2016

Benchmarking Graph Databases with Cyclone Benchmark

Yuanyuan Tang
Iowa State University

Benchmarking graph databases with cyclone benchmark

by

Yuanyuan Tang

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Wallapak Tavanapong, Major Professor

Shashi Gadia

Pavan Aduri

Iowa State University

Ames, Iowa

2016

DEDICATION

I would like to dedicate this thesis to my husband Daixiang Mou without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and financial assistance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Recent years have seen advances in graph databases and graph database management systems (GDBMS). In a typical graph data model, a node represents an entity and an edge represents a relationship between two nodes. Nodes and edges typically have associated properties (attributes) and are assigned types for fast query response times. In the application domains where relationships are of importance, GDBMS increasingly gains popularity since the relationships can be explicitly modeled and easily visualized in a graph data model. To measure performance of GDBMS, a number of graph database benchmarks have been proposed. Nonetheless, these benchmarks are not yet as rigorous as those of relational database management systems (RDBMS). Inspired by Wisconsin Benchmark, we propose Cyclone Benchmark for measuring performance of graph databases in several aspects of which some have not been investigated in the literature. Our benchmark comes with (1) two data graph models: a simple model with all nodes of the same node type and a complex model with multiple node types, (2) data graph generation programs, and (3) Create, Read, Update, and Delete (CRUD) operations. The data graph generation programs create a graph structure and annotate values of node and edge attributes in the graph to allow for a study of the impact of attribute selectivity factors as well as a correlation between attributes. The programs generate a predefined graph structure, a random graph, or a Kronecker graph that has been shown to model real-world networks well. The read operations include several graph structure queries.

We measured the average execution times of the proposed CRUD operations on several synthetic graphs generated by the benchmark with a varying number of nodes from 1,000 to 1,000,000 nodes. The graphs were stored as graphs in Neo4j, a popular native GDBMS, and as relations in MySQL, a popular RDBMS. For most CRUD operations including the graph structure queries in our benchmark, MySQL was significantly faster than Neo4j.

CHAPTER 1. OVERVIEW

Relational databases have been widely used for over 30 years because of the excellent performance in managing large amounts of data. In relational databases, real-world entities and relationships are stored in relations or tables of predefined schemas. Each table contains one or more attributes in columns. Each row contains a unique instance of data identified by a non-empty set of attributes (a key). Users are able to retrieve the data in different ways. Commonly used relational database management systems (RDBMS) are Oracle Database(1), Microsoft SQL Server (2), MySQL(3), and IBM DB2(4).

Recently, the increase in data complexity becomes one of the most important issues. A graph is an adaptive and natural way to explicitly represent connectivity among entities modeled as nodes. An edge between nodes models a relationship between the corresponding entities. Graph data models for real-world networks such as social networks, transportation networks, protein-interaction, web mining, semantic webs, and even business networks have been gaining attention. In data graphs or property graphs, queries about relationships and patterns of relationships are important and prevelant. In relational data models, relationships are represented in tables that are implicitly linked together via foreign and primary keys, making it more difficult to express graph queries such as finding a shortest path between two nodes and finding a $k-$hop neighbors of a given node. The limitations of the relational data model have led to the development of a graph data model and graph database management systems (GDBMS).

A graph data model consists of nodes, edges, and attributes representing properties associated with the nodes or the edges. A typical node represents one entity. A typical edge represents a binary relationship between two entities. A more complex graph model allows for a hyper node and a hyper edge. A hyper node represents a set of nodes whereas a hyper edge represents relationships among multiple nodes. Example GDBMS are AllegroGraph (6),

InfiniteGraph (7), IBM System G Graph Databases(8), Hyper- GraphDB (9), Neo4j (10), and Titan (11). Distributed graph processing libraries such as Apache GraphX(12) and Googles Pregel(13) have been introduced.

1.1 Graph Database Benchmarks

Even if graph databases are designed to persistently store graph data and manage them, debates are still on-going whether GDBMS using a native graph storage engine such as Neo4j or RDBMS as a storage engine is more efficient to support graph queries (14), (15),(16),(17),(18),(20). Comparison of GDBMS and several GDBMS benchmarks have been proposed (21), (22), (23), (24), (25), (26), (27), (29), (30). Despite these attempts, to the best of our knowledge, there are no graph database benchmarks that allow for a scalability study of query performance with different selectivity factors. A selectivity factor of a query in RDBMS is the ratio of the number of output rows as a result of the query to the number of input rows. The same concept can be extended to GDBMS where a node selectivity factor is the ratio of the number of output nodes as a result of the query to the total number of input nodes. An edge selectivity factor is the ratio of the number of output edges to the total number of input edges.

1.2 Contributions

Our contributions are as follows.

- We propose Cyclone Benchmark with three key features. First, the benchmark has two graph data models. A simple model has only one node type for all the nodes. The attributes and the attribute domains (a set of all possible values for the attribute) are designed to easily compute the corresponding selectivity factor in order to evaluate the performance impact. Some attributes have indexes built on them to study the impact of indexing. The other model has seven node types, which enables a study involving the connectivity pattern between different node types. Some attributes are correlated to allow for a study of the impact of correlation between attributes. Second, the benchmark comes with a program to generate synthetic data graphs. The program performs two

functionalities as follows. Given input parameters, it generates a graph structure such as a random graph, a Kronecker graph, or a pre-defined graph structure. Second, its data generator assigns attribute values to nodes and edges of the graph. Third, the benchmark has several major types of common Create, Read, Update, and Delete (CRUD) operations on graph databases. They are bulk insertion, bulk update, bulk deletion, bulk selection with a desired selectivity factor, bulk selection with multiple edge types, and graph structure queries including queries focusing on the connectivity pattern of different node types.

- We compare the performance of Neo4j and MySQL as the storage engine for two graph datasets of random graphs and Kronecker graphs. In evaluating our graph benchmark, we chose Neo4j and MySQL for performance evaluation because of the following. Neo4j is a popular GDBMS whereas MySQL is a popular RDBMS. Neo4j has more important data management features compared to other GDBMS. Furthermore, it has been used in practice in several domains in matchmaking, network management, software analytics, scientific research and organizational project management. Neo4j supports a declarative query language called Cypher as well as Java API for faster graph processing. It uses native graph query and storage engines and supports indexing, cost-based query optimization, and transactions.

1.3 Organization

The remainder of the thesis is organized as follows. In Chapter 2, we discuss relatated works on RDBMS and GDBMS benchmarks. Chapter 3 presents the proposed graph models and CRUD operations in Cyclone Benchmark. Chapter 4 provides the details of the implementation and Chapter 5 provides the experimental result of the benchmark on Neo4j and MySQL. Chapter 6 presents the conclusion and the description of the future work.

CHAPTER 2. REVIEW OF LITERATURE

In this chapter, we review some well-known benchmarks for relational databases, especially Wisconsin Benchmark, which inspired our initial design of our benchmark and benchmarks for graph databases.

2.1 Benchmarks for RDBMS

Wisconsin benchmark (31) was the first successful RDBMS benchmark. It consists of query suites and relation schemas that allow for studying of different selection and join selectivity factors. As shown in Figure 2.1, a typical relation in Wisconsin benchmark consists of a number of columns of either integer type or string type. The domain for each column is defined. For instance, the attribute "unique1" has the attribute values ranging from 0 to the number of tuples - 1 and ensures that each value is unique. The values of‘ the attribute ‘two" are either 0 or 1; the values of the attribute "four" range from 0 to 3; these values are randomly assigned to tuples. To issue a query with a certain selectivity factor, it is easily done by using different attribute values. For instance, the predicate "four = 3" returns 25% of the tuples regardless of the cardinality of the relation. This benchmark was popular for evaluating early commercial RDBMS and parallel RDBMS.

Later, Transaction Processing Performance Council's TPC-C benchmark (32) standard was developed. It supports concurrent transactions that were not supported in the Wisconsin benchmark. TPC-C involves a mix of five concurrent transactions of different types and complexity either executed immediately or queued for deferred execution. The database is comprised of nine types of relations with a wide range of record and population sizes. It simulates a complete computing environment where a population of users executes transactions against the database

Attribute Name	Range of Values	Order	Comment
unique1	0-9999	random	candidate key
unique2	0-9999	random	declared key
two	0-1	cyclic	0,1,0,1,...
four	0-3	cyclic	0,1,2,3,0,1,...
ten	0-9	cyclic	0,1,...,9,0,1,...
twenty	0-19	cyclic	0,1,...,19,0,1,...
hundred	0-99	cyclic	0,1,...,99,0,1,...
thousand	0-999	cyclic	0,1,...,999,0,1,...
twothous	0-1999	cyclic	0,1,...,1999,0,1,...
fivethous	0-4999	cyclic	0,1,...,4999,0,1,...
tenthous	0-9999	cyclic	0,1,...,9999,0,1,...
odd100	1-99	cyclic	1,3,5,...,99,1,3,...
even100	2-100	cyclic	2,4,...,100,2,4,...
stringu1	-	random	candidate key
stringu2	-	cyclic	candidate key
string4	-	cyclic	

Figure 2.1 Attribute specifications of Wisconsin benchmark(31)

being benchmarked. The benchmark is centered around the principal activities (transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stocks at the warehouses. While the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but rather represents any industry that must manage, sell, or distribute a product or service. TPC-C provides a performance measurement in a single unit, transactions per minute (tpmC), making it easy to compare various RDBMS products.

