Query Processing in Graph Databases

by

Supriya Ramireddy

(Under the Direction of John A.Miller)

Abstract

Graph data are extensively associated with state-of-the-art applications in a variety of domains which include Linked Data and Social Media. This drives the need to have graph databases that can effectively store and manage graph data. Relational query processing has become efficient due to many decades of research in the field of data management and processing, among which translating SQL into relational algebra operations plays a key role in query processing. Based on relational algebra, many graph algebras have been defined that can be used for query processing and optimization in graph databases. We propose a graph algebra which operates on graph databases, for processing queries. We have implemented a graph algebra as a part of ScalaTion and compared it with Neo4j and MySQL with respect to query processing times. Various queries are tested on datasets with a few vertices to a large number of vertices. Graph databases perform well when the database gets larger compared to relational databases. Increase in the number of joins in queries, decreases the performance of relational databases, whereas equivalent queries in graph databases comparatively exhibit good performance. Among graph databases compared in the study, ScalaTion shows better performance.

Index words: Graph Databases, Graph Query Language, Graph Algebra, Query Processing, Pattern matching, Query optimization;

Query Processing in Graph Databases

by

Supriya Ramireddy

B.Tech., Jawaharlal Nehru Technological University, 2013

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2017

Query Processing in Graph Databases

by

Supriya Ramireddy

Acknowledgments

A special thanks to my major professor Dr Miller, he has been tremendous mentor through out my research and I have had a great learning experience working with him. I would like to thank Dr Lakshmish Ramaswamy for being a part of my committee and I learnt a lot from his courses Distributed Systems and Advanced Data Intensive Computing. I would also like to thank Dr Ismailcem Budak Arpinar for accepting to be a part of my committee and timely guidance. I also thank Vinay Kumar Bingi for his help in my thesis.

I would like to dedicate my thesis to my late mother. I am grateful to my mother and father for all the support they have given me even through the times of hardship. I am indebted to my husband for his immense support and encouragement. I would like to thank friends and family for all their support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many state-of-the-art applications are generating huge amounts of data in the form of social networks, Web graphs, citation networks, etc. If the data from such applications has to be represented in the form of Relational tables, several many-to-many relationships between the entities will be created. But the same data can be represented more explicitly in the form of graphs as an edge can be drawn for each relationship. Therefore, there comes the need for the storage of graphs and retrieval techniques, which can be addressed by designing a relevant Graph Database.

As using the relational model for storing and querying graph structured data is inefficient [1], there is a need for a native graph data model (in turn, graph databases) to store and query the graph-structured data. Among the existing graph data models, Resource Description Framework (RDF) [2] and Property Graph [3] are very prominent for data storage in graph databases. RDF stores the data in the form of triples (Subject, Predicate, Object) and is standardized by the W3C. Property graphs model the data through the creation of vertices corresponding to entities and the creation of edges to represent the relationship between entities.

The real-world graph data may be dynamic and heterogeneous [4]. Different types of vertices and edges with arbitrary properties co-exist in the same graph. The number of properties and the types of properties corresponding to similar entities might be different and properties can be added dynamically. Rigid schemas can be less suitable for representing graph data. Comparing with the Entity-Relational Model (ER Model), a vertex corresponds to an entity and property of a vertex to an attribute and a relationship between entities to an edge between vertices [5]. A case study of the mapping from a graph model to the ER Model is presented in [6].

A survey of the graph databases that existed early (e.g., LDM [7], Hypernode [8] and GROOVY [9]) is presented in [10] in terms of the data models, integrity constraints, and Query and Manipulation languages. Most of the work reviewed in [10] is of theoretical interest more than practical developments. A systematic analysis and comparison of the current graph databases is presented in [11], they evaluate graph databases in terms of their support for querying a set of essential graph queries. AllegroGraph[1] is one of the earliest in the current generation graph databases, but its current development is oriented towards the Semantic Web (i.e., RDF, SPARQL and OWL). DEX[2], which is implemented in Java based on bitmaps, is oriented to provide good performance on very large graphs. The HyperGraphDB[3] database implements the hypergraph data model where the notion of an edge is extended to more than two vertices. It supports the natural representation of higher-order relations for the data in areas such as artificial intelligence and bio-informatics. InfiniteGraph[4] is a database aimed to support large graphs in distributed environments. It provides efficient data traversals on the massive and distributed data stores. Neo4j[5] is one of the popular graph databases in recent times which is implemented in Java and is based on a network oriented model. It implements an object oriented API, a native disk-based storage manager and a framework for graph traversals. Neo4j offers persistence, high performance and scalability. OrientDB[6] is multi-model database which is capable of working with Graph, Document, Key-Value, Geo-spatial and reactive models. In addition, Titan[7], Apache TinkerPop[8] are other popular graph databases. These databases are well studied and comapred in [12, 10, 13, 11].

[1]`http://www.franz.com/agraph/allegrograph/`
[2]`http://sparsity-technologies.com/`
[3]`http://hypergraphdb.org/`
[4]`http://www.objectivity.com/products/infinitegraph/`
[5]`https://neo4j.com`
[6]`http://www.orientechnologies.com/orientdb/`
[7]`http://thinkaurelius.github.com/titan`
[8]`http://tinkerpop.apache.org/`

Graph databases are mainly needed for the management of highly interconnected data. They provide solutions for recent applications in domains, such as social network analysis, telecommunications, web mining, semantic web, chemistry, biological networks, marketing, spatial analysis and criminal networks[14, 15].

Query processing mainly consists of four steps: Parsing, translation, optimization and evaluation. A parser generator paired with a grammar generates a parser for the query language. A parser produces an abstract syntax tree for a given query. The abstract syntax tree is translated to a graph algebra expression tree, which may be optimized and evaluated by the query processor. Some existing graph databases have their algebra defined with correspondence to relational algebra [16, 4]. [17] extends relational algebra with two new graph specific operators. [18] consolidates the existing graph algebra operators in the literature and proposes two additional operators. We have implemented graph algebra operators which operates on graph databases, for processing queries over graph data.

Even though, various query languages have been proposed for query processing on graph databases, a standard query language equivalent to SQL is yet to emerge. Cypher[9] is a SQL-like pattern matching declarative query language used by the Neo4j graph database which models data in the form of property graphs. PGQL [19] is developed at Oracle labs very recently which is almost as flexible as SQL. The SPARQL[10] query language is used to retrieve and manipulate data stored in the form of Resource Description Framework (RDF) triples. Apache Tinkerpop's graph traversal language, Gremlin[11], is a functional language supporting both imperative and declarative constructs. Expressing and debugging queries on graphs is easier in these specialized graph query languages. On the other hand, even though many graph queries can be expressed in the form of SQL, the relational optimizer might fail in producing a good execution plan, since such SQL queries may be large and contain expensive self joins. The special query optimizer designed for graph databases may

[9]`https://neo4j.com/developer/cypher-query-language/`

[10]`http://www.w3.org/2009/sparql/`

[11]`https://tinkerpop.apache.org/gremlin.html`

take into account graph access patterns and types of queries and develop special techniques for solving complex queries.

We have implemented graph algebra operators as a part of ScalaTion[12] which can be made use for query processing. We are able to process many types of queries from the Neo4j developer manual[13] through the consecutive execution of the algebra operators implemented. The performance of ScalaTion is compared with the graph database Neo4j and the relational database MySQL in terms of query processing. We have observed that the graph databases perform well compared with relational databases when the size of dataset gets larger and for complex queries.

The rest of this document is organized as follows: various data models for representing graph data are presented in Chapter 2. A brief overview of the existing graph algebras and use of graph algebra in query processing is provided in Chapter 3. In Chapter 4, we talk about the algebra operators of different graph algebras in detail; also our implementation of graph algebra is discussed. How the query processing is done is discussed in Chapter 5. Different techniques for query processing in graph databases are presented in Chapter 6. The performance of two graph databases and a relational database in terms of query processing are compared in Chapter 7. Chapter 8 provides conclusions and future work.

[12] `http://cobweb.cs.uga.edu/~jam/scalation.html`,`https://github.com/scalation/scalation`

[13] `https://neo4j.com/docs/developer-manual/current/cypher/clauses/match/`

Chapter 2

Data Models for Graph Databases

In graph databases, the data models are essentially centered around graphs, vertices and edges. The types of graphs can be broadly categorized at a high level as: Directed graphs and Undirected graphs. Directed graphs consists of a set of vertices and directed edges (the source and destination vertices are distinguished) between a pair of vertices, Undirected graphs consists of undirected edges (source and destination vertices are not distinguished). Undirected graphs are useful in various applications such as: communication networks, protein-protein interaction networks. Almost all the graph databases that were discussed in Chapter 1 use directed graphs for data modeling. The directed graphs are considered to be more general as an undirected graph can be easily simulated with a directed graph by replacing each undirected edge with two edges in opposite directions. The Directed graphs can be modeled in different ways by adding labels to vertices and edges.

2.1 Data Models

2.1.1 Directed Graph

A (simple) directed graph can be defined as a 2-tuple $G = (V, E)$, where V is a non-empty set of vertices and E is a set of directed edges [20]. A directed edge is an edge where the end-points are distinguished, one vertex is the source and the other vertex is the destination. It cannot have either self-loops or multiple edges. A directed graph $G = (V, E)$ can be mathematically defined as follows:

$$
\begin{aligned}
&V = \text{set of vertices}\\
&E = \text{set of directed edges}\\
&E \subseteq \{(u,v) | u \in U, v \in V, u \neq v\}
\end{aligned}
$$

Each edge $e \in E$ is specified by an ordered pair of vertices $e = (u, v)$ where $u \in V$ and $v \in V$. Directed graphs are used in applications where the relationship is represented by an edge is 1-way or asymmetric. Examples include: 1-way streets, one person likes another person but not vice versa in social networking sites, one entity is bigger than another entity, one job has to be completed before another.

2.1.2 Labeled Directed Graph

Labels are added to the vertices of directed graphs to provide more information. Labels can be added to the vertices of directed graphs through a function λ_v mapping from the vertices to labels. A labeled directed graph $G = (V, E, L_v, \lambda_v)$ can be mathematically represented as follows: the first two elements defines the vertices and the edges and are same defined in the previous definition, the next two elements for vertex labels are defined as follows:

$$
\begin{aligned}
L_v &= \text{ set of vertex labels}\\
\lambda_v &: V \rightarrow L_v \text{ (vertex labeling function)}
\end{aligned}
$$

2.1.3 Labeled Directed Multi-Graph

A directed graph with self-loops and multiple edges (parallel edges in the same direction) is called a multi-digraph or directed multi-graph. We make use of the directed multi graph in the implementation of the graph query processing through out this thesis. A labeled directed multi-graph $G = (V, E, \alpha; L_v, L_e, \lambda_v, \lambda_e)$ can be mathematically defined as follows: The topology of the graph is defined by the first three elements in the above definition, the vertices are defined same as in the previous definitions. The adjacency function α indicates the source and destination vertices for any edge e, that is, $\alpha(e) = (u, v),$ where $u \in V$ and $v \in V$.

$$E = \text{set of directed edges}$$

$$\alpha : E \to V \times V$$

Labels are added to edges to be able to differentiate between multiple edges from the same source vertex to the same destination vertex. The labels are defined through a function mapping from the edges to edge labels as follows:

$$L_e = \text{set of edge labels}$$

$$\lambda_e : E \to L_e \text{ (edge labeling function)}$$

2.1.4 Property Graph

A property graph is one of the popular graph data models supported by the various existing graph databases. It is an effective data model for graphs as the above types of graphs are subsets of property graphs and provides a rich set of features for the user to model domain-specific real world data. A property graph is a labeled, attributed, directed multi-graph. Both vertices and edges are associated with attributes/ properties. Neo4j's Cypher language is one of the declarative query languages which can be used for querying property graphs. Some of the systems that use property graphs for storing the graph data, do not have a declarative query language, they make use of imperative APIs for data access. A property graph $G = (V, E, \alpha, L_v, L_e, \lambda_v, \lambda_e, P, Q)$ can be mathematically defined as follows: The topology of the graph is defined by the first three elements in the above definition and are defined in previous definitions. The next four elements defines the labels for vertices and edges and are defined in the previous definitions. The last two elements (P, Q) define the properties for vertices and edges. As specified in [17], P is a set of vertex properties and Q is a set of edge properties. Let D be a set of domains. A property $p_i \in P$ is a function $p_i : V \rightarrow D_i \cup \epsilon$ (ϵ is the empty set) which assigns a property value from a domain $D_i \in D$ to a vertex $v \in V$. If p_i is not defined for v, then $p_i(v)$ returns ϵ. A property $q_j \in Q$ is a function $q_j : E \rightarrow D_j \cup \epsilon$ which assigns a property value from a domain $D_j \in D$ to an edge $e \in E$. If q_j is not defined for e, then $q_j(e)$ returns ϵ.

$$P = \text{ set of vertex properties}$$
$$Q = \text{ set of edge properties}$$

2.1.5 Resource Description Framework (RDF)

The RDF data model is designed for the Semantic Web[1]. RDF databases are collections of (*Subject, Predicate, Object*) triples, where each triple has a binary relation *Predicate* between *Subject* and *Object*. Because of the homogeneous nature of the RDF databases, they can be referred to as labeled directed graphs, where each triple has a directed edge from *Subject* to *Object* under label *Predicate* [21]. For example, information about a person could include the following triples:

(id1, hasName, "John"),

(id1, bornOn, "July 3,1992"),

(id1, bornIn, "id2"),

(id2, hasName, "Atlanta"),

2.2 Data structures

2.2.1 Directed Graph

A Directed graph can be either represented as a collection of adjacency lists or as an adjacency matrix [22]. The *adjacency-list* representation of a graph $G = (V, E)$ consists of an array `ch`

[1]`https://www.w3.org/RDF/`

of size $|V|$ collections, one for each vertex in V. For each vertex $u \in V$, `ch(u)` consists of all the vertices that can be reachable by one outgoing edge from u.

