertices and edges are labeled. We also assume that the query graph is a connected graph, as a disconnected graph would mean having two different queries for the same data graph. We have used the words patterns and queries interchangeably [8].

2.2.1 Query Processing on Graph Databases

Before stepping into graph pattern matching problems, we need to know how query processing takes place on graph databases. Query processing is a problem of finding small patterns or subgraphs on a graph database having a set of graphs. Processing graph data can be a complex task and hence it requires efficient algorithms that can process query graphs on graph databases. According to [9], based on how data is stored, graph databases can be grouped into two types. In the first type, the graph database can have very large graphs like Web of Data [10], social networks, etc. Query processing for such a database would be finding the optimum path between the vertices or finding a subgraph that is similar to the query graph, which we will be focusing in our research. The second type of database consists of a large set of smaller graphs. An example of this type would be in the field of bio-informatics [11]. Queries for such a database, involve finding a graph similar to the query graph or finding a graph having a subgraph similar the query graph.

Query processing in general can be categorized into two important steps - filtering and verification [14].

1. In the filtering phase, the query graph is decomposed into features and these features are later searched using an index. Each feature, represented by ID's is searched to get a set of graphs. The set of graphs are intersected to get candidate sets.

2. In the verification phase, the set of graphs are matched using a subgraph isomorphism algorithm to obtain final result set.

2.2.2 Subgraph Pattern Matching Problem

The problem of subgraph pattern matching is defined as follows.
Let $G(V, E, L, l)$ be a graph, where V is the set of vertices, E is the set of edges, $l : V \rightarrow L$ (vertex labeling function). Let $Q(V_q, E_q, L_q, l_q)$ be the query graph where V_q is the set of

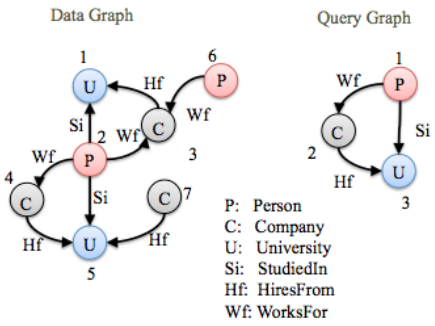

Figure 2.1: A possible data graph and query graph

vertices, E_q is the set of edges and $l_q : V_q \rightarrow L_q$. The goal of subgraph pattern matching is to find all the subgraphs from the data graph G that match the pattern graph Q. Therefore, $G'(V', E', L', l')$ is a subgraph of G if and only if

1. $V' \subseteq V$
2. $E' \subseteq E$ and $E' \subseteq V' \times V' \times L'$
3. $\forall u \in V' : l'(u) = l(u)$

2.3 Types of Pattern Matching

In this section, we discuss the different pattern-matching problems. This gives a basic idea on how each of them works in order to achieve matches for a given pattern.

2.3.1 Subgraph Isomorphism

Subgraph Isomorphism can be defined as a bijective mapping between a query graph $Q(V_q, E_q, L_q, l_q$ and a subgraph of a data graph $G(V, E, L, l)$. Thus $G'(V', E', L', l')$ is said to be a subgraph isomorphic match to Q if

1. $V' \subseteq V$

2. $E' \subseteq E$ and $E' \subseteq V' \times V' \times L'$

3. there exists a bijective function $f : V_q \rightarrow V'$ such that

   (a) A labeled edge $e = (u, v, \lambda) \in E_q \iff (f(u), f(v), \lambda) \in E'$

   (b) $\forall v \in V_q, l(v) = l'_v(f(v))$

The problem of finding all the subgraphs that are isomorphic to a query graph is NP-Hard. Ullmann developed first well known algorithm for subgraph isomorphism [5]. It laid a foundation for other pattern matching algorithms. There has been much research going on, to reduce the complexity to polynomial time and some of them will be discussed below.

2.3.2 Graph Simulation

Graph Simulation allows a quicker alternative to subgraph isomorphism by relaxing some restrictions. Query graph $Q(V_q, E_q, L_q, l_q)$ matches data graph $G(V, E, L, l)$, if there exists a binary relation $R \subseteq Vq \times V$ such that

1. for every $u \in V_q$, $\exists u' \in V$, such that $(u, u') \in R$

2. $l_q(u) = l(u')$

3. for every $v \in child(u)$, there is a $(v, v') \in R$, such that $v' \in child(u')$ [20]

2.3.3 Dual Simulation

Dual Simulation is an extension to graph simulation model, but has some additional features. It takes into account not only the children of the query vertex but also its parents. Query Graph $Q(V_q, E_q, L_q, l_q$ matches data graph $G(V, E, L, l)$, if

1. Q is a graph simulation match to G with a match relation $R_D \subseteq Vq \times V$, and

2. for every $w \in parent(u)$, there is a $(w, w') \in R$ such that $w' \in parent(w')$ [20]

The memory taken by this algorithm is two times lesser than the existing ones and gives efficient and fast results as both parent and child constraints are checked within a single loop. The complexity of the algorithm is Cubic time.