Other RDBMS benchmarks include Bristlecone (33), or Open Source Development Lab Data Base Test Suite (OSDL-DBTS) (34). These benchmarks focus on business applications suitable for RDBMS.

2.2 Benchmarks for GDBMS

With growing interests in GDBMS, several GDBMS benchmarks were proposed. LinkBenchmark (29) contains traces from the production database of Facebook's social graph data. The benchmark covers standard insert, update and deletion operations as well as key lookup, range and count queries. However, it missed a important aspect of queries which was related to graph

structure queries such as shortest paths and centrality. This weakness made this benchmark vulnerable to other benchmarks which covered queries related to graph structures.

The Linked Data Benchmark Council (LDBC) (35) proposes Social Networks Benchmark (SNB) with three categories of workloads: interactive workload, business intelligence workload, and graph analytic workload. Only the interactive workload and business intelligence workload were completed. The remaining graph analytics workload is still under development. The SNB benchmark provides a data generator that generates a synthetic data set of a social network akin to Facebook. The main entities in this data set are person, forum, posts, comments, tags, universities and companies, cities, countries and continents. The relationships could be between different types of entities or among the same type of entities. For instance, person could be a friend with another person or a person could publish a forum, a post, or a comment. A forum, a post, or a comment should have an associated tag. The benchmark only uses read-only queries. Bulk insertion, deletion and update queries are absent.

HPC Scalable Graph Analysis Benchmark (36) consists of four operations on a weighted directed graph with node degrees that follow a power-law distribution. The data generator of this benchmark generates nodes and edges with a positive weight value. The benchmark supports four types of operations: bulk insertions, retrieval of edges with the largest weight, extraction of a k-hop path from a given edge in an edge set, and extraction of a set of edges with the highest betweenness centrality metric that identifies vertices of key importance along shortest paths of the graph. The benchmark does not include many other graph operations such as updates. As the nodes do not contain attributes and the edges have only one attribute: weight, this benchmark does not have queries which focus on selection of node and edge attributes, which is important for GDBMS. The graph generated is a Recursive MATrix (R-MAT) power-law graph and is scalable by the adjusting input parameters.

McColl et al. (24) evaluated eleven open-source GDBMS (Neo4j, Titan, OrientDB, InfoGrid, FlockDB, ArangoDB, InfiniteGraph, AllegroGraph, Dex, Graphbase, and HyperGraphDB) using a single-source shortest path algorithm implemented as a level-synchronous parallel breadth-first graph traversal, Shiloach-Vishkin connected components algorithm, PageRank algorithm, parallel edge insertions and deletions (24). Four data sets (1K node, 8K edges), (32K

nodes, 256K edges), (1M node with 8M edges) and (16M nodes, 256M edges) were generated using the R-MAT generator (37). The data generator of the benchmark does not generate attributes of node or edges. Therefore, the benchmark does not have queries on node or edge attributes.

XGDBench (25) is an extension of the Yahoo Cloud Serving Benchmark for graphs stored on a cloud. The data generator of this benchmark generates a synthetic undirected graph with a node degree distribution following the power-law distribution and with node attributes of binary values. The generated graph does not have edge attributes. The queries are centered around a vertex such as reading a vertex and its attributes, inserting a vertex, updating all attributes of a vertex, deleting a vertex, scaning neighbors of a vertex, and a breadth-first traversal given a vertex.

Waterloo Graph Benchmark (WGB) (26) consists of a data generator that generates a graph with the power-law node degree distribution. The benchmark assumes that a shortest path between any two nodes in the graph is already calculated. WGB workload consists of read-only queries (i.e., find matching nodes or edges, find k-hop neighbors using the known shortest path, reachability between any two nodes, reachability graph pattern matching), update operations that change the graph structure and node or edge attributes, and iterative queries involving computation over multiple passes of data. Two iterative queries PageRank and Clustering are included. WGB benchmark does not explicitly focus on characteristics of edge or node attributes to obtain a desired selectivity factor, but does indicate that it could be extended to support such a study.

Vicknair *et al.*(18) used a directed acyclic graph (DAG) as the data set. They only compared graph structure queries. M. Miller *et al.* (28) compared performance between Neo4j and PostgreSQL on the shortest path algorithm on the OpenStreetMap road network. Jindal compared PageRank and Shortest Paths performance of Neo4j—a native GDBMS, MySQL—a row oriented RDBMS, Vertica—a column-oriented RDBMS, and VolotDB—a main-memory RDBMS (16).

Sotirios *et al.*(19) compared the performance of several GDBMS on the problem of community detection, which applied a well-known community detection algorithm for modularity

optimization, the Louvain method, as well as three other supplementary workloads such as the creation and traversal of the graph. They compared Titan, OrientDB and Neo4j on both synthetic and real networks.

Table 2.1 shows a comparison of popular benchmarks for graph databases. We compare those benchmark from both queries they have and data set they use, as well as the workload type.

Table 2.1 Comparison of Graph Database Benchmarks

	Fields	LDBC	HPC	McColl	XGDBench	WGB	Sotirios
queries	Bulk Insertion/Deletion	×	×	✓	×	×	✓
	Single Insertion/Deletion	✓	×	×	✓	✓	✓
	Bulk Selection/Update	×	×	×	×	×	×
	Single Seletion/Update	✓	×	×	×	✓	×
	Shortest Path	×	×	✓	×	×	✓
	BFS/DFS	×	✓	✓	×	×	×
	Multiple Node Types	✓	×	×	×	×	×
	Centrality/Component	✓	✓	✓	×	✓	✓
	K-hop Neighbours	×	×	×	×	✓	×
data	Scalable	✓	✓	✓	×	✓	×
	Synthesized	✓	✓	✓	×	✓	✓
	Directed	✓	✓	×	×	×	×
	Weighted	×	✓	×	×	×	×
	Transaction	×	×	×	✓	×	×

CHAPTER 3. PROPOSED WORK: CYCLONE BENCHMARK

In this chapter, we present the design of the Cyclone Benchmark for graph databases. The benchmark consists of two graph models, the data graph generator program, and a workload of common CRUD operations.

We chose to have our benchmark generate a synthetic data graph to be able to control the sizes of the data sets to enable a study of the scalability of the DBMS being evaluated. Furthermore, we could model characteristics of the real-world data and common operations on the data so that the results from the benchmark can be generalized to real-world problems that possess the same characteristics. Lastly, we could investigate other aspects that are important to the DBMS performance that may not be present in a particular real-world data set.

The first graph data model contains only one node type and five edge types as shown in Figure 3.1. We aim for the design of the attributes of nodes and edges of the generated data graph to enable studies of the impact on the DBMS performance given different selectivity factors of nodes and edge attributes, indexing versus no-indexing, and different data types. For this model, the benchmark can generate two graph structures: random graphs and Kronecker graphs that simulates the power-law distribution of node degrees in real networks. The workload for this model consists of seven major types of common CRUD operations on graph databases. They are bulk insertion, bulk update, bulk deletion, bulk selection with a desired selectivity factor, bulk selection with multiple edge types, and graph structure queries.

The second graph data model has seven node types and ten edge types as shown in Figure 3.2. The distribution of number of nodes for each node type and the distribution of edges in each edge type follows a Zipf distribution with a tunable parameter, which we will discuss in detail later. A pre-defined graph structure is used to connect different node types. We designed a query suite that focuses on the connectivity patterns among these node types.

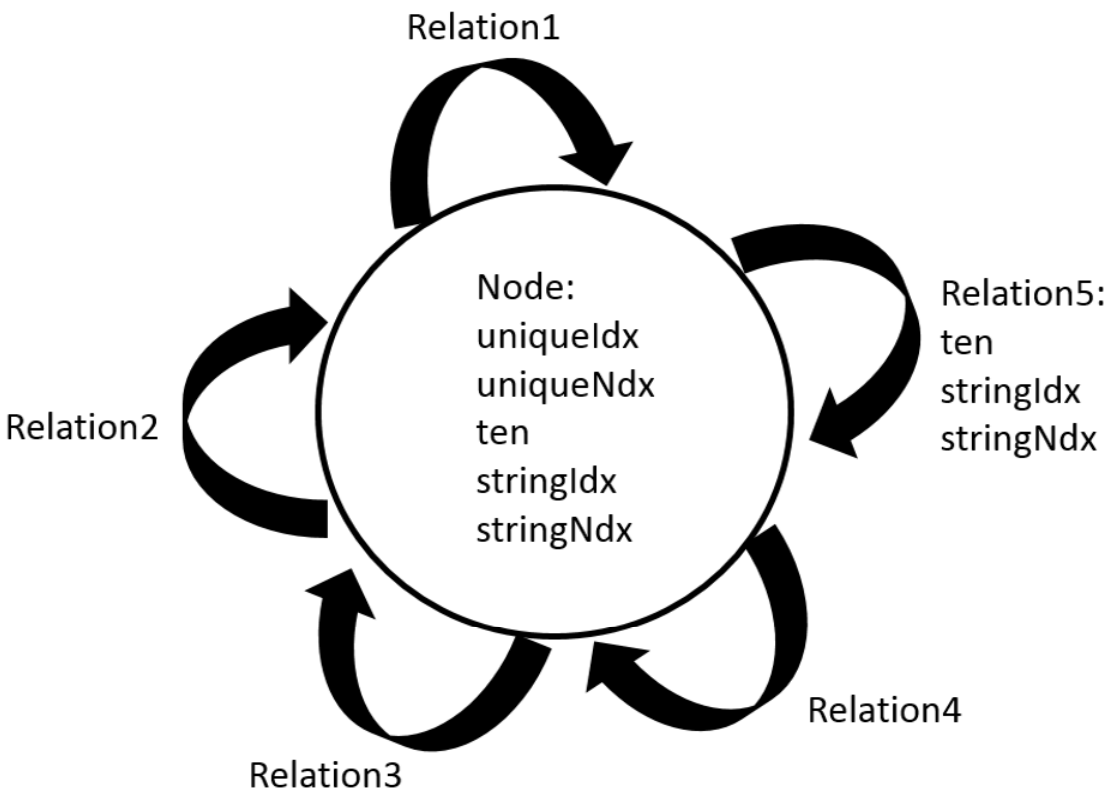

Figure 3.1 Single node type graph model with five different edge types; each edge type has the same attributes

3.1 Graph Model with Single Node Type

Our graph model (schema) is shown in Figure 3.1. The model consists of one type of nodes and five types of edges. We name the edge types "relation" to model different types of relationships between nodes commonly seen in real-life networks.