For storing the adjacency lists corresponding to each vertex, there are many options such as ArrayList, LinkedList, and Set. We prefer sets to lists because of many reasons, such as for performing the algebra operations, the `contains` operation in sets is more efficient than in the lists. `ch(u)` can be defined as follows:

$$\texttt{ch}(\texttt{u}) = \{v|(u,v) \in E\}$$

Therefore, a directed graph with adjacency list representation can be stored by *ch* as follows:

$$\texttt{ch} : \texttt{Array}[\texttt{Set}[\texttt{Int}]]$$

In the adjacency matrix representation of the graph [22], a matrix of size $|V| \times |V|$ is filled with bits (or integers, doubles), value corresponding to indices (u, v) is 1 if there is an edge from u to v, otherwise 0. For the graphs with fewer number of edges, adjacency list representation can be used. If the graph has many edges i.e., $|E|$ is close to $|V|^2$, then an adjacency matrix representation may be more relevant. But the operation of finding if there is an edge between a pair of vertices (u, v) becomes very easy in an adjacency matrix representation, whereas it is not easy in an adjacency list representation unless a set is used. The values in the adjacency matrix are filled as:

$$A(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

2.2.2 Labeled Directed Graph

The class definition of labeled directed graph in ScalaTion stores the graph using an adjacent set representation, for the same reason discussed above the vertices are implicitly 0, ..., n-1.

```
 class MDiGraph  (ch: Array[SET[Int]],
                  label: Array[TLabel],
                  inverse: Boolean,
                  name: String)
```

In the above class definition, `ch` stores the child vertices as an adjacency set, *label* maps the vertices to vertex labels where `TLabel` is the type parameter. All the edges that are reachable by outgoing edges from u are called children of u, which can be obtained from `ch(u)` in the above representation. The `inverse` is used to specify whether to store parents for all the vertices in the graph. The parents of a vertex are the vertices which are connected by the incoming edges, can be defined as follows:

$$pa(w) \ = \ \{v|(v,w) \ \in E\}$$

`name` is used to specify the name of the graph.

2.2.3 Directed Multi-Graph

The class definition of directed multi-graph in ScalaTion is an extension of the labeled directed graph, it adds edge labels (`elabel`) for all the edges corresponding to the pairs of vertices. elabel permits to have more than one edge for the same pair of vertices as they can be differentiated by the edge labels. In the following class definition for a directed multi-graph, `Pair` is the alias for `Tuple2[Int, Int]`.

```
class MuDiGraph  (ch: Array[SET[Int]],
                  id: Array[Int],
                  label: Array[TLabel],
                  elabel: Map [Pair, SET[TLabel]],
                  inverse: Boolean,
                  name: String)
```

Both parent and children vertices for any given vertex are derivable from the above definition.

Chapter 3

Related Work

Query translation and optimization are important factors involved in the query processing. Various optimization techniques have been proposed recently for tackling the complexity of graph query languages and efficient evaluation of the queries over the graph databases [23, 24]. The translation of queries into algebraic expressions for the processing and analysis is widely used in the databases [25].

The idea of using algebra operators for query processing dates back to 1970's. An algebra for relational databases was defined initially by Codd [26]. In this regard, attempts were made to make use of relational algebra for query translation in graph query languages such as SPARQL [27] in order to make use of existing work in relational databases on query planning and optimization. The RDF triples were represented in the form of table with three columns namely *subject, predicate, object.* Each triple in the RDF format is translated to a row in the table. Even though it was possible to translate most SPARQL queries into relational form, there were some issues to be addressed such as semantic mismatches and corner cases, such as 1) Unbound variables, the NULL value and headings, 2) Join behaviour with missing information, 3) The Nested OPTIONALs Problem, 4) The FILTER scope problem as listed in [27].

[23] describes the algebraic operators used in SPARQL and makes a point that the SPARQL Algebra (SA) is very similar to that of Relational Algebra (RA). The study in [28] reveals that SA and RA have the same expressive power. A set of rewriting rules for SPARQL Algebra operators equivalences for query optimization is also provided in [23].

Graph algebra is defined along the lines of relational algebra in [16], which is later extended by [4] and they claim it be atleast as expressive as relational algebra. [17] states that query optimization based on the graph algebra greatly improves on query processing times. Many graph algebras are often developed in the context of relational algebra where relational database is used as backend to process data. Though, it is a good idea to extend the relational algebra and define graph algebra, in addition graph specific operators should be defined at a high level irrespective of the database backend [17].

The existing graph algebra operators of different graph query languages are consolidated and presented as an integrated graph algebra in [18]. We have made an attempt to define the graph algebra based on a graph data model, labeled directed multi-graph. The algebra operators defined are close to the high level algebra operations for Neo4j. The algebra operators used for query processing produce results equivalent to that of Neo4j results.

Chapter 4

Graph Algebra

4.1 Review of Graph Algebras

An algebra is an intermediate language that an expression in a high-level language is translated to. There are existing proposals to define graph algebra in order to build query processing similar to that of relational algebra. Although, the relational algebra operators cannot be directly used on the graph databases, most popular graph databases extend the relational algebra with a few other graph specific operators [27, 16, 23, 4, 17]. The relational algebra can be extended for graph databases because some of them use relational database system as the back end for processing graph data. We attempt to provide a brief overview of the graph algebras of existing graph databases and then try to explain how we defined our own graph algebra, which can be used for query processing.

The idea of extending relational algebra to graph databases is being followed in many graph databases. [17] argues that extension has to be done at a higher level to support native graph databases such as Neo4j. In addition to the relational algebra based operators, two more high level operators are defined in [17]. In order to combine the new operators with the existing operators of relational algebra, they represent subgraphs as relations. The two new operators are GetNodes and Expand. The GetNodes returns a graph relation containing all the nodes of the underlying graph G. The Expand operator adds two new columns to the graph relation, where one column contains the neighbors of a particular node and the second column contains the corresponding edges.

Operators in SPARQL can be matched to corresponding operators in relational algebra [23]. The Algebraic operations in SPARQL are: SELECT (Project), AND (Join), FILTER

(Select), OPTIONAL (Left outer join), UNION (Algebraic union), MINUS (Algebraic minus). SPARQL optimization mainly focuses on optimizing the queries involving AND operator compared to other operators.

GRAD Algebra is another example of graph algebra which is extended from the GraphQL algebra, which is defined along the lines of relational algebra [4]. The selection operation takes a graph pattern as input and returns all the matching subgraphs of the data graph. The cartesian product is performed on two collections of graphs. Let S_1, S_2 be two collections of graphs, the output of cartesian product consists of set of pairs of unconnected graphs. The set operators union and minus are also included in the GRAD algebra. The union operation does not change the internal structure of the graphs. The two input graphs are put together to form a new graph without concatenation. The union is different from that of relational algebra union in that there is no need for a common structure. The composition operator is used to genrate the new graph using the information extracted from the data graph, by the set operators (union and minus). The minus operation $G_1 - G_2$ deletes all the isomorphic elements of G_1 and G_2 and returns the remaining elements of G_1. The union,minus and cartesian product operators defined above enable putting together a collection of graphs. If graph entities have to be merged together, then these cannot be used. Therefore, the join operation is introduced. The input to the join operation is a collection of graphs and the join predicate. The join is performed in terms of cartesian product and select. After the cartesian product produces set of pairs of graphs, a selection is performed to retain only the pairs of subgraphs satisfying the predicate and to merge the elements of subgraphs that satisfy the predicate. Two other operations, project and rename, can also be defined in terms of composition operation. The five operators Select, Cartesian Product, Composition, Union and Minus are complete in terms of expressive power. Other algebraic operators can be defined in terms of these five operations.

Table 4.1: Algebra Operators

Algebra	Operators
Cypher Algebra	Select, Project, Union, minus, Join, GetNodes, Expand
SPARQL Algebra	Select, Project, Union, minus, Join, left outer join
GRAD Algebra	Select, Cartesian Product, Primitive Composition, Union, Difference
Gremlin Algebra	Select, Project, Concatenate, Union, Join

4.2 Algebra Operations

The Graph Algebra proposed by the various existing graph databases use relational database as a basis. Our graph algebra at a high level consists of Select, Expand, Union, Minus and Intersect, for defining which we make use of the following notations:

$\iota : V \to \mathbb{N}$ (vertex id function)

$\lambda_v : V \to L_v$ (vertex labeling function)

$\lambda_e : E \to L_e$ (edge labeling function)

getVertices:

The input to this operator is a graph and it returns all the vertices of the given graph.

getLabels:

The input to this operator is a graph and a set of vertices, it returns the vertex labels corresponding to the given vertices.

Select:

selectByVertex:

The input to this operator is a set of vertices and the label of the vertex that needs to be filtered. All the vertices that match the given vertex label are selected and the output is a set of filtered vertices.

selectByVertex has a variation in which the input is a graph and the label of a vertex to be selected. All the vertices corresponding to a given vertex label are selected. All the edges between the selected vertices are inlcuded in the graph. The output is a sub-graph of the input graph with the filtered vertices and the respective edges between them.

selectByEdge:

The input is a set of vertices and the label of the edge that has to be filtered. All the vertices that has the edges between them with the label same as the input will be selected in addition to the edges, and the map of vertices mapping to the edge label are returned as output.

selectByEdge has a variation, in which ths input is a graph and the label of edges that has to be filtered. Output is the subgraph consisting of all the vertices of the input graph, but only the edges having the given input edge label.

$\sigma_p(V) = \{v | v \in V \wedge p(v)\}$

$\sigma_p(E) = \{e | e \in E \wedge p(e)\}$

where p is a predicate on the vertex or edge to be selected.

Expand Forward:

The Expand Forward operation $\varepsilon^l_{\rightarrow}$ gives all the end vertices v reachable via an outgoing edge label l from any given set of start vertices V'.

$\varepsilon^l_{\rightarrow}(V') = \{v | u \in V' \wedge (u, v) \in \alpha(E) \wedge \lambda_e(u, v) = l\}$

Expand Backward:

Expand Backward operation gives all the start vertices for a given set of end vertices V' and a given edge label l. This operator expands the incoming edges. In other words, it is the opposite of the Expand Forward operation.

$\varepsilon^l_{\leftarrow}(V') = \{u | v \in V' \wedge (u, v) \in \alpha(E) \wedge \lambda_e(u, v) = l\}$

Union:

The input for the union operation consists of two sub-graphs. The output sub-graph consists of all the vertices of both the graphs with no duplicates, along with all the edges between those vertices without duplicates.

$G(V, E, \alpha; L_v, L_e, \lambda_v, \lambda_e) = G^1(V^1, E^1, \alpha^1; L^1_v, L^1_e, \lambda^1_v, \lambda^1_e) \cup G^2(V^2, E^2, \alpha^2; L^2_v, L^2_e, \lambda^2_v, \lambda^2_e)$ can be defined as follows:

$V = V^1 \cup \{v | v \in V^2 \wedge (v \notin V^1 \vee \lambda_v^1(v) \neq \lambda_v^2(v))\}$

$E = E^1 \cup \{e | e \in E^2 \wedge (\alpha^2(e) \notin E^1 \vee \lambda_e^1(e) \neq \lambda_e^2(e))\}$

$\alpha : E \to V \times V$

$L_v = L_v^1 \cup L_v^2$

$L_e = L_e^1 \cup L_e^2$

$\lambda_v : V \to L_v$ s.t. $\lambda_v(v) =$ if $v \in V^1, \lambda_v^1(v)$ else $\lambda_v^2(v)$

$\lambda_e : E \to L_e$ s.t. $\lambda_e(e) =$ if $e \in E^1, \lambda_e^1(e)$ else $\lambda_e^2(e)$

Minus:

The Minus operation takes two sub-graphs as input and returns a sub-graph as output. The output sub-graph consists of the vertices of G^1 which are not present in G^2 as well as the edges between the selected vertices of G^1.

$G(V, E, \alpha; L_v, L_e, \lambda_v, \lambda_e) = G^1(V^1, E^1, \alpha^1; L_v^1, L_e^1, \lambda_v^1, \lambda_e^1) - G^2(V^2, E^2, \alpha^2; L_v^2, L_e^2, \lambda_v^2, \lambda_e^2)$ can be defined as follows:

$V = \{v | v \in V^1 \wedge (v \notin V^2 \ \vee \lambda_v^1(v) \neq \lambda_v^2(v))\}$

$E = \{e | e \in E^1 \wedge (e \notin E^2 \vee \lambda_e^1(e) \neq \lambda_e^2(e))\}$

$\alpha : E \to V \times V$

$L_v = L_v^1 - L_v^2$

$L_e = L_e^1 - L_e^2$

$\lambda_v : V \to L_v$ s.t. $\lambda_v(v) = \lambda_v^1(v)$

$\lambda_e : E \to L_e$ s.t. $\lambda_e(e) = \lambda_e^1(e)$

Intersect:

The Intersect operation takes two sub-graphs as input and returns a sub-graph as output. The output sub-graph consists of the common vertices which are present in both G^1 and G^2 as well as the edges between the selected vertices and are present in both the graphs.