2.3.4 Dual Iso

DualIso is also known as Dual based Isomorphism. This algorithm is faster and similar to Ullmann's subgraph isomorphism algorithm but provides more effective pruning based on Dual Simulation. It initially finds all the feasible matches of each vertex in the query graph to a set of vertices in the data graph solely based on label match. Then dual simulation is applied to prune the vertices from the data graph. It then uses a search algorithm to recursively find the matches in a depth-first manner. For each traversal, dual simulation is applied. This change in the approach make the algorithm to run much faster. [8]

2.3.5 Graph Iso

The Graph Isomorphism problem is to find whether two labeled directed graphs are Isomorphic. A query graph $Q(V_q, E_q, L_q, l_q)$ can be said to be a graph isomorphic match to a data graph $G(V, E, L, l)$ if

1. $|V_q| = |V|$

2. there exists atleast one subgraph isomorphic match for Q in G

For a more conventional definition of graph isomorphism see [21].

2.4 About Graph Database

Our graph database is built using JavaFX 8, which provides a rich graphical interface. As JavaFX[11] is built on top of native Java code, it can call any API from the Java library and can be seamlessly integrated into the code. A latest stable release of Java SE 8 is required to run this tool. We used Eclipse Luna IDE with JDK 8 to implement the code. All the algorithms are written in Java and Scala. We have also used Kryo[12], a fast and efficient object graph serialization for Java framework to store and retrieve data graphs.

2.4.1 Architecture

This section gives an overall picture of how our database works. Figure 2.2 shows the architecture of our system. The key aspects of the system are the following:

1. The starting point of the application starts with loading of a basic JavaFX stage. It is a top level container classes for JavaFX content.

2. The stage can hold one (or more) Scene, which has the base classes of the Scene Graph API. This is the starting point for construction of any JavaFX content. This is a node in itself and can hold any number of nodes within. Some examples of JavaFX nodes are shapes (circle, rectangle, line, etc.), text, images, tables, media, etc. The scene graph holds the state of every single node within itself. This includes the position of the nodes in terms of x and y coordinates (in case of 2D), its orientation and visual effects. Visual effects like blur, shadow, glow, etc. can be added to any node. Additional functionality of dragging the nodes can be added by translating the initial coordinates to the new position.

[11] http://docs.oracle.com/javase/8/
[12] https://github.com/EsotericSoftware/kryo

3. We can call any Java public API's from the scene. This allows us to us all the functionalities provided by Java.

4. A border layout is added to the application on top of the scene. This layout has a panel on top of which vertices can be dragged and query graphs can be constructed.

5. The system takes in as input, a file for data graph with Comma Separated Values that has vertices, edges, vertex labels and edge labels. Once the file is loaded, the database stores the file in a serialized object at the location */var/mydb/*. It is later deserialized when the query is executed against it.

6. Once the query is constructed, the available pattern matching algorithms can be selected for subgraph pattern matching. The algorithms return results that are displayed using JavaFX TableView.

2.4.2 User Interface and Features

The GUI is structured into four components : Toolbox, Infobox, Main and Results. The Toolbox contains the tools to build query graphs. It has a node/vertex and an edge. The Infobox has options to load a data file, select a pattern-matching algorithm, run the query against a selected datagraph and export the query graph to a CSV file. The Main is a panel where the query graph can be constructed. The Results area shows the time taken to get matches. The matches are displayed in a table view.

To insert data into the database, the data has to be inserted through a file. The file should be in CSV format, which is then converted to a data graph on upload. The contents of the graph is serialized using Kryo [14], i.e., the object is converted into bytes and then saved to a file in a pre-defined location in the system or server. When the user later selects