The data graph generator can create an Erdos-Renyi random graph (40). It also accepts Kronecker graphs and graphs with the power-law distribution of node degrees as input. We will discuss the detail of the implementation in Chapter 4. These two latter graphs were shown to model real-world networks well. The data graph generator creates attributes of nodes and edges, which allows for studying the impact of different selectivity factors explicitly. This distinguishes our benchmark from all the other existing graph benchmarks.

Table 3.1 Attributes of Nodes

Attribute Name	Type	Range of Values	Characteristics	Comment
uniqueIdx	Integer	$0 \sim \|V\| - 1$	Unique	Primary key, Index
uniqueNdx	Integer	$0 \sim \|V\| - 1$	Unique	Candidate key, No Index
Ten	Integer	$0 \sim 9$	Randomly assigned	No Index
StringIdx	String	"aaaa" $\sim$ "jjjj"	Randomly assigned	Index
StringNdx	String	"aaaa" $\sim$ "jjjj"	Randomly assigned	No Index

The workload consists of bulk selection, insertion, deletion, and update with different selectivity factors, bulk selection with multiple edge types, and graph structure queries.

3.1.1 Data Generator for Attributes of Nodes and Edges

Given the underlying graph $G = \{V, E\}$ where V is a set of nodes and E is a set of edges, we assign all nodes in the graph to the same node type. $|V|$ and $|E|$ denote the number of nodes and edges in the graph, respectively. All edges in the graph are divided evenly among the five edge types. Therefore, the number of edges with each edge type is 20% of the total number of edges in the graph. It is easy to expand the model to include more edge types. As a real-fie example, the nodes in the graph model could represent different cities and different edge types could represent the different ways for traveling from one city to another. Traveling by plane, by train, by car, by bicycle and on foot could be five different edge types.

To generate actual values of node attributes and edge attributes, we adapted the idea from the Wisconsin benchmark to generate the attribute values where different selectivity factors can be easily specified. Table 3.1 summarizes the attributes and the domains of these attributes.

For each node, the values of attributes *uniqueIdx* and *uniqueNdx* are unique integers between 0 and $|V| - 1$ in the graph. We use two different attributes to enable exploration of the impact of indexing on node attributes. We set *uniqueIdx* to have an associated index whereas *uniqueNdx* is not associated with any index. Since the values of *uniqueIdx* and *uniqueNdx* are unique, we can study a very fine grained selectivity factor such as $1/|V|$ or $2/|V|$, and so on.

Table 3.2 Attributes of Edges

Attribute Name	Type	Range of Values	Characteristics	Comment
Ten	Integer	$0 \sim 9$	Randomly assigned	No Index
StringIdx	String	"aaaa" $\sim$ "jjjj"	Randomly assigned	Index
StringNdx	String	"aaaa" $\sim$ "jjjj"	Randomly assigned	No Index

The attribute *ten* is an integer randomly chosen from [0,9]. This is to study the impact of performance given coarse selectivity factors on node attributes. For instance, to have a 10% node selectivity factor, we choose any one of the attribute values. For a 20% node selectivity factor, we choose any two of the attribute values, for instance, $ten = 0$ or $ten = 1$. The attribute $stringNdx$ is a string randomly chosen from "aaaa" to "jjjj," inclusive. The attribute $stringIdx$ has values similar to the attribute $stringNdx$, but has an index associated with it. The integer and string data types are two most common types of attributes in graph databases. We want to see whether the underlying DBMS performs differently for integers and strings or not.

Each edge has three attributes (*ten*, $stringNdx$, and $stringIdx$) as shown in Table 3.2 to control different edge selectivity factors. The attribute *ten* is an integer randomly chosen from [0,9]. similar to the attribute *ten* of a node. For example, the query choosing edges in $relation1$ with the edge attribute $ten = 0$ returns 10% of the edges in that relation. A query with the edge attribute $ten < 2$ returns 20% of the edges in that relation.

3.1.2 Workload

The workload currently consists of eleven queries categorized into six major categories: bulk selection, insertion, deletion, update with selectivity factor, bulk selection with multiple edge types, and structure queries.

3.1.2.1 BI:Bulk Insertion

We insert new nodes and edges that are not duplicates of existing nodes and edges in the graph. The numbers of new nodes and new edges are specified in terms of the percentage of number of existing edges and nodes, respectively before bulk insertion begins.

3.1.2.2 BU:Bulk Update

This operation includes updates of node attributes and edge attributes. The update operation changes the value of the attribute *ten* for nodes and edges. By using a different expression $ten < x$ or $ten > x$ where x is a certain value, this operation could touch different numbers of nodes or edges, resulting in a different selectivity factor. For instance, the expression $ten < 2$ results in 20% of the nodes or edges being updated.

3.1.2.3 BD:Bulk Deletion

This operation deletes all the new edges and new nodes added by the BI operation. Therefore, it should be performed after BI.

3.1.2.4 BS-S:Bulk Selection with Selectivity Factor

This category consists of bulk selection of nodes and bulk selection of edges in *relation*1 based on attribute values for different selectivity factors.

To study GDBMS indexing performance on an integer attribute, we use a condition on the attribute *uniqueIdx* to get different selectivity factors ranging from 0% to 100%. For the evaluation without indexing on integer, we use a condition on the attribute *uniqueNdx* to indicate a different selectivity factor.

Similarly, for evaluation involving strings without any index, we use *stringNdx*. Because the values of the strings are one of "aaaa" to "jjjj", we can query with different selectivity factors in multiple of 10. For instance, a query with 10% selectivity factor is done by specifying the condition *stringNdx*="aaaa'.' To get a higher selectivity factor, choose more values to include in the conjunctive condition.

3.1.2.5 BS-M: Bulk Selection involving Multiple Relations

- Two relation bulk selection ($BS - M : 2R$): Given edge types x and y, find all nodes with both out-going edge type x and edge type y. A real-life example for this query is to find all people who play sports and attend a college.

- Find orphan nodes ($BS-M:Orphan$):We have five queries involving finding different classes of orphan nodes. They are orphan nodes not having incoming edges of "relation1" type, orphan nodes not having incoming edges of "relation1" and "relation2", orphan nodes not having incoming edges of "relation1", "relation2", and "relation3", orphan nodes not having incoming edges of "relations 1 to 4", and orphan nodes without any incoming edges. A real-life example is to find all the cities that cannot be reached by plane or all the cities that cannot be reached by train or by plane.

3.1.2.6 SQ: Graph Structure Queries

We use a capital variable such as V to denote a node type and E for an edge type. We use a variable v to denote a particular node and e to denote a particular edge.

- Find k-hop neighbors of a given node ($SQ:k-Hop$)

  Given a start node v and an edge type E, find all the nodes at a distance of k edges in an out-going edge type E from the start node v. For instance, find a friend of friend of friend of a given person.

- Rank of neighbors ($SQ:Ranking$)

  Given a start node v and two edge types E_i and E_j, return k neighbors of v with the edge type E_i with the highest number of out-going edges of edge type E_j.

  For instance, list k friends of a given person ranked by the number of sports each friend plays.

- Find a shortest path among a given pair of nodes ($CN:SP$)

  Given a start node v_i and an end node v_j, find a shortest path from v_i to v_j and return the length of the path.

- Compute a degree centrality of a given edge type ($CN:Central$).

  Return a single node with the highest number of incoming edges of a given edge type. For instance, find an article that most number of people cites.

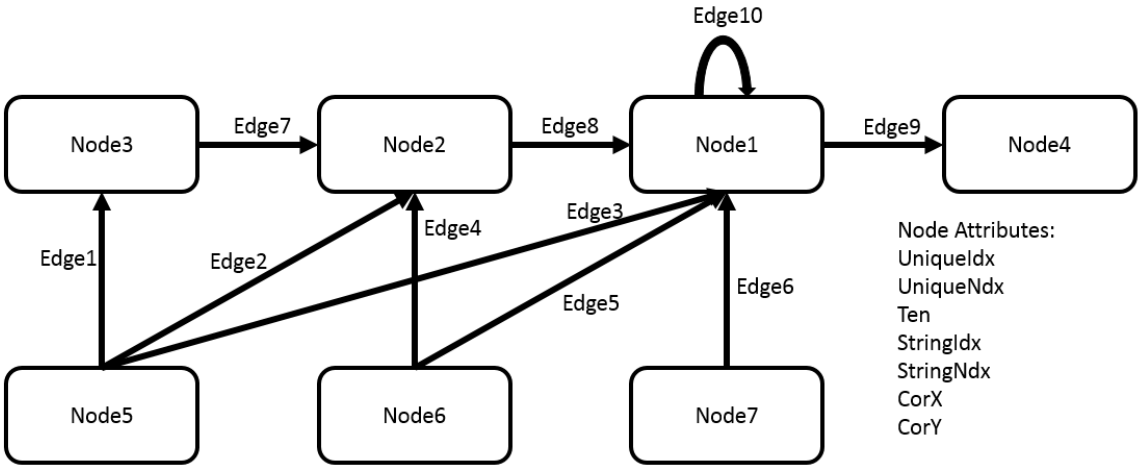

Figure 3.2 The complex graph model with seven different node types and ten different edge types

- Find all nodes with k out-going degree of a given edge type ($CN : degree$) For example, it can be used to find people who have the same number of friends.