$G(V, E, \alpha; L_v, L_e, \lambda_v, \lambda_e) = G^1(V^1, E^1, \alpha^1; L_v^1, L_e^1, \lambda_v^1, \lambda_e^1) \cap G^2(V^2, E^2, \alpha^2; L_v^2, L_e^2, \lambda_v^2, \lambda_e^2)$ can be defined as follows:

$V = \{v | v \in V^1 \wedge v \in V^2 \wedge \lambda_v^1(v) = \lambda_v^2(v)\}$

$E = \{e | e \in E^1 \wedge e \in E^2 \wedge \lambda_e^1(e) = \lambda_e^2(e))\}$

$\alpha : E \to V \times V$

$L_v = L_v^1 \cap L_v^2$

$L_e = L_e^1 \cap L_e^2$

$\lambda_v : V \to L_v$ s.t. $\lambda_v(v) = \lambda_v^1(v) \cap \lambda_v^2(v)$

$\lambda_e : E \to L_e$ s.t. $\lambda_e(e) = \lambda_e^1(e) \cap \lambda_e^2(e)$

4.3 Comparison of Graph Databases with Relational Databases

Graph data can be stored in the form of relational tables. Two tables would have to be created for storing the graph data [4]. The data corresponding to a vertex can be represented in a table with all the information (vertex id, vertex label, vertex properties) corresponding to a vertex as attributes in the table as shown below:
VTable($vid, vlabel, p_1, p_2, ..., p_m$).

The data corresponding to an edge can be represented in another table with all the information corresponding to an edge (source vertex id, destination vertex id, edge label, edge properties) as attributes in the table as shown below:
ETable($vid1, vid2, elabel, q_1, q_2, ..., q_n$).

Chapter 5

Query Processing using Graph Algebra

We make use of the graph algebra defined in Chapter 4 for query processing. Given a query, it should be parsed using a parser to generate an abstract syntax tree which is translated to algebraic operators. The query optimizer can reorder the operations in order to reduce the size of intermediate results. Once the query optimizer produces the query plan, it has to be evaluated. We refer to the query plan generated by Neo4j for a cypher query, and make use of the same plan inorder evaluate the query using our operators. There are many parallels between cypher algebra and and our algebra. We use expansion over edges very often for most types of the queries as does Neo4j [29].

The following are some of the queries that are illustrated in the Neo4j Developer manual[1] section 3.3.1 which can be evaluated using the algebraic operators that we have defined in chapter 4. As our graph store does not support any declarative query language yet, the query is returned as a series graph algebra operations.

In the paragraphs below, each of nine graph queries (shown in Table 5.1) that are used for testing on various database systems are explained and the corresponding SQL queries are given in Appendix A. The schema of the database used consists of vertices of types (Movie, Person).

Query 1 returns all the vertices in the graph. The getVertices() operation returns all the vertices.

[1]`https://neo4j.com/docs/developer-manual/current/cypher/clauses/match/`

Table 5.1: Queries

	Neo4j Query	ScalaTion
1.	MATCH (n) RETURN n.name	g1.getLabels (g1.getVertices)
2.	MATCH (movie:Movie) RETURN movie.name	g1.getLabels (g1.selectBySchema (θ, "Movie") (g1.vertexSet))
3.	MATCH (u {name: 'Oliver Stone'})−− (v) RETURN v.name	g1.union (g1.getLabels (g1.expandAll (g1.selectByVLabel (θ, "Oliver Stone") (g1.vertexSet))), g1.getLabels (g1.expandBackAll (g1.selectByVLabel (θ, "Oliver Stone") (g1.vertexSet))))
4.	MATCH (u {name: 'Oliver Stone'}) −>(v: Person) RETURN v.name	g1.getLabels (g1.expandBySchema (g1.selectByVLabel (θ, "Oliver Stone") (g1.vertexSet), "Person"))
5.	MATCH (u {name: 'Oliver Stone'}) −> (v) RETURN v.name	g1.getLabels (g1.expandAll (g1.selectByVLabel(θ, "Oliver Stone") (g1.vertexSet)))
6.	MATCH (u { name: 'Oliver Stone' }) −[r]−>(v) RETURN r.role	g1.expandEdges (g1.selectByVLabel (θ, "Oliver Stone") (g1.vertexSet))
7.	MATCH (u { name: 'Wall Street' }) <− [r]− (v) where r.role = 'eLabel' RETURN v.name	g1.getLabels (g1.expandBack (g1.selectByVLabel (θ, "Wall Street")(g1.vertexSet), eLabel))
8.	MATCH (u {name: 'Wall Street'}) <− [r] − (v) where r.role = 'eLabel1' or r.role = 'eLabel2' RETURN v.name	g1.union (g1.getLabels (g1.expandBack (g1.selectByVLabel (θ,"Wall Street") (g1.vertexSet), eLabel1)), g1.getLabels (g1.expandBack (g1.selectByVLabel (θ, "Wall Street") (g1.vertexSet), eLabel2)))
9.	MATCH (u { name: 'Charlie Sheen' })−>(v) −>(w) RETURN w.name	g1.getLabels (g1.expandAll (g1.expandAll (g1.selectByVLabel (θ, "Charlie Sheen") (g1.vertexSet))))

Query 2 returns all the vertices of type Movie. All the vertices with a given a type in the graph can be retrieved using selectByVertexType operation. Result of this query are the names of all movies.

Query 3 retrieves all the vertices which are connected to a given vertex, with no relevance about the edge label or the direction of the edge. This can be achieved by making use of both expand and expandBack operations with just the vertex as the input. Union operation can be used to combine the result of expand and expandBack operations.

Query 4 constrains the previous query with a label on vertices, so the vertices that are connected by outgoing edges with Oliver that are type Movie will be returned. selectByVertex operation is first performed to get all the vertices corresponding to Oliver Stone and only the vertices that are labeled Movie that are connected are returned as result by the expandByType operation.

Query 5 returns all the vertices connected with the Person node by an outgoing relationship. The difference from the previous query is there is no restriction on the type of the vertex which is connected to the given vertex. expand operation is used instead of expandByType operation.

Query 6 returns the type of each outgoing relationship from the given vertex which can be done by the expandEdge operation. By reversing the direction of the relationship, expandBack operation Returns the type of each incoming relationship to the given vertex.

For given vertices and given directed edges, expand either in forward or backward direction to get the vertices that are connected through the given relationship, which can be done making using of expand for outgoing edges of a vertex and using expandBack for incoming edges of a vertex.

Query 7 is processed by using an expandBack operation as all the vertices that are connected through an incoming edge named ”acted” to the given vertex.

Query 8 returns all the vertices that are connected to the given vertex with the incoming edge named either ”acted” or ”directed”. expandBack operation can be used twice and the result is combined using union. To match on the vertex and one of the given relationships and return the connected vertices.

Query 9 retrieves the vertices that are connected to the given vertex through forward edges and are two hops away from the given vertex.

The below query forms a triangle matching, whose solution can be obtained by applying expand operations consecutively in ScalaTion as does Neo4j.

$match\ (x3 : person) - [: knows]- > (x1 : person) - [: knows]- > (x2 : person) - [: knows]- > (x3 : person)\ where\ x3.name =\sim\ "u.*"\ return\ x1,\ x2,\ x3$

The order in which the query is executed is shown below:

$match\ (x3 : person)\ where\ x3.name =\sim\ "u.*"\ return\ x3$

$match\ (x2 : person) - [: knows]- > (x3 : person)\ where\ x3.name =\sim\ "u.*"\ return\ x2, x3$

$match\ (x1 : person) - [: knows]- > (x2 : person) - [: knows]- > (x3 : person)\ where\ x3.name =\sim\ "u.*"\ return\ x1, x2, x3$

$match\ (x3 : person) - [: knows]- > (x1 : person) - [: knows]- > (x2 : person) - [: knows]- > (x3 : person)\ where\ x3.name =\sim\ "u.*"\ return\ x1, x2, x3$

The results of the above queries are the intermediate results while trying to solve the triangle query, and the result of the last query above is the result of the triangle query.

The types of queries can be categorized as shown in the Figure 5.1. In the figure, 'Agrees' means that it is implemented in ScalaTion and result is same as that of Neo4j.

Queries	Execution
Get all nodes	Agrees
Get all nodes with a label	Agrees
Related nodes	Agrees
Match with labels	Agrees

(a) Basic node finding

Outgoing relationships	Agrees
Directed relationships and variable	Agrees
Match on relationship type	Agrees
Match on multiple relationship types	Agrees
Match on relationship type and use a variable	Not implemented yet

(b) Relationship basics

Node by id	Agrees
Relationship by id	Not implemented yet
Multiple nodes by id	Not implemented yet

(c) Get node or relationship by id Node by id

Figure 5.1: Types of Queries

Chapter 6

Query Processing using other techniques

In order to query RDF repositories, the SPARQL language has been proposed by W3C. Graph pattern matching is the most used paradigm by the query languages that are used for querying RDF graphs. Query processing techniques vary depending on the underlying storage mechanism. For example if the relational model is used for storing RDF graphs, then the queries are translated from RDF query language such as SPARQL to SQL and then the query is processed on the underlying database. Optimization in RDF involves the join ordering which is determined by drawing alternative query plans. The query plan consists of a tree of operators. The operators can be reordered to compute the operations with less number of intermediate results first. The estimation of cardinality of intermediate results is similar to that of finding the frequency of sub-patterns of a query graph pattern [30].

The open source RDF-3x (RDF Triple eXpress) [24] is a state-of-the-art graph database for management of graph data stored in the format of RDF data model. When the RDF Data grows in size, the storage, indexing and query processing becomes complex. So a comprehensive solution is provided by RDF-3x by storing the RDF triples in a gaint table and indexing the data for storage and query processing.

SPARQL query language is developed to query over the data stored in the form of RDF format. It supports conjunctions (.) and disjunctions (union) of triple patterns, which can be implemented with SPARQL algebra. Each of the conjunctions corresponds to a join operation. A SPARQL query is of the form given below:

Select ?variable1 ?variable2

Where {pattern1 . pattern2 }

Each pattern consists of subject, predicate, object, and each of them is either a variable or a literal. Variables can occur in multiple patterns and imply joins. All possible variable bindings have to be found and returned from select clause as solutions to query processing. RDF triples may be mapped into relational tables in many forms: 1) One giant table with three attributes namely: subject, predicate and object, 2) Triples with the same predicate in one table, 3) Predicates of same type grouped into a cluster.

Clustered-property tables are used in the open-source systems like Jena[1] and Sesame [31]. The giant triples tables are made use in the RDF-3x. But if giant tables are used for storing triples, the queries might involve too many self-joins which is avoided in RDF-3x making use of a set of indexes and very fast processing of merge joins. In order to answer queries involving patterns with more variables than literals which involve several joins, all six combinations of subject, predicate and object in indexes (SPO, SOP, OSP, OPS, PSO, POS) are maintained. Consider the following example
Select ?a ?c where {?a ?b ?c}
It computes all bindings of ?a ?c with any predicate, the actual bindings of ?b are not relevant. These kind of queries are solved making use of aggregated indices where only two out of three columns are stored. Two entries and an aggregated count are stored. This is done for each of the three possible pairs out of a triple and in each collation order (SP, PS, SO, OS, PO, OP), adding another six indexes. In addition to these indexes for pairs in triples, all three one-value indexes containing just (value1, count) entries are also built. These indexes are later used to estimate the cardinalities of the intermediate results during query optimization and to answer the queries. In order to manage space for such an aggressive indexing, a compression mechanism is used for storing the indexes. When the ordered triples are stored in a table, it is very likely that the neighboring triples may have common prefixes, in which the case only the diiference between the triples is stored. In order to manage the space for excessive indexing, triple stores use a dictionary compression, where the IRIs and

[1]`http://jena.sourceforge.net/`

literals are mapped to unique IDs and the triple stores are stores integers rather than strings. RDF-3x uses an incremental approach where each new data item gets an increased internal ID.

Each query in SPARQL can be parsed and expanded into a set of triple patterns. When a query consists of single triple pattern, the indexes can be used and the query can be answered using a single range scan. If there are more than one single pattern, then the results of individual patterns can be joined to answer the query. The key issue in optimizing SPARQL execution plans is join ordering. The literature on join ordering either uses the dynamic programming or greedy algorithms. But join ordering for RDF and SPARQL cannot be answered using those techniques because of their intrinsic characteristics. SPARQL queries will contain multiple star-shaped queries, because of the properties centered around the same entity, for which a strategy for creating bushy join trees is essential rather than left-deep or right-deep trees. Therefore, the index strategy used in RDF-3x encourages merge joins rather than hash or nested-loop joins. The following is an example for SPARQL query that joins two triple patterns:

```
SELECT ?x, ?y WHERE {
?x Acted Movie: 'Wall Street' .
?y Directed Movie: 'Wall Street'}
```

SPARQL comes with a powerful graph matching facility, whose basic constructs are triple patterns. Many queries involve matching the patterns on the RDF triples data. All most all the query languages make use of subgraph matching for solving simple queries. The problem of subgraph pattern matching involves finding all the matching patterns in the data graph. There are many techniques in the literature for sub-graph pattern matching. Among which subgraph isomorphism is popular. The problem of finding all the subgraphs that are isomorphic to query graph is NP-Hard. There has been much research going on to reduce the time complexity to polynomial. Cypher and GraphQL [15] query results satisfy subgraph isomorphism for pattern matching whereas SPARQL query results satisfy graph homomorphism. Variations of graph simulation techniques that are proposed in [32] can be used for pattern matching and are solvable in polynomial time by relaxing some conditions of pattern matching. The simulation techniques proposed include Graph Simulation, Dual Simulation, Strong Simulation, Strict Simulation, Tight Simulation and CAR-Tight Simulation. As finding the patterns in massive graphs can be very time consuming, devising efficient pattern matching algorithms to reduce time complexity is an active field of research.