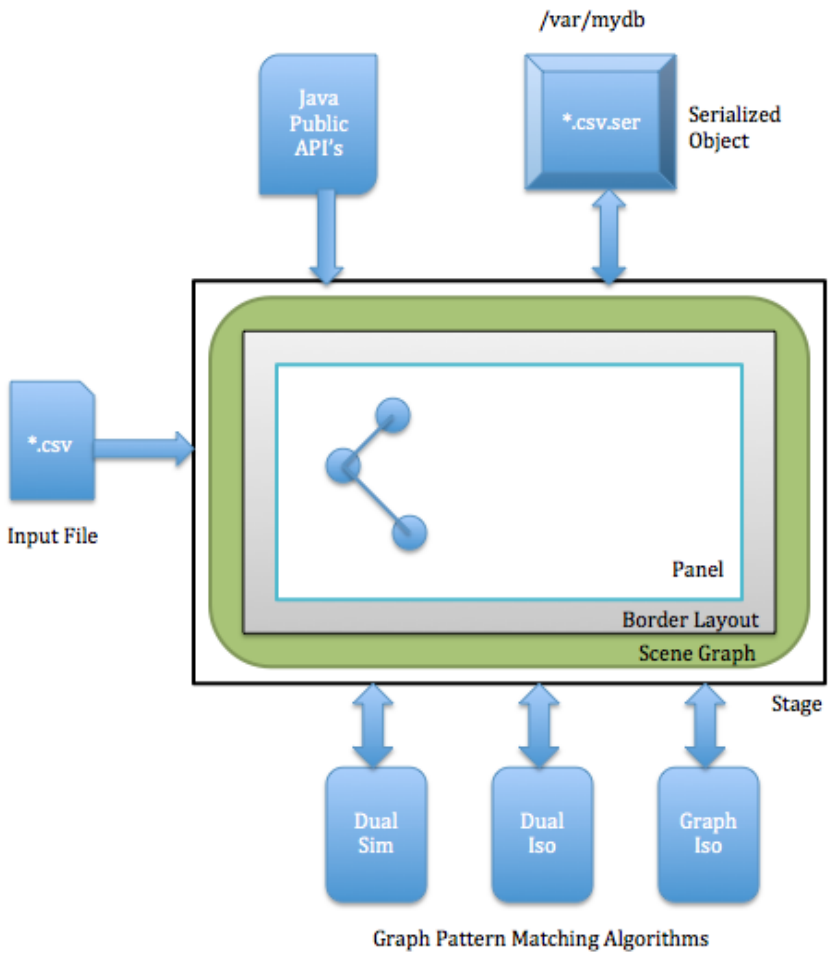

Figure 2.2: The Architecture of our Graph Database

the data file to run the query against, the file is retrieved back by a process of deserialization. Kryo serialization is one of the fastest and most efficient ways to serialize data.

2.4.3 Functionality

A new node is added by dragging the primary node from the Toolbox on to the Main, which is a Panel. The node is represented as a circle with the vertex number in the center. The vertex number auto increments as the nodes are dragged on to the Main Panel. Each node

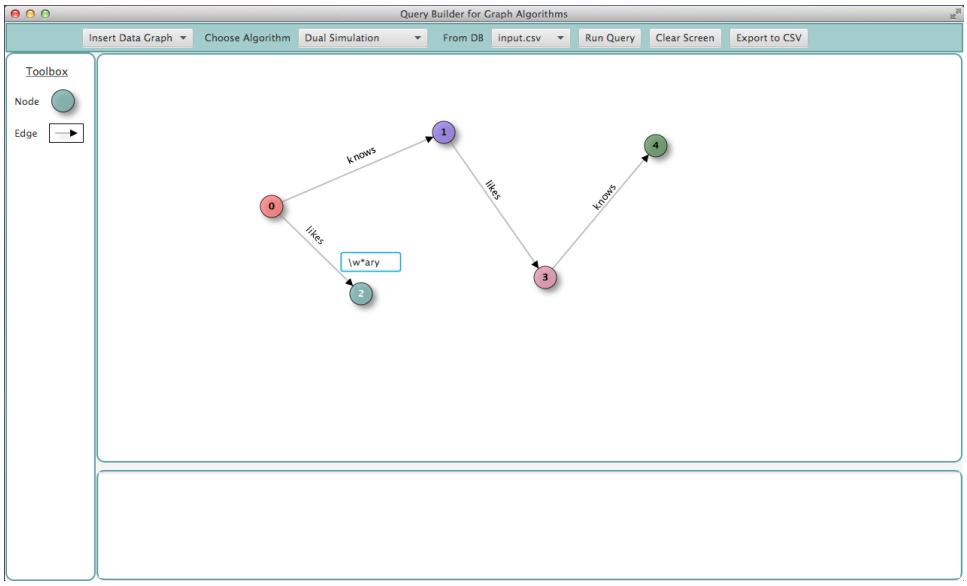

Figure 2.3: The user interface of graph database

can be dragged anywhere on the main screen which helps in alignment of the graph.