3.2 Graph Model with Multiple Node Types

The graph model with a single node type is suitable for queries with selectivity factors. However, it cannot model complex relationship types in reality. For example, a common social network usually contains several types of entities such as persons, tags, messages and photos, which are connected by multiple relation types between different entity types. Therefore, a good benchmark should also support queries involving multiple node and relationship types.

3.2.1 Graph Model

The multiple node type graph model contains seven different node types and ten different edge types. Each node type and edge type has the same attribute types for scalability study in Figure 3.2.

The design guideline for the complex graph model is to make the graph structure with connections among different node types. For instance, Node5 has three different outgoing edge

types. It could represent a kind of entity which has three different relationships with other entities. The long link in our graph model from Node5->Node3->Node2->Node1->Node4 could enable us to study the connectivity path which involves as many as five node types. We believe it is enough for most cases in real world networks (35).

In order to make the graph model flexible, we apply a Zipf distribution to compute the number of nodes for each node type. The Zipf distribution can be used to account for the relative popularity of a few members of a population and the relative obscurity of other members of a population. Examples include the following. The world population lives in several large cities, a greater number of medium-sized cities, and a vast number of small towns. Another example is that a few websites get lots of hits; a greater number of websites gets a moderate number of hits. A vast number of websites hardly get any hits at all. When the Zipf factor is zero, all node types get the same number of nodes. If the Zipf factor is one, the number of nodes for each node type skews significantly, leaving many more number of nodes for one node type while the rest have few number of nodes. Given the Zipf factor and the total number of nodes as parameters, the benchmark determines the number of nodes for each node type according to the Zipf distribution.

Among ten different edge types, "Edge10" is the only one connecting nodes in the same node type. We use randomness when generating edges of "Edge10" type. For the other edge types, we also apply a Zipf distribution. Take "Edge8" as an example. We first evenly divide "Node1" into 100 groups where each group has a different rank. For each "Node2", we choose a node in "Node1" and generate an "Edge8" with a probability which follows the Zipf distribution. In other words, the higher ranked nodes have a higher probability to be and vice versa. It is also the common case in reality. For example, "Node2" type represents cities and "Node1" type represents the people. Not all cities have the same population. Some cities have more people and some cities have fewer people.

3.2.2 Data Generator for Attributes of Nodes and Edges

Besides the application of the Zipf distribution, we also use the same data generator as discussed in Chapter 3.1.1 to generate the attributes for nodes and edges. Each node has seven

attributes: UniqueIdx, UniqueNdx, Ten, StringIdx, StringNdx, CorX, and CorY. Each edge has three properties: Ten, StringIdx and StringNdx. All these attributes follow the same design as in the simple graph model with a single node type except CorX and CorY. The values of the two attributes are of "double" type and are generated in pairs. Given a correlation coefficient which shows how strongly pairs of variables are related, we artifact a set of pair of double numbers and assign the pair randomly to a node.

3.2.3 Queries

Due to different edge types in this graph model, we design a workload involving a series of complex queries. These queries touch a significant amount of data, often a two-step hop in Node1 with Edge10 and a two-step hop in other node types with two different edge types. These queries typically start at a single point and the query complexity is sublinear to the data set size. We design 7 different queries in this complex interactive (CI) query suite, which we believe is sufficient to capture characteristics of most real-world queries. These 7 queries is labeled from "Q1" to "Q7" respectively.

- Q1: Given a node v with the type "Node1", find all nodes with the "Node4" type which are connected with v's neighbor(s) by edges of "Edge10" type, "Node4.ten" is less than 2, ordered by the values of Node4.ten.

  For instance, given a person, find the city (with less than 2 malls) where his/her friends lives in. In this example, Node1 represents persons and Node4 represents cities. Edge10 represents the friendships.

- Q2: Given a node v of type "Node1", find the neighbor(s) of v in the relationship of "Edge10" that are connected with a node of type "Node2" whose the attribute "Ten" value is less than 5 and greater than 3 and those nodes are also connected with a node of type "Node3" where "Ten = 0".

  For instance, given a person (Node1 type), find her friends who live in a city which has less than 5 malls and but more than 2 malls, and the city has a living standard rating of 0.

- Q3: Given a node v of type "Node1", find v's neighbors and v's neighbors of neighbors in "Edge10", who are connected with a node of type "Node2", ordered by the number of incoming edges from "Node3" nodes.

  For instance, given a person, find her friends who live in a city, ordered by the number of cars in the city.

- Q4: Given a node v of type "Node1", in all the nodes with "Node2" type which is connected with v, find a node of type "Node3" which is order by "Ten" and limit the returned results to 20.

  For instance, edge10 is friendship, given a person, we find the comments he/she made for those movie he/she watched. Those comments is ordered by the rank she made for the movie.

- Q5: Given a node v of type "Node1" and a node a of type "Node3" with "Ten = 1", find the neighbors of v, which are connected to a through a node of type "Node 2".

  For instance, given a person and a tag, find all the comments made by her friends which is under the tag.

- Q6: Given a node v of type "Node1", find all nodes of type "Node6" which are connected with v's neighbors in "Node1" through a node of "Node2", ordered by the number of outgoing edges to "Node2" nodes. For instance, given a person, find all the post her friends made, ordered by the number of replies made to the post.

- Q7: Given a node v of type "Node1, find the neighbors of v in "Node1" that are connected with a node of type "Node7". For instance, given a person, find all her friends who have made a comment.

CHAPTER 4. IMPLEMENTATION

In this chapter, we present the implementation of the benchmark. There are two parts to the implementation. The first part is a data generator written in Java for generating comma separated values (CSV) files of data and importing them into Neo4j and MySQL, respectively. The second part is CRUD statements in Cypher for Neo4j and in SQL for MySQL.

4.1 Graph Model with Single Node Types

4.1.1 Storing Data Graphs using RDBMS

We define six relational schemas as follows.

Node (uniqueIdx: int, uniqueNdx: int, ten: int, stringIdx: String, stringNdx: String)

RelationX (AuniqueIdx: int, BuniqueIdx: int, ten: int, stringIdx: String, stringNdx: String)

where X $\in \{1, 2, 3, 4, 5\}$.

The *Node* table is to store nodes and attributes of nodes as aforementioned. The remaining tables are to store edges for different edge types. The Relation1 table is for the edge type *Relation*1. The Relation2 table is for the edge type *Relation*2 and so on. The primary key attribute of the Node table is *uniqueIdx* and has an index associated with it. As we use the InnoDB storage engine, the only index supported is a B-tree clustered index.

For each of the Relation table, the primary key is composed of the primary key of the start node and the destination node. Each permutation of two nodes is allowed in the relation table once.

To avoid broken edges in a graph database, deleting a node without deleting its associated relations is not allowed. To enforce this rule in RDBMS, we use foreign keys in all the relation tables with the constraints *on delete cascade on update cascade* for each table.

4.1.2 Implementation of Queries in Neo4j and MySQL

4.1.2.1 BI:Bulk Insertion and BD:Bulk Deletion

We use SQL INSERT and DELETE statements in MySQL and Cypher CREATE and DELETE statements in Neo4j to implement bulk insertion and bulk deletion. Let x be the desired selectivity factor. In this case, it is the percentage of additional nodes to be inserted to the total number of existing nodes in the graph. The bulk insertion of nodes includes INSERT statements for $x\%$ additional nodes with the values of the attribute $uniqueIdx$ from N to $\lfloor N * 1.x \rfloor - 1$ and DELETE statements for these nodes.

4.1.2.2 BU:Bulk Update

- Update node attributes: We used the values of attributes $uniqueIdx$ and $uniqueNdx$ to control selectivity factors when indexing and no indexing were used, respectively. The updated attribute is *ten*, and we intentionally updated the value of this attribute to a value that is outside of the domain of this attribute.

Cypher:	MATCH (n) WHERE n.uniqueIdx < 500 SET n.ten = 11
	MATCH (n) WHERE n.uniqueNdx < 500 SET n.ten = 11
SQL:	UPDATE node SET node.ten = 11 WHERE node.uniqueIdx < 500
	UPDATE node SET node.ten = 11 WHERE node.uniqueNdx < 500

- Update edge attributes: We used the attribute *ten* of the edge type "Relation1" to control the selectivity factor and update the value to a value outside of the domain of this attribute. The following examples show the selectivity factor of 10% and 20%, respectively.