The datasets such as Yago and DBPedia, automatically retrieve data from *Wikipedia* and store in RDF format. Such data sets are updated very dynamically which needs an effective mechanism for update queries. gStore[33] provides a solution for these update queries, by building an index and also processes the SPARQL queries with wildcards. gStore models RDF graph as labeled, directed multi-edge graph. gStore addresses the issue of SPARQL update queries and makes it possible to process the SPARQL queries with wildcards. Unlike some query processing systems for RDF which use relational databases as backends, they propose to make use of RDF graph constructed of triples and use a novel index mechanism which enables them to provide an effective maintenance algorithm for SPARQL update queries.

Chapter 7

Performance Evaluations

Data Generation

We make use of randomly generated directed multi-graphs of different sizes, and create the same graphs in both Neo4j and ScalaTion to be used for processing the queries. Small-world graphs containing different number of vertices are created. The following table contains the information about the data graphs created.

No. of Vertices	No.of Vertex labels	No. of Edges
10000	5000	9999
20000	10000	19999
30000	15000	29999
40000	20000	39999
50000	25000	69999
60000	30000	59999
70000	35000	69999
80000	40000	79999
90000	45000	89999
100000	50000	99999

(a) Graph Data

The same graphs can be stored in the form of three relational tables for the SQL-based implementation of the queries [16]. We have created three tables: Person (pid, name), Movie (mid, name) and Relations (id1, id2, role), where Person and Movie tables contain the vertices of graph and Relations table stores the edges along with the source and destination vertices. We have created a SQL view VTable (id, name) of all the vertex tables, to be used for queries where the type of vertices is not relevant.

We conducted these experiments on a system with an Intel Core i5 2.70 GHz processor running the 64-bit OS X Yosemite 10.10.5 distribution of the Mac operating system with

8 GB of memory. The versions of Neo4j, MySQL and Scala are 3.1.3, 6.3.4 and 2.12.4, respectively.

Query Evaluation

We have evaluated the query processing times of ScalaTion in comparison with the popular graph database Neo4j. The queries are from the Neo4j developer manual that are also presented in Chapter 4. Each query was run 10 times and the time excluding the first time is averaged to get the best estimate. This was done to exclude the time for JIT compilation which occurs when running for the first time. The data values for queries are randomly selected from the data graph and the same values are used for both ScalaTion and Neo4j queries. We plot the times in both Linear and Log Scale, to see how the time varies with increase in dataset sizes.

We have chosen different types of queries from Neo4j developer manual. These queries consists of simple queries which require a single table for SQL queries and also complex queries which require join of three tables. Based on which we are able to evaluate the performance of relational databases in comparison with graph databases when the query complexity increases and also dataset size increases.

Query 1 retrieves the names of all vertices in the data graph. As we can see in Figure 7.1, the times for ScalaTion and MySQL are almost similar, whereas Neo4j is nearly 20 times slower.

Query 2 retrieves the names of vertices corresponding a particular type. In the graph databases, all the vertices has to be traversed, whereas in SQL, all the vertices of a type are in one table, which makes it obvious that SQL outperforms Neo4j and ScalaTion for this query, which is shown in Figure 7.2.

Query 3 involves two edges in the graph query and the SQL query consists of sub-queries and a union operation. The processing times of ScalaTion are at least 2 times less than Neo4j as shown in Figure 7.3, and the performance of ScalaTion is much better than MySQL

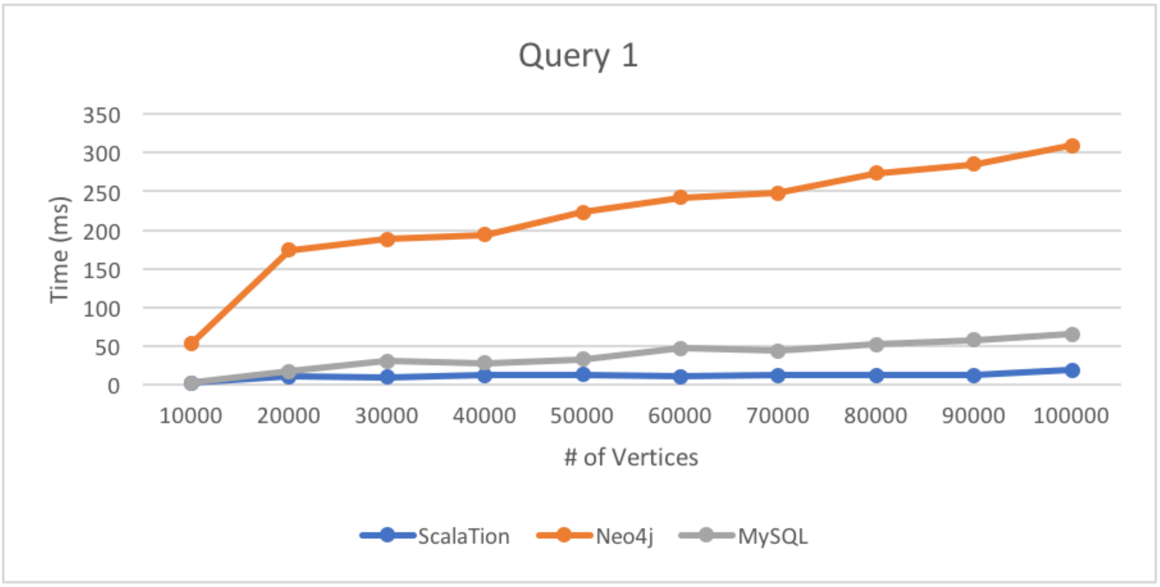

Figure 7.1: Running Time For Query 1

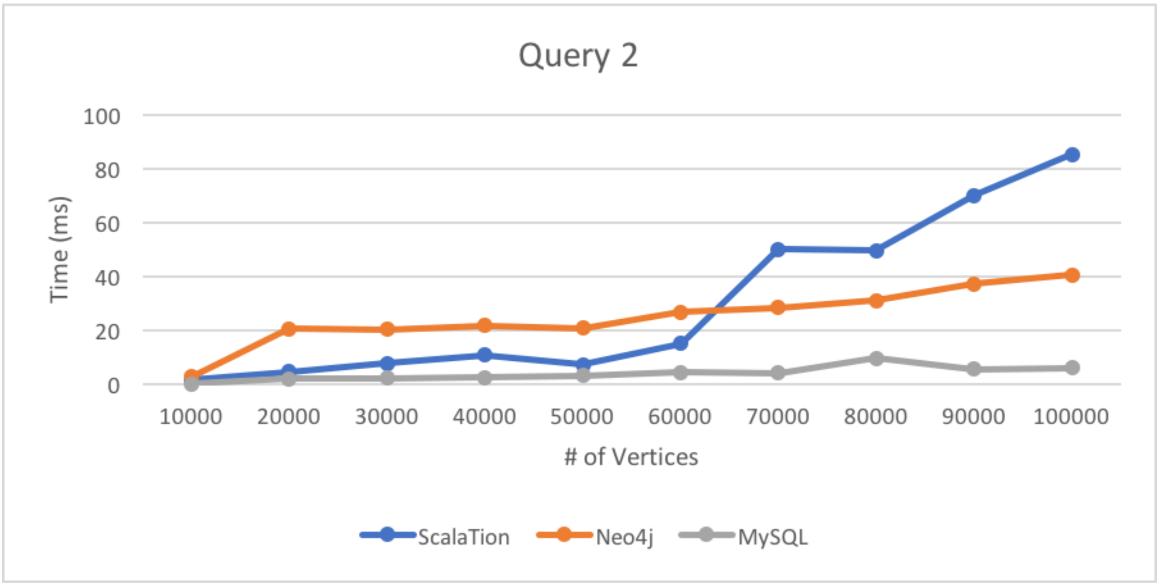

Figure 7.2: Running Time For Query 2

as shown in 7.4, it can be observed from the Figure 7.6 that SQL inloves union and other sub-queries which makes it slower. Whereas it can be seen from Neo4j query plan in Figure

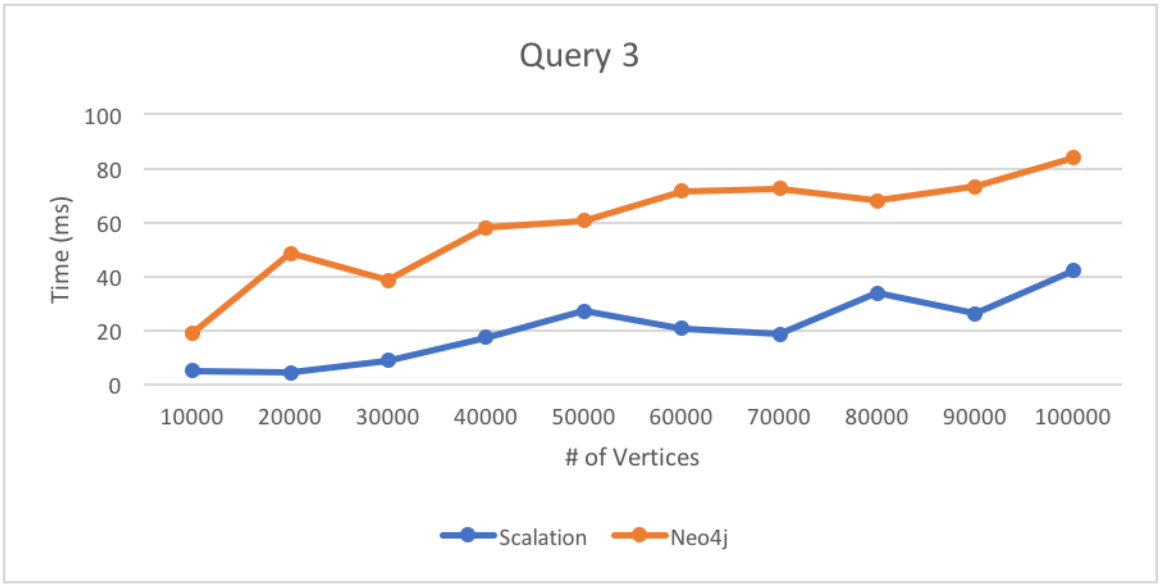

Figure 7.3: Running Time For Query 3

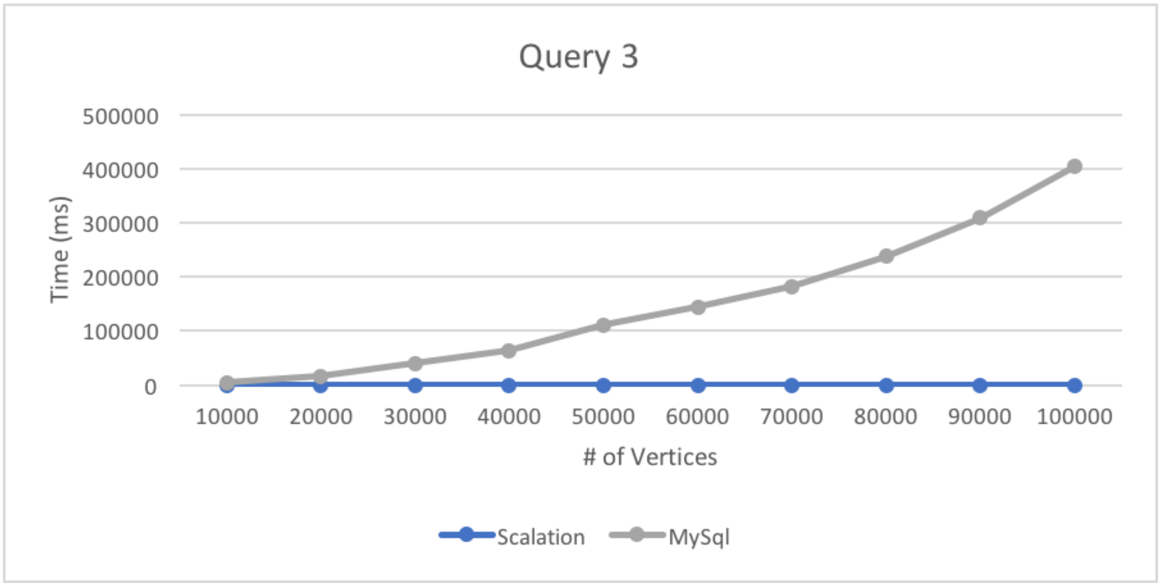

Figure 7.4: Running Time For Query 3

7.5, that all the vertices are scanned in the first step. In the second step, the vertices with given name are filtered, in the third step, the vertices that are connected (either forward or backward edges) to the filtered vertices are expanded. The names of the vertices selected through the expand operation, are selected in the fourth step, which is the desired result.