An edge connects two nodes defining a relationship between them. A line represents an edge with an arrowhead directing towards another node. To draw an edge, we need to click on the edge in the Toolbox and then click on the node where the edge is bound to start and then click on the node, which is the end. This creates a line between them. The two end points of the line are bound to the center of two circles. As the node is dragged, the end point of the line associated with the node also translates to give a smooth transition visually.

The Label for each vertex is uniquely identified by a color. There are 120 colors chosen from the spectrum as default colors where each color is mapped to a number. When a node is dragged onto the screen, a default label is associated with that node. To edit the label, right click on the node and select 'Edit Label' from the context menu. On click, a color picker opens up, displaying all 120 colors out of which a single color can be chosen at a time. On hover, the number associated with the color is displayed. The selected color is then added to the node, which defines a new label to the node. Note that this functionality of mapping

color to a label works only from 0 to 119 colors. If the query vertex has an integer greater than this or has a string as its label, the value can be entered via textbox on edit. The label can also be a regular expression, which is discussed in detail in Section IV.

Each edge in the GUI is given a default value when added to the user interface for building a query graph. To add an edge label, the user has to right click on the edge to edit the label and enter the string. This defines the relation of the vertex where the edge starts to the vertex where the edge ends. Once the edge is added, it considered as a candidate to be matched with an edge in the data graph. An example query graph constructed can be seen in Figure 2.3.

2.5 Edge Label in Dual Simulation

Dual simulation [1] is an improvement to the graph simulation [20] model. While matching query graphs to data graphs, it considers children of a query vertex as well as its parents to yield better results. However, this pertains to only vertex labeled graphs. Inspired by SPARQL [3] triple pattern, we added edge labeled functionality to Dual Simulation [1], where an entity comprises of subject-predicate-object. The edge is functionally mapped to a label. A simple example in social networks would be 'Alex knows Ted' where Alex is the subject, knows is the predicate and Ted is the object. The predicate defines a relationship between the subject and the object.

The algorithm to find matches for an edge label starts by running Dual Simulation on the query graph and data graph. Once the graphs are loaded, a label map is constructed for each of them, which is a HashMap that maps each label to sets of vertices containing those labels. Query vertices are iterated over to find feasible mates in the data graph. This procedure starts by matching the label for each query vertex with the label from the labelMap for data graph to get set of vertices for the matched label.

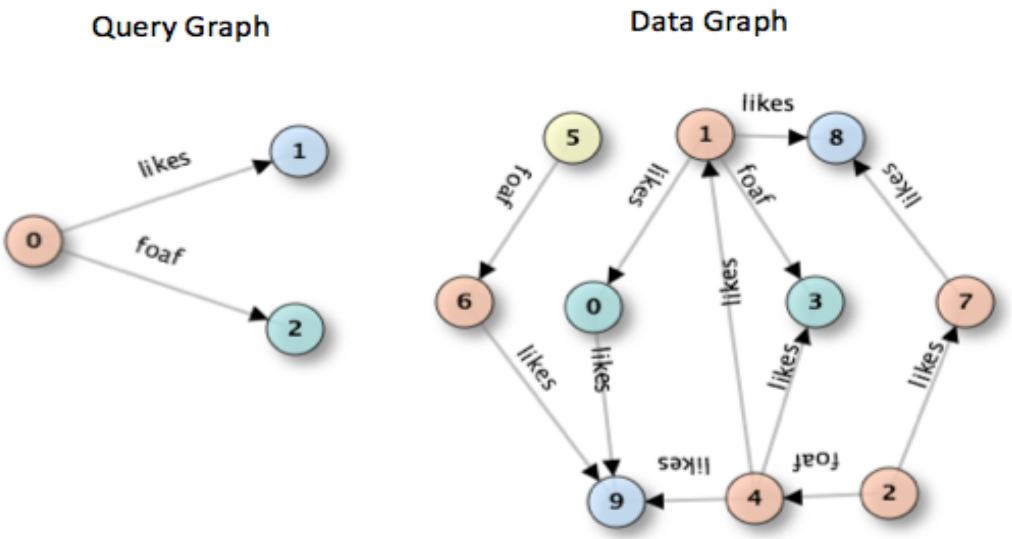

Figure 2.4: An example data graph and query graph

After getting the feasible mates from step 1, Dual simulation algorithm is applied on the feasible mates to find mappings from the query graph Q to data graph G. These are represented as multi-valued function '*phi*' that maps each query graph vertex 'u' to a set of data graph vertices '$\{v\}$'. This procedure starts by eliminating mapping '$u \rightarrow v$' when

1. v's children fails to match u's children
2. v's parents fail to match u's parents
3. an edge label on an edge from $v \rightarrow v'$ fail to match an edge label on an edge from $u \rightarrow u'$.