Cypher:	MATCH ()-[r:relation1]->() WHERE r.ten>8 SET r.ten=11
	MATCH ()-[r:relation1]->() WHERE r.ten>7 SET r.ten=12
SQL:	UPDATE relation1 SET relation1.ten = 11 WHERE relation1.ten > 8
	UPDATE relation1 SET relation1.ten = 12 WHERE relation1.ten > 7

4.1.2.3 BS-S:Bulk Selection with Selectivity Factor

- Selection of nodes: We used the value of the attributes $uniqueIdx$ and $uniqueNdx$ to control selectivity factors when indexing and no indexing were used, respectively. For instance, in the following statement, the selectivity factor is 500/(N - 1), where N is the total number of nodes.

Cypher:	MATCH (n) WHERE n.uniqueIdx < 500 RETURN n
	MATCH (n) WHERE n.uniqueNdx < 500 RETURN n
SQL:	SELECT * FROM node WHERE uniqueIdx < 500
	SELECT * FROM node WHERE uniqueNdx < 500

- Selection of edges: As shown in the following, when the expression "r.ten < 1" is used, the selectivity factor is 10%. When the expression "r.ten < 2" is used, the selectivity factor is 20%.

Cypher:	MATCH ()-[r:relation1]->() WHERE r.ten < 1 RETURN r
	MATCH ()-[r:relation1]->() WHERE r.ten < 2 RETURN r
SQL:	SELECT * FROM relation1 WHERE ten < 1
	SELECT * FROM relation1 WHERE ten < 2

4.1.2.4 BS-M: Bulk Selection involving Multiple Relations

- Two relation bulk selection ($BS-M:2R$): We used two different edge types as an example for the query we wrote.

Cypher:	MATCH (n)-[:relation1]->(c),(n)-[:relation2]->(d) RETURN distinct n.uniqueIdx
SQL:	SELECT x.AuniqueIdx FROM relation1 x, relation2 y WHERE x.AuniqueIdx = y.AuniqueIdx GROUP BY x.AuniqueIdx

- Find orphan nodes ($BS-M:Orphan$): As examples, we show queries involving two edge types. In our experiments, we implemented queries involving one edge type, two edge types, three edge types, four edge types, and five edge types.

Cypher	MATCH (n) WHERE NOT (()-[:relation1]->(n) OR ()-[:relation2]->(n)) RETURN n.uniqueIdx;
SQL	SELECT uniqueIdx FROM node WHERE uniqueIdx IN (SELECT uniqueIdx u1 FROM node WHERE NOT EXISTS (SELECT 1 FROM relation1 WHERE node.uniqueIdx = relation1.BuniqueIdx UNION SELECT 1 FROM relation2 WHERE node.uniqueIdx = relation2.BuniqueIdx))

4.1.2.5 SQ: Graph Structure Queries

- Find k-hop neighbors of a given node ($SQ:k-Hop$): In Cypher, we could easily change the value of "k" in the query for the edge type of interest. In the following examples, we use "k = 2."

Cypher:	MATCH (n-{uniqueIdx:0})-[:relation3∗k]->(m) RETURN DISTINCT m.uniqueIdx
SQL (k = 2):	SELECT DISTINCT b.BuniqueIdx FROM relation3 a, relation3 b WHERE a.bunique2 = b.aunique2 AND a.aunique2 = 0

- Rank of neighbors:

Cypher:	MATCH (n)-[:relation1]->(c)-[:relation2]->(b) WHERE n.uniqueIdx = 0 WITH COUNT(b) AS number, c ORDER BY number DESC RETURN c.uniqueIdx, number LIMIT k
SQL:	SELECT AuniqueIdx, COUNT(DISTINCT BuniqueIdx) AS rank FROM (SELECT r2.AuniqueIdx, r2.BuniqueIdx FROM relation2 r2 WHERE r2.AuniqueIdx IN (SELECT r1.BuniqueIdx FROM relation1 r1 WHERE r1.AuniqueIdx = 0)) AS x GROUP BY AuniqueIdx ORDER BY rank DESC LIMIT k

- Find a shortest path between a given pair of nodes ($SQ : SP$): Given a start node v_1 and the end node v_2, find the shortest path from v_1 to v_2 and returns the length of the path. Neo4j supports this type of query in Cypher. However, there is no support in MySQL. We implemented the query as a stored procedure so that the computation is done inside the database server, avoiding unnecessary communication overhead if using a JDBC or ODBC implementation. The Cypher query that finds a shortest path between the start node with $uniqueIdx = 4$ and the end node with $uniqueIdx = 70$ and returns the length of the found shortest path is shown below.

Cypher:	MATCH (a:node{a.uniqueIdx = 4}), (b:node{b.uniqueIdx = 70}) WHERE p = shortestPath((a)-[∗]- (b)) RETURN length(p)

- Compute a degree centrality of a given edge type ($SQ : Central$). We computed the degree centrality for each of the five different edge types. We take "relaiton1" as an example query.

Cypher:	MATCH p = ()-[:relation1]- >n RETURN n,count(p) as count ORDER BY count DESC LIMIT 1
SQL (k = 2):	SELECT BuniqueIdx, count(BuniqueIdx) as counted FROM relation1 GROUP BY BuniqueIdx ORDER BY counted DESC LIMIT 1

- Find all nodes with k out-going degrees of a given edge type ($SQ : degree$). We used "relation1" as an example.

Cypher:	MATCH (n)-[:relation1]->(c) WITH count(n) as number, n WHERE number = k RETURN n.uniqueIdx
SQL:	SELECT x.uniqueIdx FROM node x, relation1 r1 WHERE x.uniqueIdx = r1.BuniqueIdx GROUP BY x.uniqueIdx HAVING COUNT(DISTINCT r1.AuniqueIdx) = k

4.2 Graph Model with Multiple Node Types

4.2.1 Storing Data Graphs using RDBMS

We define 7 tables for 7 node types and 10 tables for 10 different edge types. The schemas are as follows:

Schemas for 7 different node types: NodeX (where X = 1, 2, 3, 4, 5, 6, 7)

(uniqueIdx: int, uniqueNdx: int, ten: int, stringIdx: String, stringNdx: String, corX: double, corY: double)

Schemas for 10 different table types: EdgeX (where X = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
(AuniqueIdx: int, BuniqueIdx: int, ten: int, stringIdx: String, stringNdx: String)

Each *NodeX* table stores nodes and attributes of nodes of its node type as aforementioned. The remaining tables are to store edges for different edge types. The primary key of each *NodeX* table is *uniqueIdx* and has an index associated with it. For each *EdgeX* table, the primary key is composed of the primary key of the start node and the destination node. Each permutation of two nodes is allowed in the Edge table once.

To avoid broken edges in the graph database, deleting a node without deleting its associated relationships is not allowed. To enforce this rule in RDBMS, we use foreign keys in all the Edge tables with the constraints *on delete cascade on update cascade*, for each table. Since the storage of nodes is unsorted, we use a clustered index on the attribute *uniqueIdx*.

4.2.2 Implementation of Queries in Neo4j and MySQL

- Q1: Given a node n of type "Node1", find all the nodes of type "Node4" which are connected with n's neighbor(s) via "Edge10" type and the values of $Node4.ten$ are less than 2 and ordered by the values of $Node4.ten$.

Cypher:	MATCH (a:node1 {UniqueIdx : 1}) -[:edge10]-> (n:node1), (n)-[:edge9]->(b:node4) WHERE b.ten < 2 RETURN n ORDER BY b.ten
SQL :	SELECT e3.node4ID, n.uniqueIdx, n.ten FROM edge10 e1 JOIN edge10 e2 ON e1.BuniqueIdx = e2.AuniqueIdx JOIN edge9 e3 ON e2.BuniqueIdx = e3.AuniqueIdx JOIN node4 n ON e3.BuniqueIdx = n.uniqueIdx WHERE e1.AuniqueIdx = 1 ORDER BY n.ten;

- Q2: Given a node n of type "Node1", find the neighbors of n in "Edge10" type that is connected with a node of type "Node2" whose the value of the attribute *ten* is between 3 and 5 exclusive and those nodes are also connected to a node of type "Node3".

Cypher:	MATCH (n2)-[r:edge8]->(n1:node1), (n:node1{UniqueIdx:1})–>(n1), (n3:node3)–>(n2: node2) WHERE n3.ten = 0 and n2.ten <5 and n2.ten > 3 with n1, COUNT(r) AS number ORDER BY number RETURN n1;
SQL:	SELECT COUNT(e4.BuinqueIdx), e4.BuniqueIdx FROM edge8 e1 JOIN node2 n2 ON n2.uniqueIdx = e1.AunqiueIdx JOIN node1 n1 ON e1.BuniqueIdx = n1.uniqueIdx JOIN edge4 e3 ON e3.BuniqueIdx = n2.uniqueIdx JOIN node6 n3 ON e3.AuniqueIdx = n3.uniqueIdx JOIN edge10 e4 ON n1.uniqueIdx = e4.AuniqueIdx WHERE n1.uniqueIdx = 1 AND n2. ten < 5 and n2.ten > 3 and n3.ten = 0 GROUP BY e4.BuniqueIdx ORDER BY COUNT(e4.BuniqueIdx);

- Q3: Given a node n of type "Node1", find n' s neighbors and n's neighbors of neighbors in "Edge10" relationship, who are connected with a node of type "Node2", ordered by the number of incoming edges from node type "Node3".