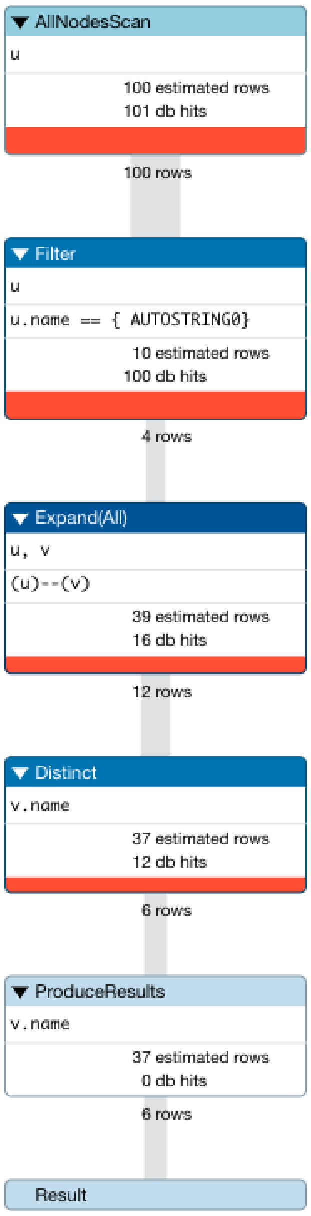

Figure 7.5: Neo4j query plan for Query 3

Query 4 involves one edge in the graph query and the SQL query consists of a sub-query but not union, so the time taken in SQL is atleast three times more than that in ScalaTion. Neo4j is atleast seven times slower than ScalaTion, which is shown in Figure 7.7.

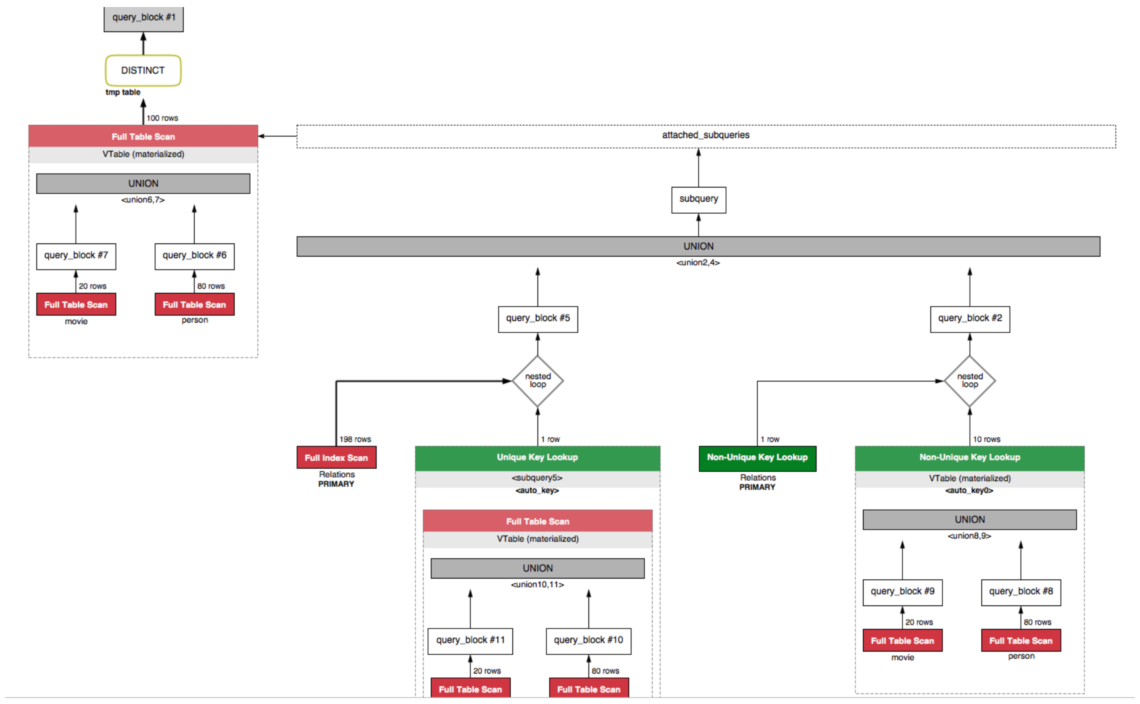

Figure 7.6: MySQL query plan for Query 3

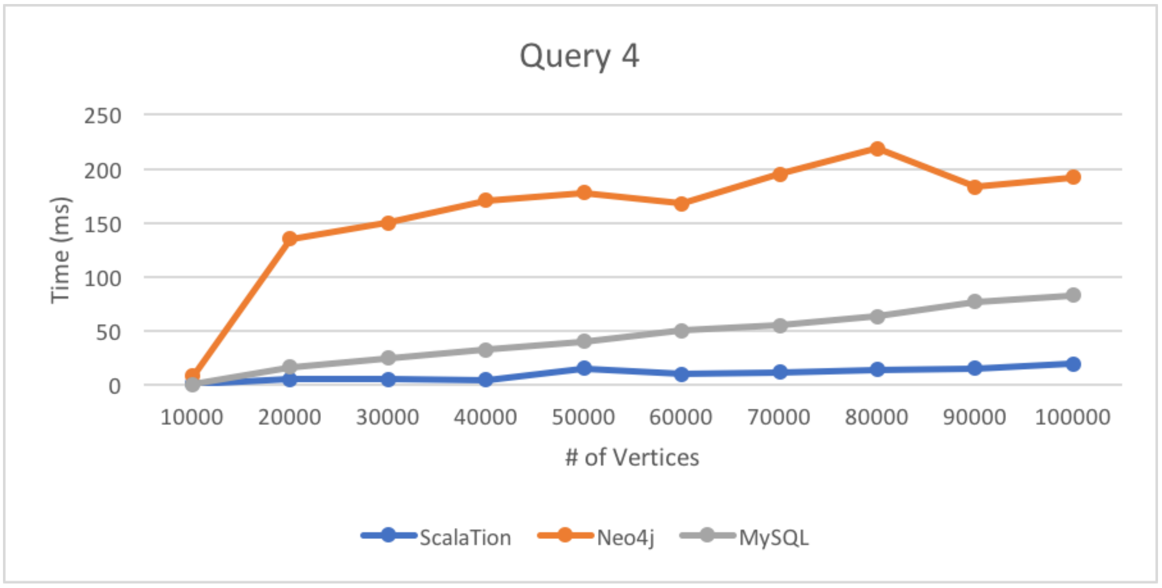

Figure 7.7: Running Time For Query 4

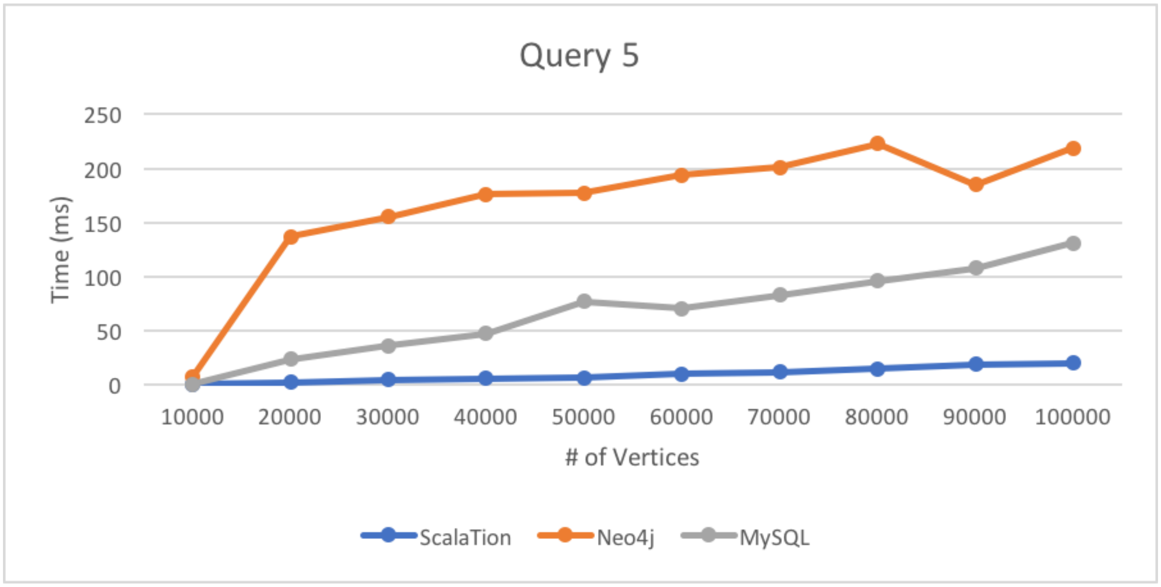

Figure 7.8: Running Time For Query 5

Query 5 is different from query 4 in that, the resulting vertices are selected with no restriction on the type, which is more simple than in query 4. In SQL, the vertex table can be directly used to retrieve the names of resulting vertices. As we can see in Figure 7.8, the time difference in ScalaTion and Neo4j gradually increases and at 100000 vertices, it can clearly seen that Neo4j takes more time than the other two. Query 6 involves one edge in the graph query and one sub-query in MySQL, we can see in Figure 7.9 that ScalaTion is at least fifteen times faster compared to Neo4j and at least two times faster compared to MySql.

Query 7 involves two sub-queries in SQL and an edge in the graph query, the performance of MySql is worse than both ScalaTion and Neo4j as shown in Figure 7.10.

Query 8 is similar to Query 7 except that the edge label can be filtered based on two given values. We can see that the time in MySql gradually increases.

Query 9 involves the join of 3 tables in the SQL query, so we can clearly see that graph queries perform well for these types of queries as shown in Figure 7.12. It can be seen from

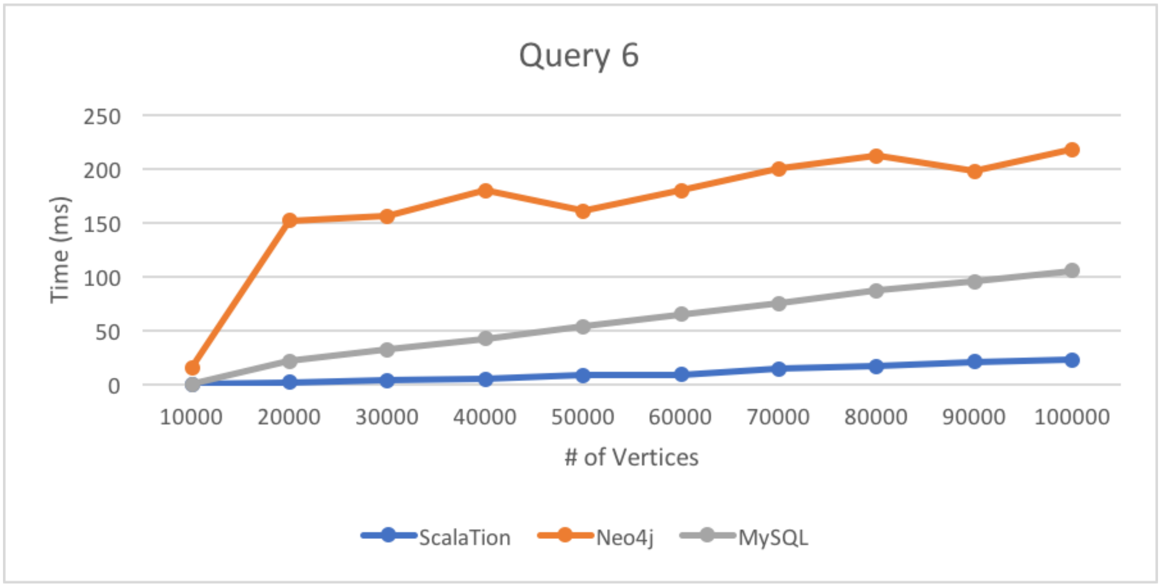

Figure 7.9: Running Time For Query 6

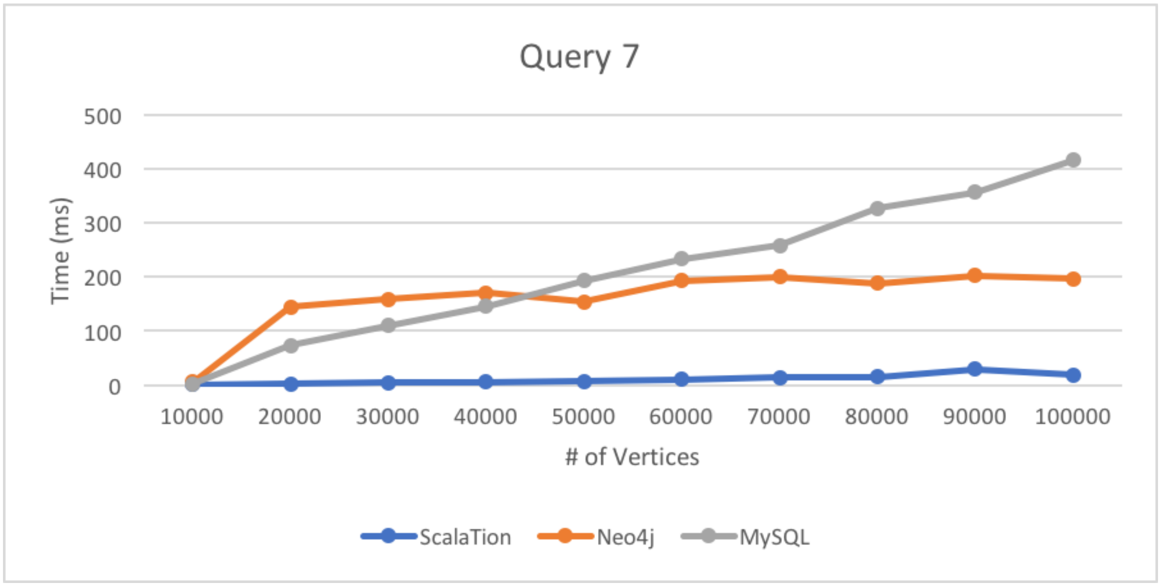

Figure 7.10: Running Time For Query 7

the MySQL query plan as shown in Figure 7.13 that the processing of SQL query involves