In the example shown for the data graph and query graph in Figure 2.4, all the vertices are denoted with a number. The data graph has 10 vertices nad query graph has 3 vertices and each vertex has an associated label which is represented with a color. The labels for all the data graph vertices starting from 0 to 9 are

$$\{1, 2, 2, 1, 2, 0, 2, 2, 3, 3\}$$

and the labels for query graph vertices starting from 0 to 2 are

$$\{2, 3, 1\}$$

The edges that have a property are denoted as edge label.

To understand how the algorithm operates, consider a query graph, which is to be checked for matches in the given data graph in Figure 2.4. The steps that take place are given below:

1. In the initial step, the query vertices $\{0, 1, 2\}$ are iterated over and each vertex label is checked in the label map if there is a match. Every match returns a set of integers that are the corresponding matched vertices in the data graph. The feasible mates returned from step one are shown in Table 2.1.

   Table 2.1: Feasible Mates in Dual Simulation

Query	Data
$\phi(0)$	$\{1, 2, 4, 6, 7\}$
$\phi(1)$	$\{8, 9\}$
$\phi(2)$	$\{0, 3\}$

   This step eliminates vertex 5.

2. Once we have the mappings ϕ from step 1, the dual simulation algorithm starts pruning vertices if it fails to match child, parent or edge label criteria. This is carried out by checking for the children of the vertices obtained from Step 1. Each vertex has a set of Labels which holds the edge information. The Label class has the vertex where the edge ends and the corresponding edge label in it. While intersecting the vertices of the query and data graph, the label class is iterated over to get vertices and edge labels. Both the vertices from the query graph and data graph are checked, and if they have the same node label, then the edge label is matched. Once there is a match, it is added

to a result set and if not, it is removed from the set of possible matches. In the first iteration vertices $\{2, 4, 6, 7\}$ are removed from $\phi(0)$.

3. Then the search is continued by creating a new copy of ϕ is and the algorithm start searching within this ϕ'. If ϕ' has no vertices, then the algorithm keeps backtracking until it finds the vertices that meet all the criteria. As vertex $\{9\}$ loses its parent and edge label match and vertex $\{0\}$ loses the edge label match, these vertices are removed from ϕ'.The matched vertices are added back to ϕ.

4. Finally ϕ has all the matches and the results are shown in Table 2.2

Table 2.2: Results obtained after Dual Simulation

Query	Data
0	1
1	8
2	3

2.6 Pattern Matching Using Regular Expressions

A regular expression, also termed as regex, is a sequence of characters that forms a search pattern, used to match a target string or multiple occurrences of the string. It is a technique that is being used in a search engine to find pages from the web, in a word processor to find a string literal in a file or to find and replace a string, to extract a specific segment from a html file, etc. For example, if we want to find a person in the data graph where the firstname starts with 'Tim', a regular expression that can be added to the query vertex can be '$\backslash bTim\backslash w*\backslash b$'. Regular expressions can be used to tackle many complicated searches that may involve finding email formats, using special characters, a search string with alpha numeric values and also a query with minimal characters to find the entire string. Many quantifiers are available that help to find how often an expression occur.

In our algorithm, we are using the java.util.regex package provided by Java for regex pattern matching. The Java library has a pattern class and a matcher class. The pattern class accepts a regular expression as one of its arguments. It also checks if the sequence of characters forms a valid regular expression. Then the regex is applied on the input text from left to right. The matcher class depicts the specified pattern and performs a match against a given input string. Java also supports several special characters known as metacharacters. These characters affect the way that the pattern matches. Java performs NFA (Nondeterministic Finite Automaton) based regular expression pattern matching. NFA is said to be memory efficient but not necessarily time efficient. In this context, DFA (Deterministic Finite Automaton) is said to be better. For n number of states, NFA has complexity $O(n)$ to perform transition table lookups in order to process each input symbol. [12]

A data graph can have millions of nodes, each of them having labels that represent an entity. While forming a pattern for the query graph, a label can be represented as a sequence of characters or regex that can be used as a search pattern to match labels in the data graph. This can be highly useful when the exact label in the data graph is unknown. These labels can be node labels or edge labels. When the algorithm starts comparing the query with the data graph, a matcher checks if the regular expression specified in the query matches the string in the data. If so, the node will be added as a possible result to the pattern-matching problem.