Cypher:	MATCH (post:node3)-[r1:edge7]->(forum:node2) -[r2:edge8]->(friend:node1)<-[:edge10∗0..2]-(person:node1{uniqueIdx:1}) WITH forum, COUNT(r1) AS number ORDER BY number DESC RETURN forum.uniqueIdx, number
SQL:	SELECT e7.BuniqueIdx, COUNT(e7.AuniqueIdx) AS number FROM edge8 e8 JOIN edge7 e7 ON e8.AuniqueIdx = e7.BuniqueIdx JOIN (SELECT DISTINCT ex.AuniqueIdx FROM (SELECT DISTINCT e2.BuniqueIdx FROM edge10 e1 INNER JOIN edge10 e2 ON e1.BuniqueIdx = e2.AuniqueIdx WHERE e1.AuniqueIdx = 1 UNION SELECT DISTINCT e2.BuniqueIdx FROM edge10 e1 INNER JOIN edge10 e3 ON e1.BuniqueIdx = e3.AuniqueIdx JOIN edge10 e2 ON e3.BuniqueIdx = e2.AuniqueIdx WHERE e1.AuniqueIdx = 1) newtable JOIN edge8 ex ON newtable.BuniqueIdx = ex.BuniqueIdx) newnewtable ON e7.BuniqueIdx = newnewtable.AunqiueIdx GROUP BY e7.BuniqueIdx ORDER BY number DESC;

- Q4: Given a node n of node type “Node1”, in all the nodes of “Node2” type which are connected with n, find a node with “Node3” which is order by the value of Ten.

Cypher:	MATCH (n3:node3)-[:edge7]->(n2:node2)-[r:edge8]->(n1:node1{UniqueIdx:1}) RETURN n3.uniqueIdx ORDER BY n3.ten
SQL:	SELECT e7.AuniqueIdx FROM node3 n3 JOIN edge7 e7 ON n3.uniqueIdx = e7.AuniqueIdx JOIN node2 n2 ON e7.BuniqueIdx = n2.uniqueIdx JOIN edge8 e8 ON n2.uniqueIdx = e8.AuniqueIdx WHERE e8.BuniqueIdx = 1 GROUP BY e7.AuniqueIdx ORDER BY n3.ten;

- Q5: Given a node V of node type of "Node1" and a node A of node type of "Node3" with "Ten = 1", find the neighbors of V in the relationship of "Edge10", which are connected to A through a node of "Node 2".

Cypher:	MATCH (n3:node3)–>(n2:node2)-[r:edge8]->(n1:node1)<–(n:node1{uniqueIdx:1})WITH n1, COUNT(r) AS number ORDER BY number DESC RETURN n1 LIMIT 20
SQL:	SELECT r2.AuniqueIdx, r2.BuniqueIdx, r2.c FROM (select e8.AuniqueIdx, e.BuniqueIdx, COUNT(e8.AuniqueIdx) AS c FROM (SELECT f2.BuniqueIdx FROM edge10 f1 INNER JOIN edge10 f2 ON f1.BuniqueIdx = f2.AuniqueIdx WHERE f1.AuniqueIdx = 1) e JOIN edge8 e8 ON e8.BuniqueIdx = e.BuniqueIdx GROUP BY BuniqueIdx ORDER BY COUNT(e8.AuniqueIdx))r2 JOIN edge7 e7 ON r2.BuniqueIdx = e7.BuniqueIdx JOIN node3 n3 ON e7.AuniqueIdx = n3.uniqueIdx AND n3.uniqueIdx = 1;

- Q6: Given a node n of type "Node1", find all nodes of type "Node6" which are connected with n's neighbor(s) of type "Node2" through a node of type "Node2", ordered by the number of outgoing edges to the nodes of type "Node2".

Cypher:	MATCH (n3:node3)-[:edge7]->(n2:node2)-[r:edge8]->(n1:node1 {uniqueIdx:1}) RETURN n3 ORDER BY n3.ten
SQL:	SELECT e7.AuniqueIdx FROM node3 n3 JOIN edge7 e7 ON n3.uniqueIdx = e7.AuniqueIdx JOIN node2 n2 ON e7.BuniqueIdx = n2.uniqueIdx JOIN edge8 e8 ON n2.uniqueIdx = e8.AuniqueIdx WHERE e8.BuniqueIdx = 1 GROUP BY e7.AuniqueIdx ORDER BY n3.ten;

- Q7: Given a node n of type "Node1, find the neighbors of n of type "Node1" that are connected with a node of type "Node7".

Cypher:	MATCH (forum:node7)-[r2:edge6]->(friend:node1) <-[:edge10]-(person:node1{uniqueIdx:1}) RETURN forum
SQL:	SELECT e6.AuniqueIdx FROM edge6 e6 JOIN (SELECT DISTINCT e2.BuniqueIdx FROM edge10 e1 INNER JOIN edge10 e2 ON e1.BuniqueIdx = e2.AuniqueIdx WHERE e1.AuniqueIdx =1 UNION SELECT DISTINCT e.BuniqueIdx FROM edge10 e WHERE e.AuniqueIdx = 1)f ON e6.BuniqueIdx = f.BuniqueIdx;

CHAPTER 5. EXPERIMENTS AND RESULTS

The experiments were performed on a PC with Intel Core i7-4500U CPU at 1.80GHz and with 8 Gbytes of memory. MySQL Server version 5.6.22 and Neo4j Community Edition version 2.2.5 were used. We used the default cache model in Neo4j which is the file buffer cache. Even though the object cache is considered to offer higher performance, it is not available in the Neo4j Community Edition. We used the default cache sizes for both databases:1 GBytes in Neo4j and 1 MBytes in MySQL. The InnoDB storage engine was used in MySQL.

In the entire process of running the benchmark, to avoid interference from other programs, we closed all other user applications except for the MySQL sever and the command line client when executing the queries in RDBMs and Neo4jShell when executing queries in GDBMs. The running time was collected as a performance metric. In each query, we noticed that the execution times were different for the first few runs. We believe it is due to a cache warmup. To get a reliable result, we executed each query more than 30 times and discarded the first 10 times. We computed the execution time as the average of the last 20 runs.

5.1 Data Generation and Execution Results for Random Graph

5.1.1 Data Generation

In our experiment, we generated two different sparse data sets, one with 5000 nodes, and the other with 10000 nodes. Table 5.1 shows the number of nodes and edges, respectively. We also generated a set of new edges which does not exist in the database, which were used in S2 for the bulk insertion operation.

The information about the attributes of the table “Node” with 5000 nodes is shown in Table 5.2. For the data set with 10000 nodes, all the attributes are the same except that the

Table 5.1 Sizes of the two data sets

Data Set No.	Number of Nodes	Number of Edges in Each Relation
1	5000	25000
2	1000	100000

Table 5.2 Attribute Specifications of Table Node

Attribute Name	Range of Values	Order	Comment
uniqueNdx	0∼ N - 1	Unique	Candidate key, No Index
uniqueIdx	0∼ N - 1	Unique	Primary key, Index
Ten	0∼9	Random	Ramdom%10, No Index
StringIdx	"aaaa" ∼ "jjjj"	Random	–
StringNdx	"aaaa" ∼ "jjjj"	Random	–

values of $uniqueIdx$ and $uniqueNdx$ ranging from 0 to 9999. Since the values of the attribute $StringINdx$ are randomly chosen from ten different strings, the occurrence of each string value is 10% of the total number of rows in the table. Let N be the number of nodes in the graph.

5.1.2 BI:Bulk Insertion and BD:Bulk Deletion

As shown in Figure 5.1,5.2,5.3,5.4, the execution time of inserting/deleting of nodes/edges in Neo4j is more than 10 times of that of MySQL. At the same selectivity factor, the execution times of insertion of nodes in Neo4j increases with a larger database while those in MySQL are almost the same. However, the execution times of deleting nodes increase for the larger database at the same selectivity factor.

For inserting/deleting edges, what is interesting is that the increase amount of the execution times for the database with 5000 nodes to the database with 10000 nodes at a certain selectivity factor is different in Neo4j and MySQL. Typically, the increase amount of times taken by Neo4j is not as large as that of MySQL for the same selectivity factor. It might indicate that Neo4j would be more efficient when inserting/deleting edges for a larger graph.

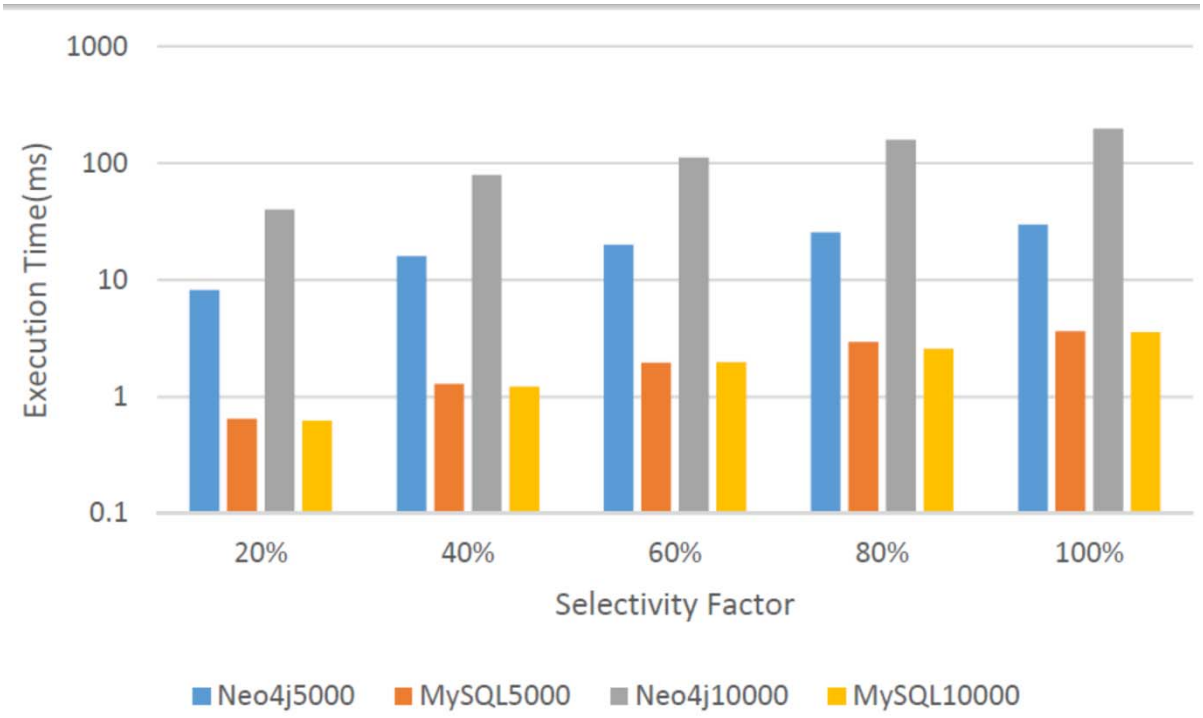

Figure 5.1 Execution time of insertion of nodes with different selectivity factors