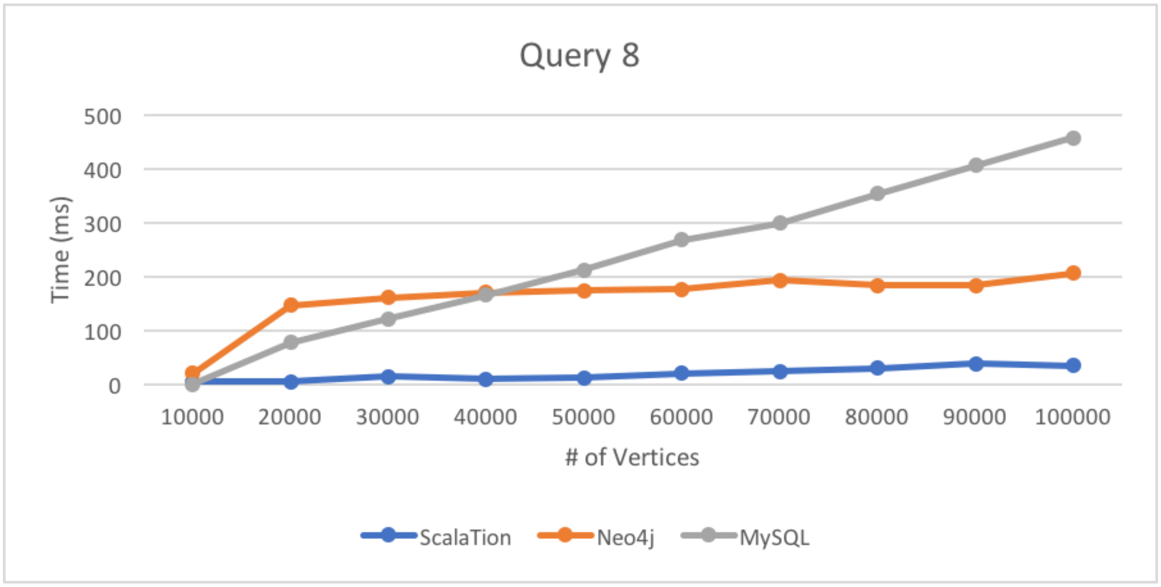

Figure 7.11: Running Time For Query 8

three joins which are expensive operations whereas in the graph database, it can be seen in the Figure 7.14 that the Neo4j query plan uses the expand operations.

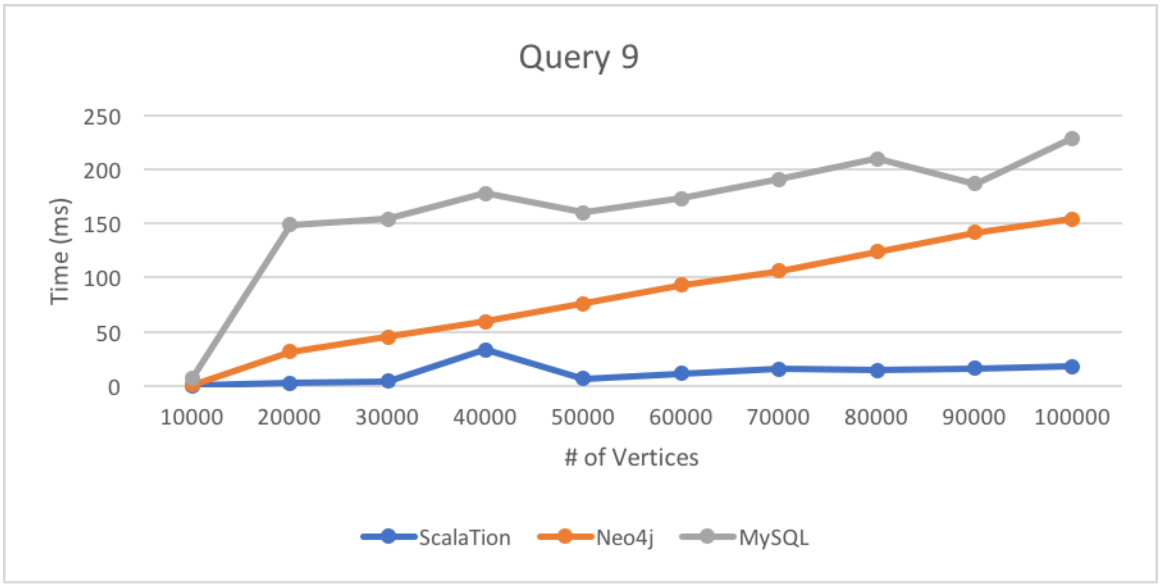

Figure 7.12: Running Time For Query 9

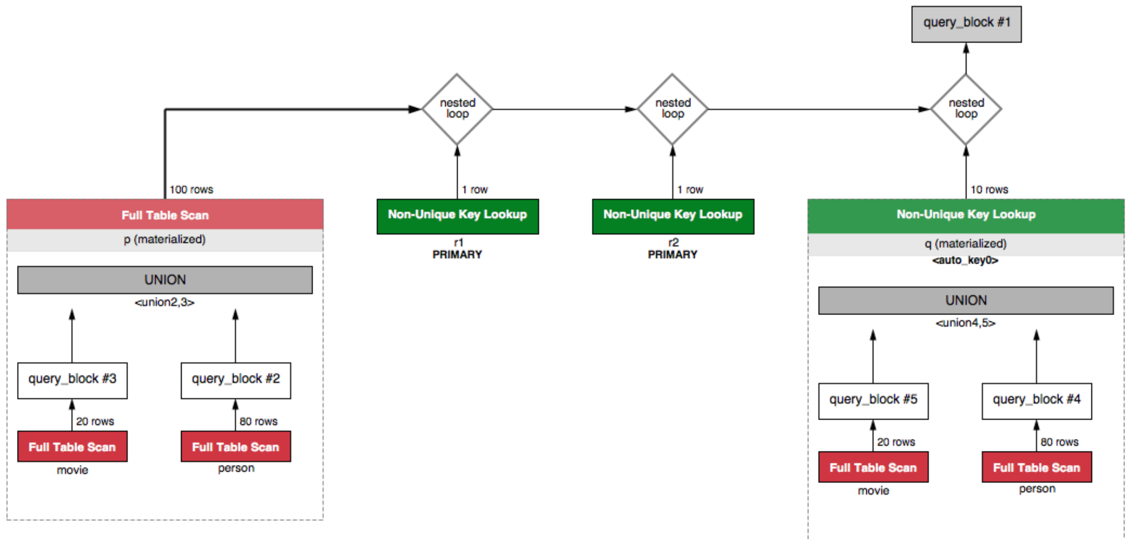

Figure 7.13: MySQL query plan for Query 9

According to [34], the differences between Neo4j and MySQL increases with the number of edges or joins, We can see the similar trend in our results as ScalaTion and Neo4j out perform MySQL for queries with joins and when the dataset size increases. [1] evaluates the query results with respect to Neo4j and MySQL. [35] compares Neo4j with other graph databases such as Jena, HyperGraphDB, DEX and the experiments shows that DEX and Neo4j are more efficient compared to other graph databases in the study.

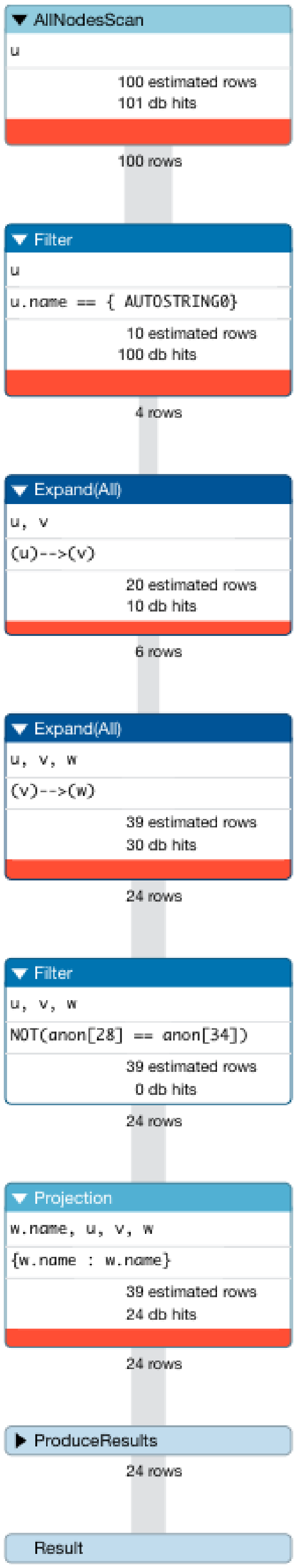

Figure 7.14: Neo4j query plan for Query 9

Chapter 8

Conclusions and Future Work

In this research we have implemented graph algebra operations which can be used for query processing in graph databases. We are able to process various types of queries by using the algebra operations defined. Some of the queries from Neo4j developer manual are executed in ScalaTion, Neo4j and MySQL. We have verified that the query results of all the databases are the same. We compared the query processing times of the graph databases ScalaTion and Neo4j, and also against MySQL. Our experiments show that when the query has joins, graph databases perform well compared to relational databases for the execution of queries. Among graph databases compared in the study, ScalaTion exhibits better performance.

For the declarative languages such as SQL, SPARQL and Cypher, the system's optimizer is responsible for buidling an optimized execution plan. But currently, our API does not have its own optimizer, therefore the developer is responsible for building the query plan for which we follow the query plan of Cypher. We have implemented a mechanism for query processing in graph database which makes use of algebra operations, and the order in which the query is evaluated is not fully automated, which we would like to automate in future. The data structure we used for storing graphs makes it possible to build indices similar to that of RDF-3x [24], which might make the query processing that is currently being implemented faster.

The join operation in graph databases most likely refer to the join of edges in which case the ordering of joins plays an important role in minimizing the query processing time, therefore we intend to develop a mechanism for determining the order of evaluation of different joins.

The simulation techniques defined in ScalaTion for pattern matching are very efficient compared to the Subgraph isomorphism which is used in most of the graph databases. We would like to combine the graph pattern matching techniques defined in ScalaTion with the query processing, as query processing involves matching of query graph against the data graph.

For processing the given query, either of the following two operations can be done: 1) The operations can be chained together (see Ebean for example), 2) A separate query language can be defined.

The algebra operations can be made parallel as future work, which will improve on the query processing times. Other functionalities in query processing such as aggregate functions can also be implemented.

Inorder to improve the evaluation, the results can be made more consistent if the queries are run on more number of iterations as we did only for 10 iterations.

A labeled directed multi-graph can be mapped to a RDF graph, so that the datasets available at W3C[1] can be made use for bench-marking. The recently developed HPC Scalable Graph Analysis Benchmark [36] can also be used.

[1] `https://www.w3.org/wiki/RdfStoreBenchmarking`

Bibliography

[1] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference.* ACM, 2010, p. 42.

[2] Tech. Rep. [Online]. Available: https://www.w3.org/RDF/

[3] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the Association for Information Science and Technology*, vol. 36, no. 6, pp. 35–41, 2010.

[4] A. Ghrab, O. Romero, S. Skhiri, A. Vaisman, and E. Zimányi, "Grad: On graph database modeling," *arXiv preprint arXiv:1602.00503*, 2016.

[5] K. Kaur and R. Rani, "Modeling and querying data in nosql databases," in *Big Data, 2013 IEEE International Conference on.* IEEE, 2013, pp. 1–7.

[6] M. Buerli and C. Obispo, "The current state of graph databases," *Department of Computer Science, Cal Poly San Luis Obispo, mbuerli@ calpoly. edu*, vol. 32, no. 3, pp. 67–83, 2012.

[7] G. M. Kuper and M. Y. Vardi, "A new approach to database logic," in *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems.* ACM, 1984, pp. 86–96.

[8] M. Levene and A. Poulovassilis, "The hypernode model and its associated query language," in *Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9).* IEEE, 1990, pp. 520–530.

[9] ——, “An object-oriented data model formalised through hypergraphs,” *Data & Knowledge Engineering*, vol. 6, no. 3, pp. 205–224, 1991.

[10] R. Angles and C. Gutierrez, “Survey of graph database models,” *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[11] R. Angles, “A comparison of current graph database models,” in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on.* IEEE, 2012, pp. 171–177.

[12] A. B. Ammar, “Query optimization techniques in graph databases,” *arXiv preprint arXiv:1609.01893*, 2016.

[13] P. Macko, D. Margo, and M. Seltzer, “Performance introspection of graph databases,” in *Proceedings of the 6th International Systems and Storage Conference.* ACM, 2013, p. 18.

[14] B. A. Eckman and P. G. Brown, “Graph data management for molecular and cell biology,” *IBM journal of research and development*, vol. 50, no. 6, pp. 545–560, 2006.

[15] H. He and A. K. Singh, “Graphs-at-a-time: query language and access methods for graph databases,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM, 2008, pp. 405–418.

[16] ——, “Query language and access methods for graph databases,” in *Managing and mining graph data.* Springer, 2010, pp. 125–160.

[17] J. Hölsch and M. Grossniklaus, “An algebra and equivalences to transform graph patterns in neo4j,” in *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, 2016.

[18] H. Thakkar, D. Punjani, S. Auer, and M.-E. Vidal, "Towards an integrated graph algebra for graph pattern matching with gremlin," in *International Conference on Database and Expert Systems Applications.* Springer, 2017, pp. 81–91.