2.7 Experimental Results

In this section we evaluate how Dual Iso[8] with edge label functionality performs. We have tested it against the Dual Iso without edge labels and also with the one with String labels. The original DualIso has labels as integer. Our goal was to have the same speed and efficiency as the one without edge labels. We have used synthetic graphs in our experiments. Factors

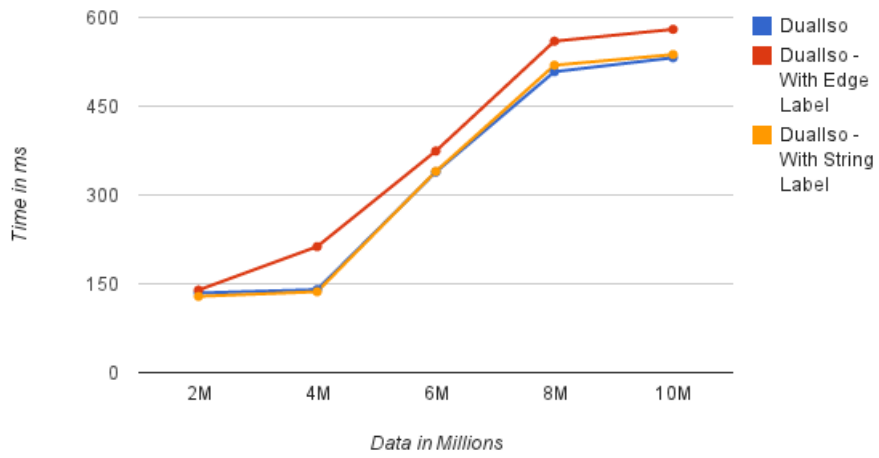

Figure 2.5: The average runtimes of Data Vertices from 2 to 10 million with Query Size=20, Labels=100, EdgeLabels=3, Maximum outdegree=2

that determine the performance are based on size of data graphs and size of query graphs. All our algorithms are written using Java.

To generate synthetic graphs, we have used our own graph generator code written in java. It constructs a graph with desired number of vertices based on parameters such as number of distinct node labels, number of distinct edge labels and maximum outdegree from a single vertex in the graph. A query graph is generated by using Breadth First Search (BFS) on the data graph to get a subgraph within the data graph for desired number of vertices.

All the experiments are run on a machine with two 2GHz Intel Xeon E5-2620 CPUs, each having six hyper-threaded cores for a total of 24 threads, and 128GB of DDR3 RAM. The implementation of DualSim, DualIso, and GraphIso are written in Java version 8.

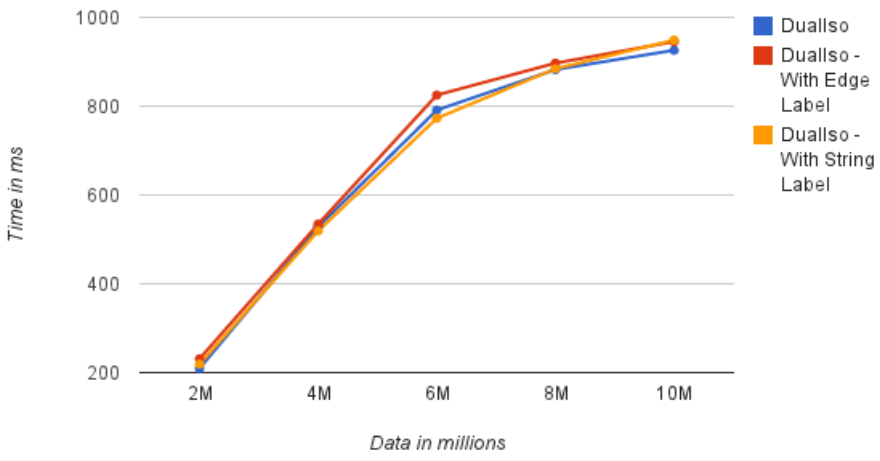

Figure 2.6: The average runtimes of Data Vertices from 2 to 10 million with Query Size=50, Labels=100, EdgeLabels=3, Maximum outdegree=2