5.1.3 BU:Bulk Update

In this workload, we discussed the result on the 5000 node data set as an example. In Figure 5.5, we showed the execution time for this data set with and without index. We updated the attributes of the nodes with different selectivity factors. The difference between the execution time with index and without index is tiny. The execution time taken by Neo4j is greatly larger than that in MySQL. Figure 5.6 shows the execution time of updating edges for the four different graph databases, two in Neo4j and the others in MySQL. It is noticed that in MySQL, the execution time for updating edges in a small number of data set does not change significantly (MySQL5000). It increases with the increase in the selectivity factors (MySQL10000).

5.1.4 BS-S:Bulk Selection with Selectivity Factor

In this workload, we discussed the result on the 5000 node data set as an example in Figure 5.7 and Figure 5.8. The selective factor was controlled by select different values in properties

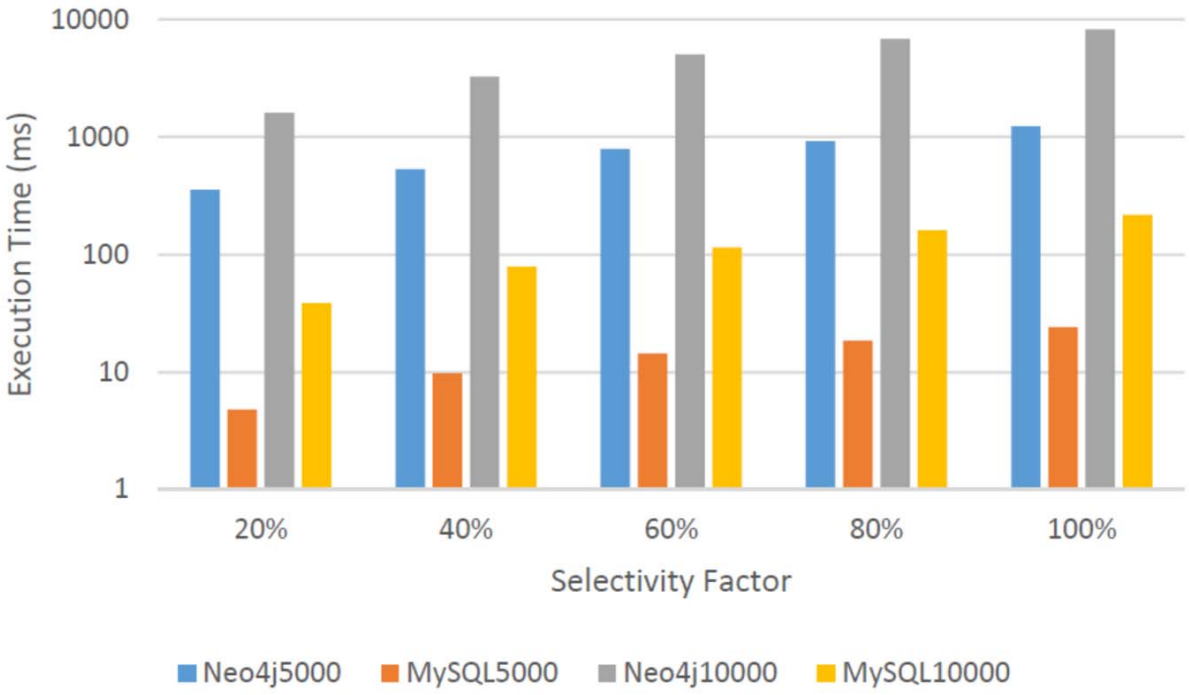

Figure 5.2 Execution time of insertion of edges with different selectivity factors

of node. Figure 5.7 applied *uniqueIdx* and *uniqueNdx*. Figure 5.8 applied *stringIdx* and *stringNdx*. Both of Figure 5.7 and Figure 5.8 reveal that Neo4j takes twice the time without index than with index. In MySQL, queries with index and without index give almost the same execution times. We also found that no matter which data type we chose, the results were almost the same. Therefore, there is no significant difference between the selection with Integer type and the selection with String type.

5.1.5 BS-M: Bulk Selection involving Multiple Relations

- Two relation bulk selection ($BS-M:2R$). As showed in Table 5.3, Neo4j almost cost 10 times the execution time of MySQL.

- Find orphan nodes ($BS-M:Orphan$).

  Figure 5.10 shows that the execution time to find orphan nodes in MySQL is much longer than that in Neo4j.

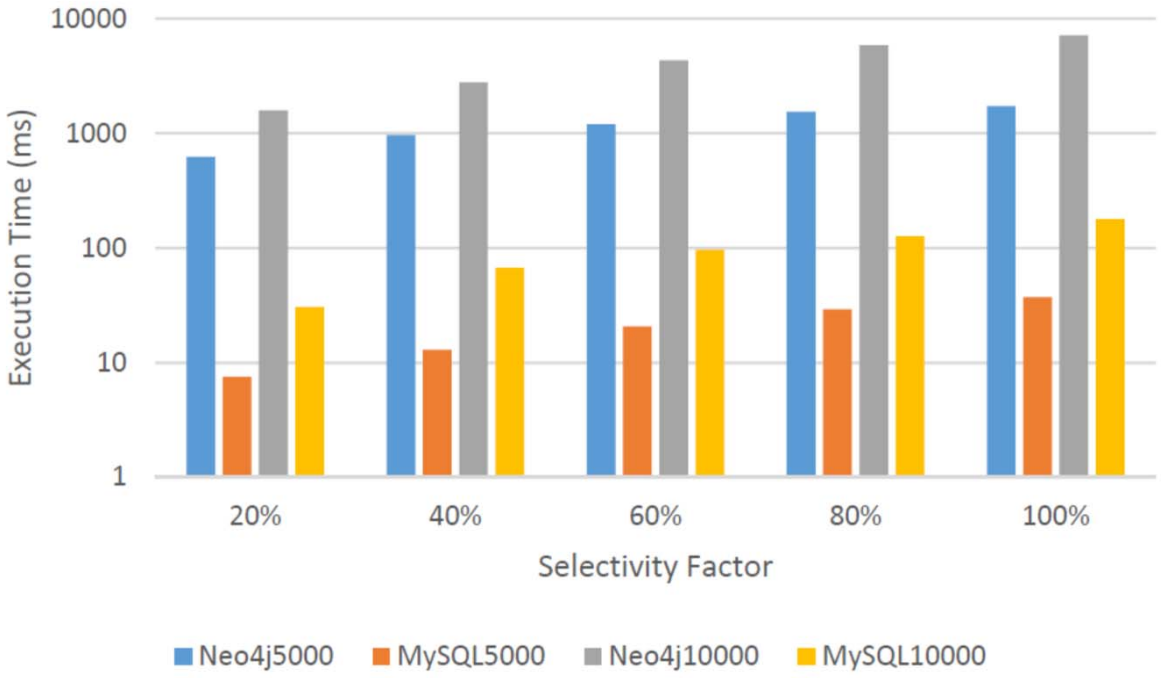

Figure 5.3 Execution time of deleting nodes with different selectivity factors

Table 5.3 Execution time (ms) of two relation bulk selection

# Nodes / Database	5000	10000
Neo4j	2052	8668
MySQL	318	1248

5.1.6 SQ: Graph Structure Queries

- Find k-hop neighbors of a given node ($SQ : kHop$). Figure 5.11 shows that the execution time increases greatly with the increasing value of k. Even the execution time in Neo4j is greatly larger than that in MySQL, the rate of increasing is smaller than that in MySQL.

- Rank of neighbors ($SQ : Ranking$). Table 5.4 shows that the execution time in Neo4j is larger than that in MySQL.