[19] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "Pgql: a property graph query language," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems.* ACM, 2016, p. 7.

[20] J. A. Bondy, U. S. R. Murty *et al.*, *Graph theory with applications.* Citeseer, 1976, vol. 290.

[21] C. Gutierrez, C. Hurtado, and A. O. Mendelzon, "Foundations of semantic web databases," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* ACM, 2004, pp. 95–106.

[22] T. H. Cormen, *Introduction to algorithms.* MIT press, 2009.

[23] M. Schmidt, M. Meier, and G. Lausen, "Foundations of sparql query optimization," in *Proceedings of the 13th International Conference on Database Theory.* ACM, 2010, pp. 4–33.

[24] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 19, no. 1, pp. 91–113, 2010.

[25] C. J. Date, *An introduction to database systems.* Pearson Education India, 2006.

[26] E. F. Codd, *Relational completeness of data base sublanguages.* IBM Corporation, 1972.

[27] R. Cyganiak, "A relational algebra for sparql," *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, vol. 35, 2005.

[28] R. Angles and C. Gutierrez, "The expressive power of sparql," *The Semantic Web-ISWC 2008*, pp. 114–129, 2008.

[29] A. Gubichev, "Query processing and optimization in graph databases," Ph.D. dissertation, München, Technische Universität München, Diss., 2015, 2015.

[30] A. I. Maduko, "Graph summaries for optimizing graph pattern queries on rdf databases," Ph.D. dissertation, uga, 2009.

[31] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: An architecture for storing and querying rdf data and schema information," *Spinning the semantic web: Bringing the world wide web to its full potential*, vol. 197, 2003.

[32] J. A. Miller, L. Ramaswamy, K. J. Kochut, and A. Fard, "Directions for big data graph analytics research," *International Journal of Big Data*, vol. 2, no. 1, 2015.

[33] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[34] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops.* ACM, 2013, pp. 195–204.

[35] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazan, and J. Larriba-Pey, "Survey of graph database performance on the hpc scalable graph analysis benchmark," *Web-Age Information Management*, pp. 37–48, 2010.

[36] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, T. Meuse, and E. Robinson, "Hpc scalable graph analysis benchmark," *Citeseer. Citeseer*, vol. 2009, pp. 1–10, 2009.

Appendix A

SQL Queries

1. **select** name **from** VTable

2. **select** name **from** Movie

3. **select distinct** name **from** VTable **where** id **in** (**select** id1 **from** Relations **where** id2 **in**(**select** id **from** VTable **where** name='vertexLabel') **union all select** id2 **from** Relations **where** id1 **in**(**select** id **from** VTable **where** name='vertexLabel'))

4. **select distinct** name **from** Person **where** pid **in** (**select** id2 **from** Relations **where** id1 **in** (**select** id **from** VTable **where** name = 'vertexLabel'))

5. **select distinct** name **from** VTable **where** id **in** (**select** id2 **from** Relations **where** id1 **in** (**select** id **from** VTable **where** name='vertexLabel'))

6. **select** role **from** Relations **where** id1 **in** (**select** id **from** VTable **where** name = 'vertexLabel')

7. **select** name **from** VTable **where** id **in** (**select** id1 **from** Relations **where** role = 'eLabel1' **and** id2 **in** (**select** id **from** VTable **where** name = 'vertexLabel'))

8. **select** name **from** VTable **where** id **in** (**select** id1 **from** Relations **where** id2 **in** (**select** pid **from** VTable **where** name = 'vertexLabel') **and** (role='eLabel1' **or** role='eLabel2'))

9. **select** q.name **from** VTable p, Relations r1, Relations r2, VTable q **where** p.id=r1.id1 **and** r1.id2=r2.id1 **and** r2.id2=q.id **and** p.name = 'vertexLabel'