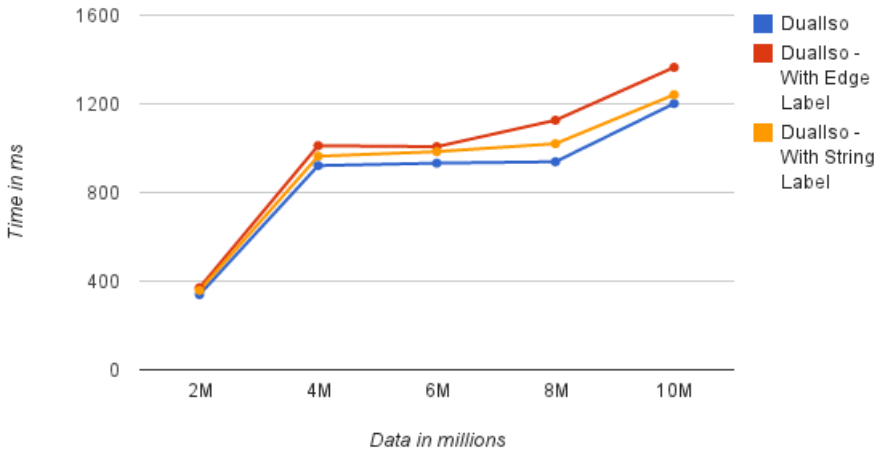

Figure 2.7: The average runtimes of Data Vertices from 2 to 10 million with Query Size=100, Labels=100, EdgeLabels=3, Maximum outdegree=2

2.7.1 Impact of Data Graphs and Query Graphs

We have tested DualIso with data sizes ranging from 2 million to 10 million. The graphs shown below have been tested for data sizes of given range with query sizes of 20, 50 and

100 vertices. We can observe that as the number of data vertices increases, the execution time also increases. Increase in the number of query vertices also increases the computation time. The comparison results are showin in Figure 2.5, 2.6 and 2.7 for query vertices 20, 50 and 100 respectively. We can see that the performance is quite close the original algorithm even after adding the edge label functionality.

2.8 Related Work

Graph Databases and subgraph pattern matching algorithms focus on diverse areas, including speed, scalability, ease of pattern matching, data storage and much more. We have categorized our survey based on three factors: (1) ease of use to build queries using an interactive interface for graph database; (2) edge labeled functionality that is existing now in almost all of social media; (3) usage of regular expressions for labels that is highly beneficial in pattern matching.

When it comes to building queries using an interface, Graphite [13] works on the same lines. It finds exact and approximate matching subgraph in a large attributed graphs and helps to visualize subgraphs. It uses G-Ray algorithm for pattern matching. But we wanted to integrate with our high performance algorithm to perform subgraph pattern matching.

Many algorithms have been proposed for subgraph pattern matching problem. Some are meant for exact matching and some are for inexact matching. We have extended our work from [8], to include edge functionality to an already efficient subgraph pattern algorithm for subgraph isomorphism problem that previously considered node labels only.

Work has been done on regular expression on pattern queries [12]. Some of the existing database models implement regular expression but none of the existing systems, to the best of our knowledge, focus on all three of the research interests as stated above.

2.9 Conclusion and Future Work

We have developed a prototype graph database that allows users to easily construct a query graph and get all the matches for the pattern in a large labeled data graph. We have also implemented pattern matching using regular expressions that matches the user provided regular expression to an entire label match in the data graph. Since Dual Simulation is proved to be highly efficient, we have used that as one of the subgraph pattern matching algorithms along with Dual Isomorphism and Graph Isomorphism. An additional functionality of edge labels has been added to Dual Simulation. We found that the rich graphical user interface provided by JavaFX [1] allowed us to create an interactive platform to build query graphs and usage of efficient algorithms make our database highly useful in the field of graph pattern matching for subgraph isomorphism problem.

Adding additional algorithms like Tight Simulation [3] with edge label functionality to the database can be a future work. Deletion of a single node and single edge can be added to the UI functionality. Currently, the database can add data graphs to the system through files that are in CSV format. Adding functionality to accept data manually or building query languages to insert, delete and update data can be future work for this graph database. Import and export using JSON can be added. Also variables can be added to the query graphs, in order to correlate certain vertices in the data graphs.

In terms of comparison, Dual Iso can be compared with another graph database with respect to Edge labels, Usage of regular expression and both. Comparison can also be made by using replacing current java.util.regex package to any other faster java regex package.

Chapter 3

Summary

We have presented a new graph database that allows users to easily construct a query graph and get all the matches for the pattern in a large labeled data graph. We have also implemented pattern matching using regular expressions that matches the user provided regular expression to an entire label match in the data graph. Since Dual Simulation is proved to be highly efficient, we have used that as one of the subgraph pattern matching algorithms along with Dual Isomorphism and Graph Isomorphism. An additional functionality of edge labels has been added to Dual Simulation. We found that the rich graphical user interface provided by JavaFX [1] allowed us to create an interactive platform to build query graphs and usage of efficient algorithms make our database highly useful in the field of graph pattern matching for subgraph isomorphism problem.