Appendix B

Log Graphs

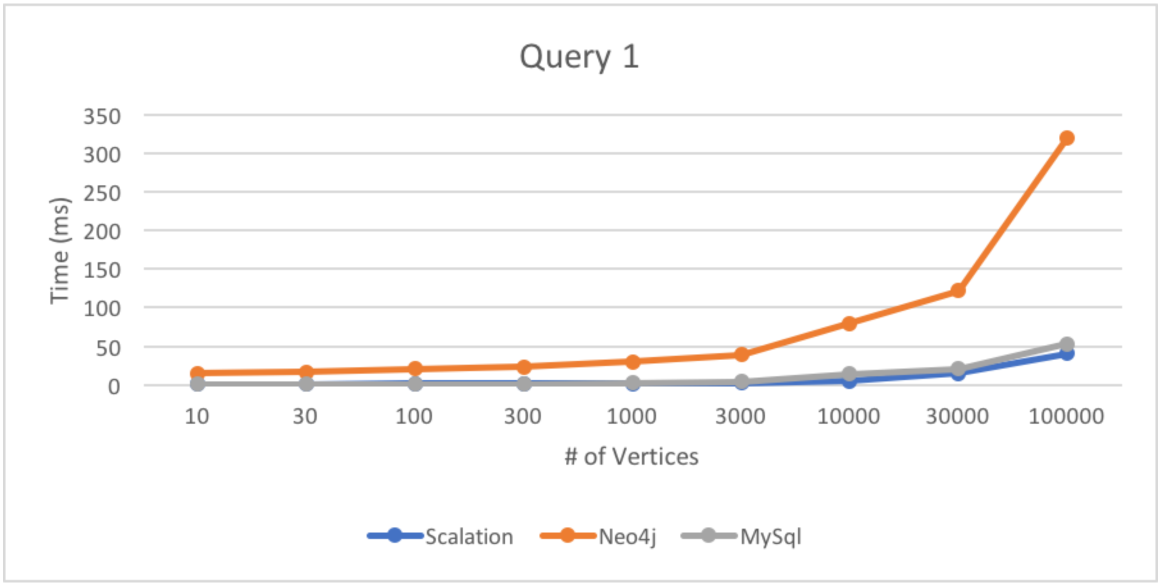

Figure B.1: Running Time For Query 1

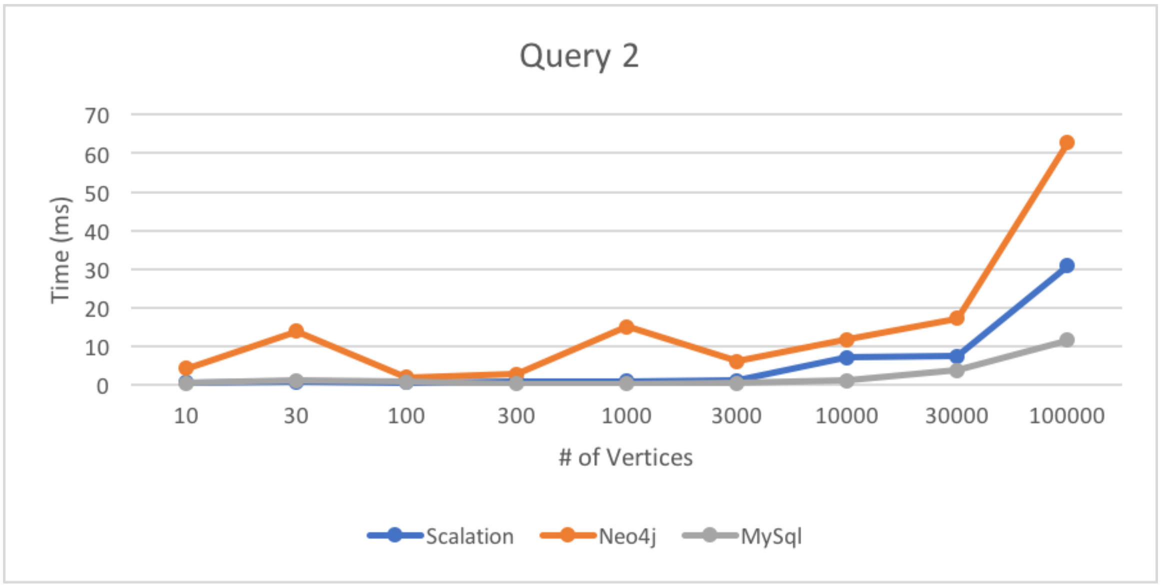

Figure B.2: Running Time For Query 2

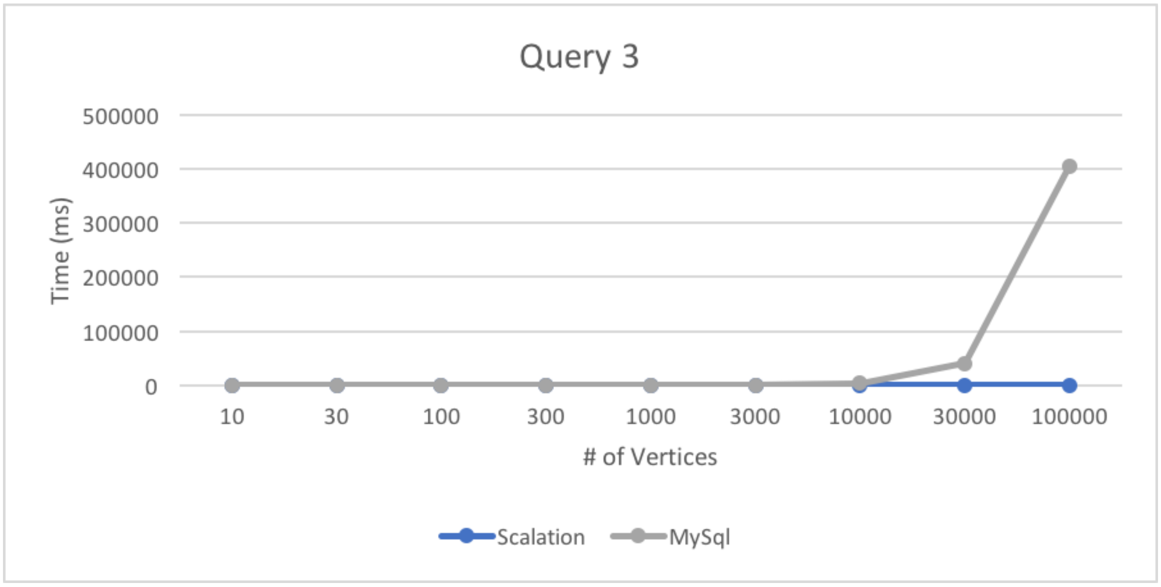

Figure B.3: Running Time For Query 3

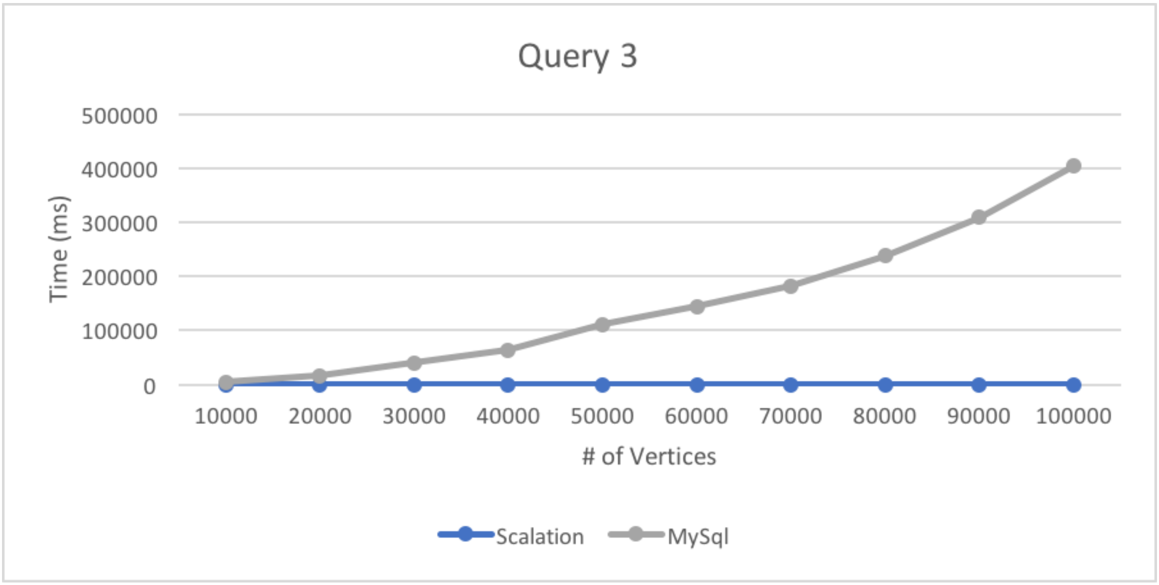

Figure B.4: Running Time For Query 3

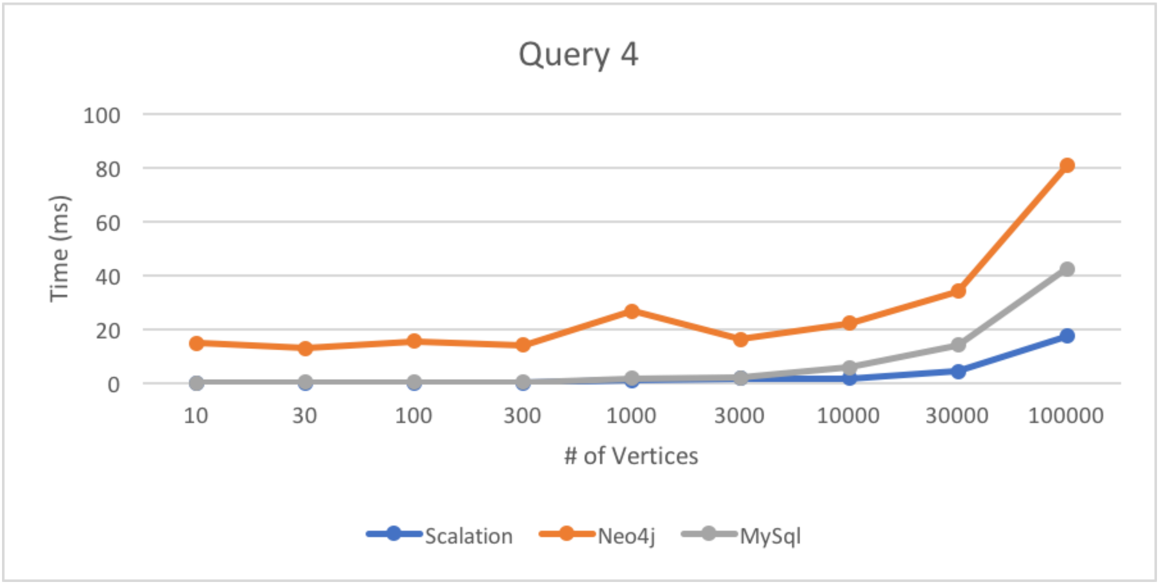

Figure B.5: Running Time For Query 4

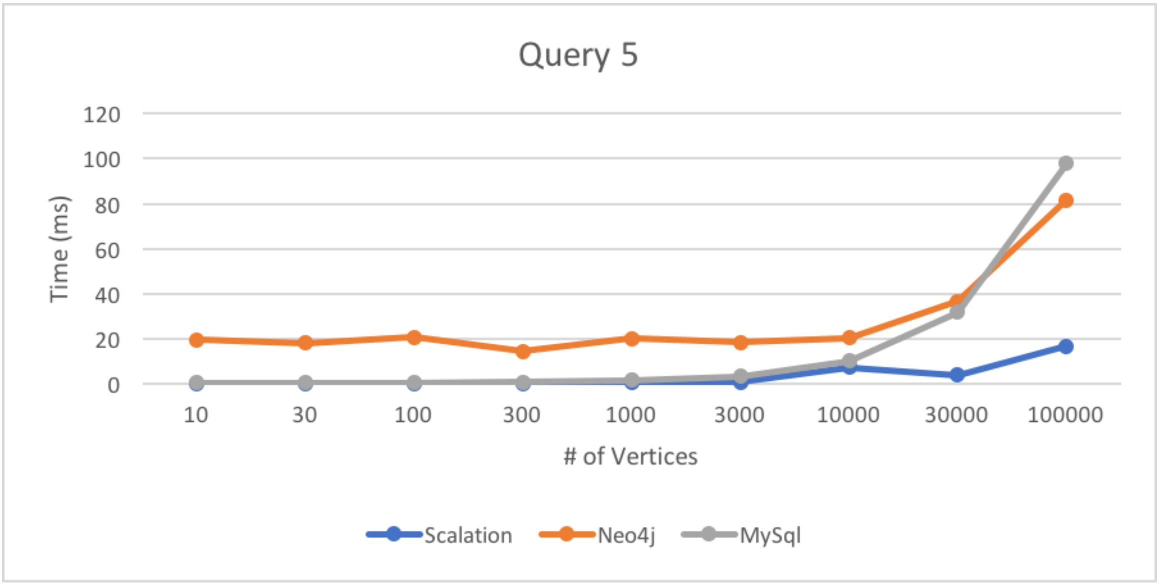

Figure B.6: Running Time For Query 5

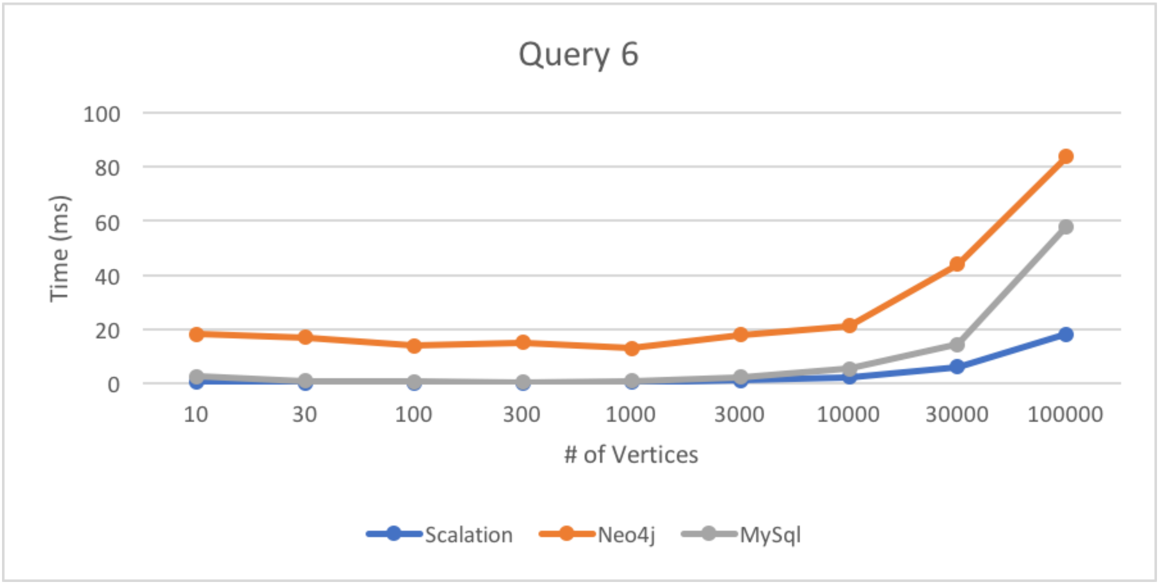

Figure B.7: Running Time For Query 6

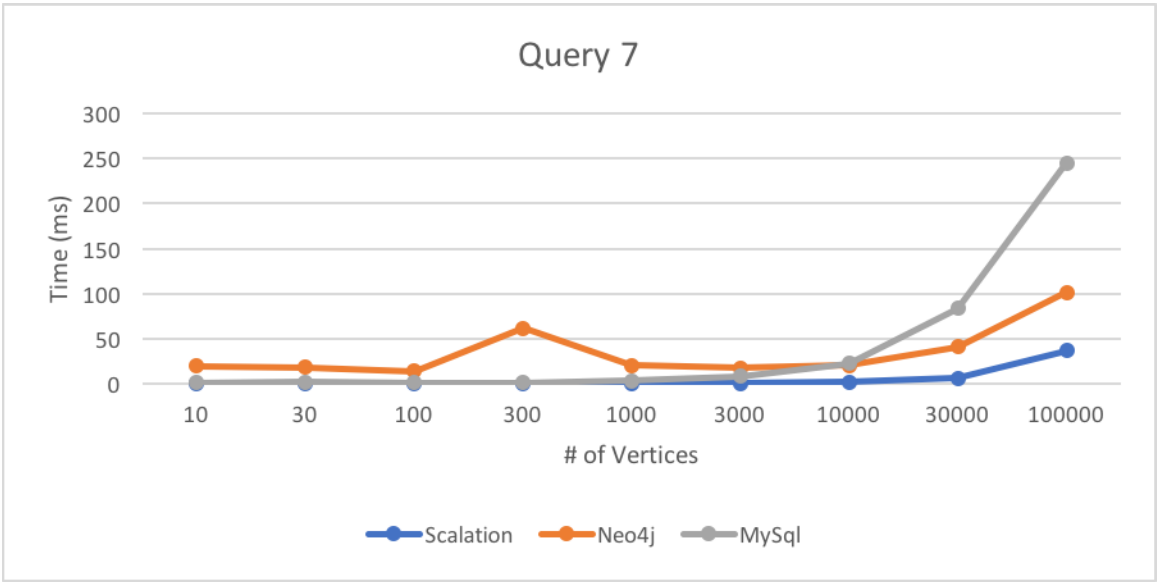

Figure B.8: Running Time For Query 7

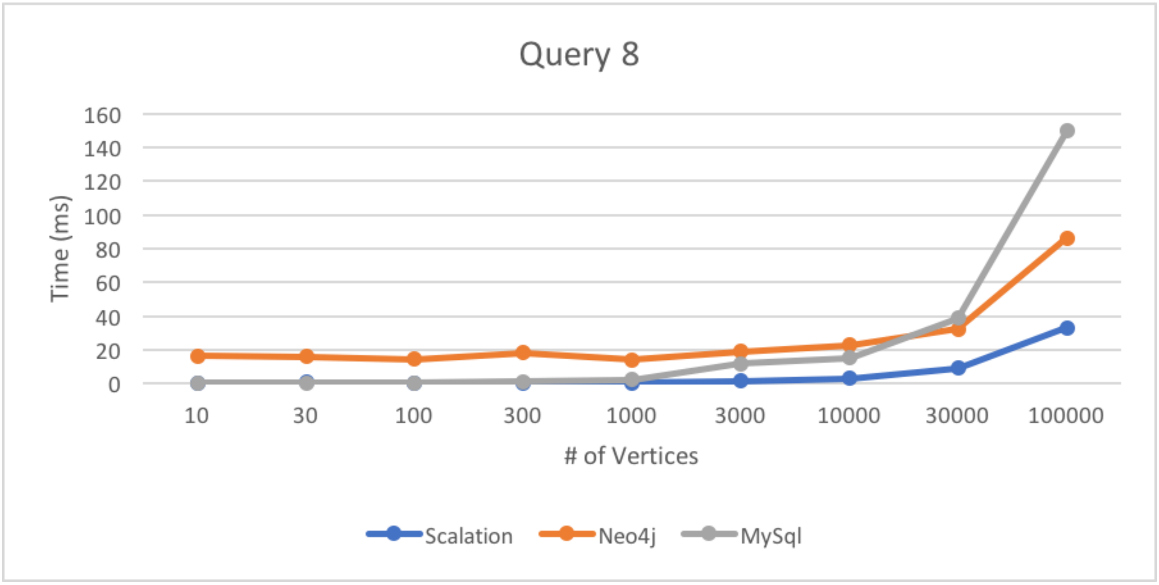

Figure B.9: Running Time For Query 8

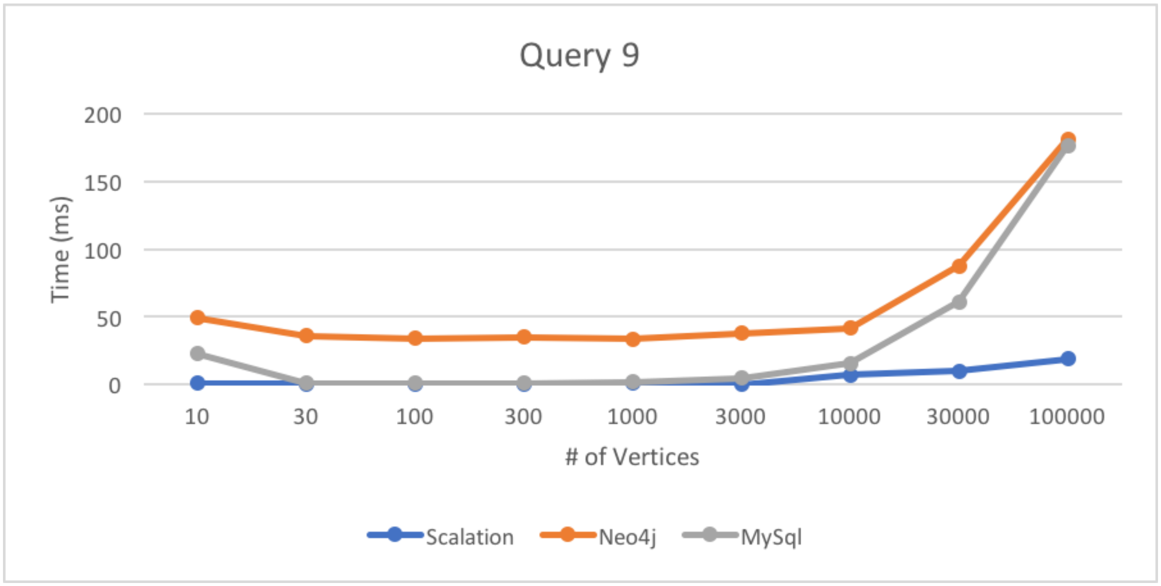

Figure B.10: Running Time For Query 9

Appendix C

Developer ToDo List

1. Parallel implementation of the graph algebra operators.
2. Define the Data structure for property graph using a heterogeneous data structure for storing the properties of vertices and edges.
3. Compare the performance of the graph algebra defined in ScalaTion with the algebra of other graph databases.
4. Id and schema for the vertices are newly added, the classes MuDualIso.scala, MuDualSim.scala, MuGraphSim.scala, MuStrictSim.scala, MuTightSim.scala has to be modified to check for the equality of schema and id's as well.
5. Index is built from (source vertex, edge) $\rightarrow$ target vertex. All possible permutations of indexes can be built, and also other indexes similar to that of RDF-3x.
6. Merge the two classes for random generation of graphs, MuGraphGen and RandomGraph.
7. Chaining of algebra operations for processing queries.
8. implement functions from Neo4j developer manual[1].
9. Equivalence rules for algebra operations like the rewriting rules in relational algebra for optimization.

[1] `http://neo4j.com/docs/developer-manual/current/cypher/functions/`