Adding additional algorithms like Tight Simulation [3] with edge label functionality to the database can be a future work. Deletion of a single node and single edge can be added to the UI functionality. Currently, the database can add data graphs to the system through files that are in CSV format. Adding functionality to accept data manually or building query languages to insert, delete and update data can be future work for this graph database. Import and export using JSON can be added. Also variables can be added to the query

graphs, which can be parsed to identify the label in the data graphs.

In terms of comparison, Dual Iso can be compared with another graph database with respect to Edge labels, Usage of regular expression and both. Comparison can also be made by using replacing current java.util.regex package to any other faster java regex package.

Bibliography

[1] Ma, Shuai, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. "Capturing topology in graph pattern matching." Proceedings of the VLDB Endowment 5, no. 4 (2011): 310-321.

[2] Yu, Fang, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. "Fast and memory-efficient regular expression matching for deep packet inspection." In Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, pp. 93-102. ACM, 2006.

[3] PrudHommeaux, Eric, and Andy Seaborne. "SPARQL query language for RDF." W3C recommendation 15 (2008).

[4] Livi, Lorenzo, and Antonello Rizzi. "The graph matching problem." Pattern Analysis and Applications 16.3 (2013): 253-283.R.

[5] Ullmann, Julian R. "An algorithm for subgraph isomorphism." Journal of the ACM (JACM) 23.1 (1976): 31-42.

[6] Fard, Arash, M. Usman Nisar, Lakshmish Ramaswamy, John A. Miller, and Matthew Saltz. "A distributed vertex-centric approach for pattern matching in massive graphs." In Big Data, 2013 IEEE International Conference on, pp. 403-411. IEEE, 2013.

[7] Arash Fard, M. Usman Nisar, John A. Miller, and Lakshmish Ramaswamy, "Distributed and Scalable Graph Pattern Matching: Models and Algorithms," International Journal of Big Data (IJBD), Vol. 1, No. 1, January-March 2014, pp. 1-14.

[8] Saltz, Matthew, Ayushi Jain, Abhishek Kothari, Arash Fard, John A. Miller, and Lakshmish Ramaswamy. "Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs." In Big Data (BigData Congress), 2014 IEEE International Congress on, pp. 498-505. IEEE, 2014.

[9] Cheng, James, Yiping Ke, and Wilfred Ng. "Efficient query processing on graph databases." ACM Transactions on Database Systems (TODS) 34, no. 1 (2009): 2.

[10] Auer, Sren, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. Springer Berlin Heidelberg, 2007.

[11] ZHANG, Hong-mei, Xin-bing YU, and Jin XU. "Screening of Calcineurin B-like Protein Gene, The Functional Gene of Adult Clo norchis sinensis by Bioinformatical Method [J]." Chinese Journal of Biologicals 4 (2005): 010.

[12] Fan, Wenfei, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. "Adding regular expressions to graph reachability and pattern queries." In Data Engineering (ICDE), 2011 IEEE 27th International Conference on, pp. 39-50. IEEE, 2011.

[13] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. Graphite: A visual query system for large graphs. In Data Mining Workshops, 2008. ICDMW08. IEEE International Conference on, pages 963966. IEEE, 2008.

[14] Fan, Zhe, Yun Peng, Byron Choi, Jianliang Xu, and S. Bhowmick. "Towards efficient authenticated subgraph query service in outsourced graph databases." (2013): 1-1.

[15] Csardi, Gabor, and Tamas Nepusz. "The igraph software package for complex network research." InterJournal, Complex Systems 1695.5 (2006).

[16] McKay, Brendan D. "Nauty users guide (version 2.4)." Computer Science Dept., Australian National University (2007).

[17] Foggia, Pasquale. "The vflib graph matching library, version 2.0." (2001).

[18] Jouili, Salim, and Valentin Vansteenberghe. "An empirical comparison of graph databases." In Social Computing (SocialCom), 2013 International Conference on, pp. 708-715. IEEE, 2013. Henzinger, Monika Rauch, Thomas A. Henzinger, and Peter W. Kopke.

[19] "Computing simulations on finite and infinite graphs." In Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on, pp. 453-462. IEEE, 1995.

[20] Nisar, M. Usman, Arash Fard, and John A. Miller. "Techniques for graph analytics on big data." In Big Data (BigData Congress), 2013 IEEE International Congress on, pp. 255-262. IEEE, 2013.

[21] McKay, Brendan D. Practical graph isomorphism. Department of Computer Science, Vanderbilt University, 1981.