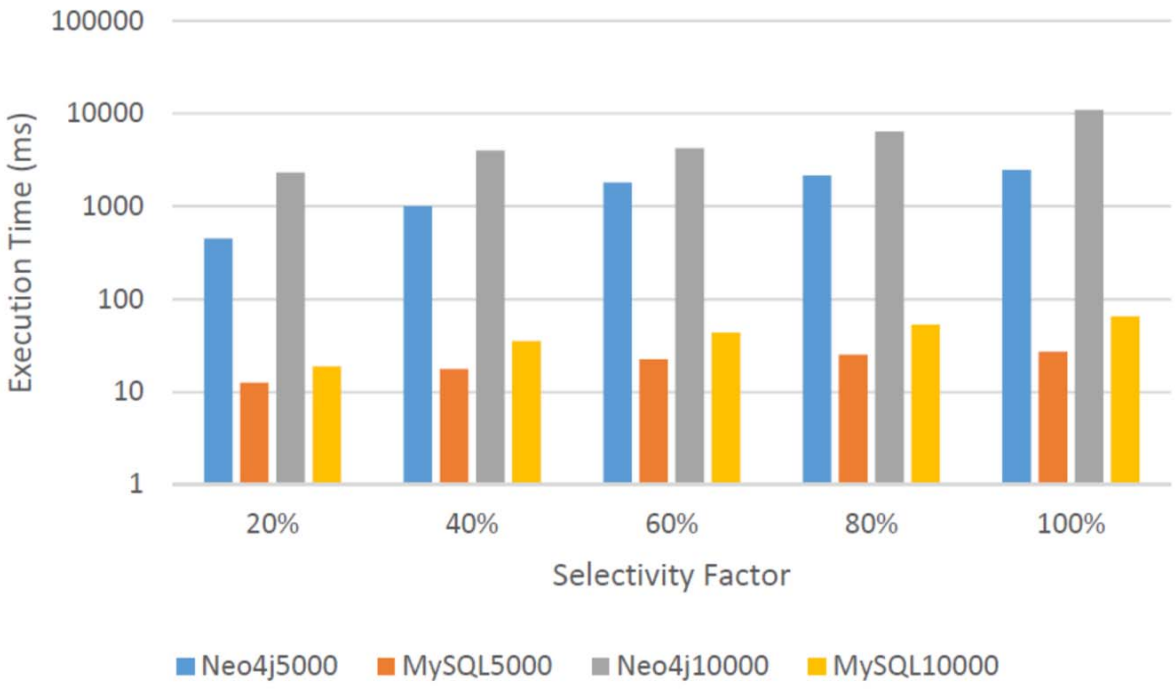

Figure 5.4 Execution time of deleting edges with different selectivity factors

Table 5.4 Execution time (ms) of ranking of neighbors

DBMS \ # Nodes	5000	10000
Neo4j	25	56
MySQL	1.5	1.8

- Find a shortest path among a given pair of nodes. For this query, we provided two nodes (the starting node and the ending node). We made sure that the same two nodes in Neo4j and MySQL were used. A shortest path was computed by a stored procedure in MySQL and is the counterpart of "shortestpath" function in Neo4j. Especially, in this query we noticed that the cold and hot setup have a critical influence on the performance. See Figure 5.12. Tests were performed with cold, which we denote it "first time" and hot setup to determine the time needed for a database to load data into memory.

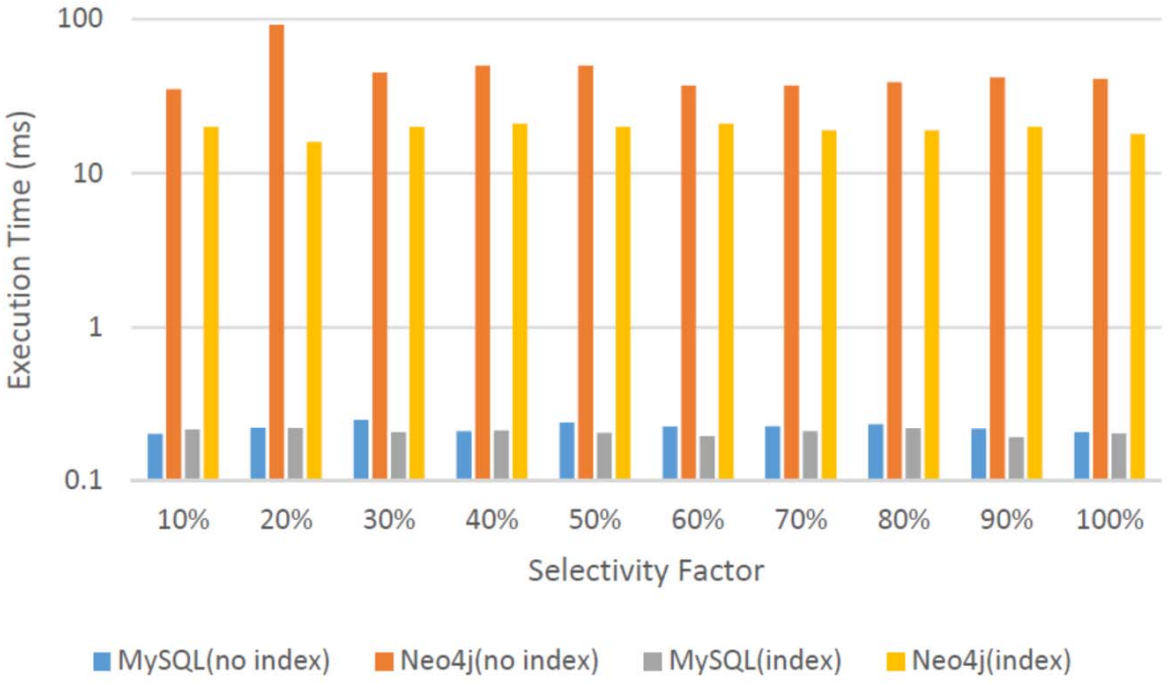

Figure 5.5 Execution time of updating nodes with different selectivity factors

The measured cold time is the time required to finish a given set of queries immediately after flushing caches or after reboot. In reality this is achieved by rebooting the system. The hot test run consists of executing the same set of queries as from the cold run, in the same sequence order. The measured hot time is the time required to finish a given set of queries immediately after performing the cold run without flushing any cache or restarting the database. All of the cold tests were executed with cleared disk cache at operating system level. Figure 5.12 shows that Neo4j is slower than MySQL for the first time. But after cache warmup, Neo4j is faster than MySQL.

- Compute a degree centrality of a given edge type. We executed the queries for five different edge types. Figure 5.13 shows that Neo4j takes about twice as much time as MySQL. The execution times increase with the increase in the database size.

- Find all nodes with k out-going degree of a given edge type. Figure 5.14 shows the result of "relation1".

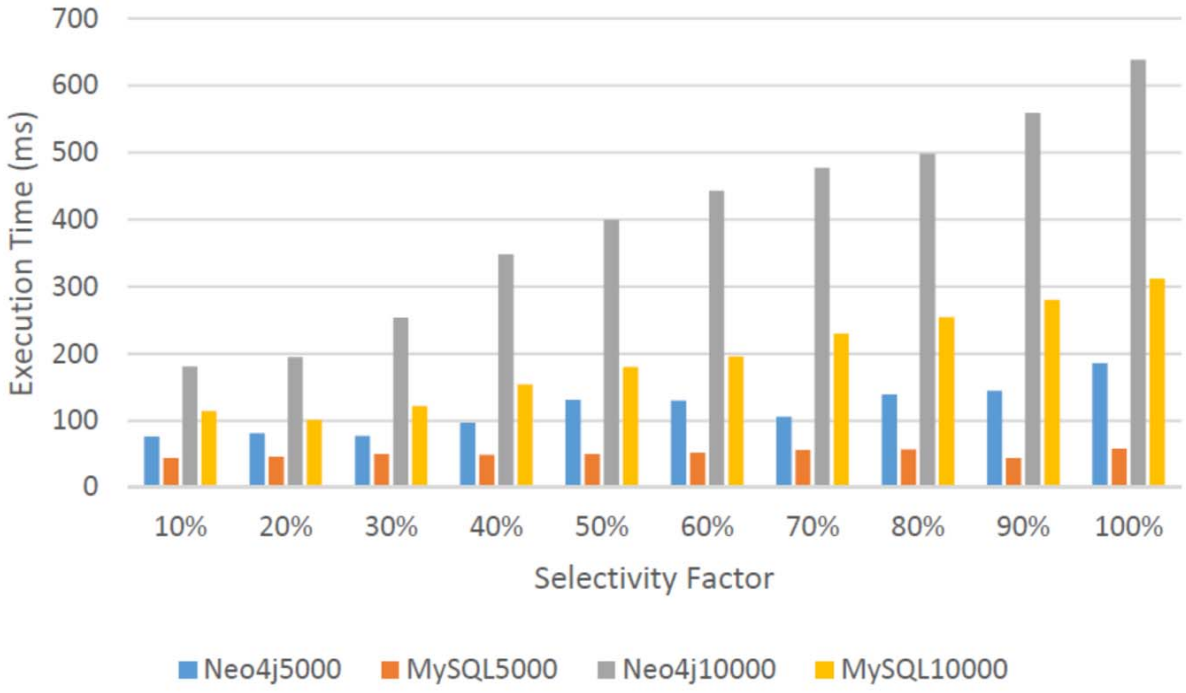

Figure 5.6 Execution time of updating edges with different selectivity factors

5.1.7 Results with 1 Million Nodes

Besides comparing two data set (5000 nodes and 10000 nodes), we also explore a larger data set of a random graph with 1,000,000 nodes. We explored whether the volume of a random graph would have effect on the execution time when executing structure related queris. Therefore, we showed the results from SQ (Graph Structure Queries).

- Find k-hop neighbors of a given node. Figure 5.15 shows that the execution time in Neo4j is larger than that in MySQL. However, as the value of k increases, the difference between the execution time decreases.

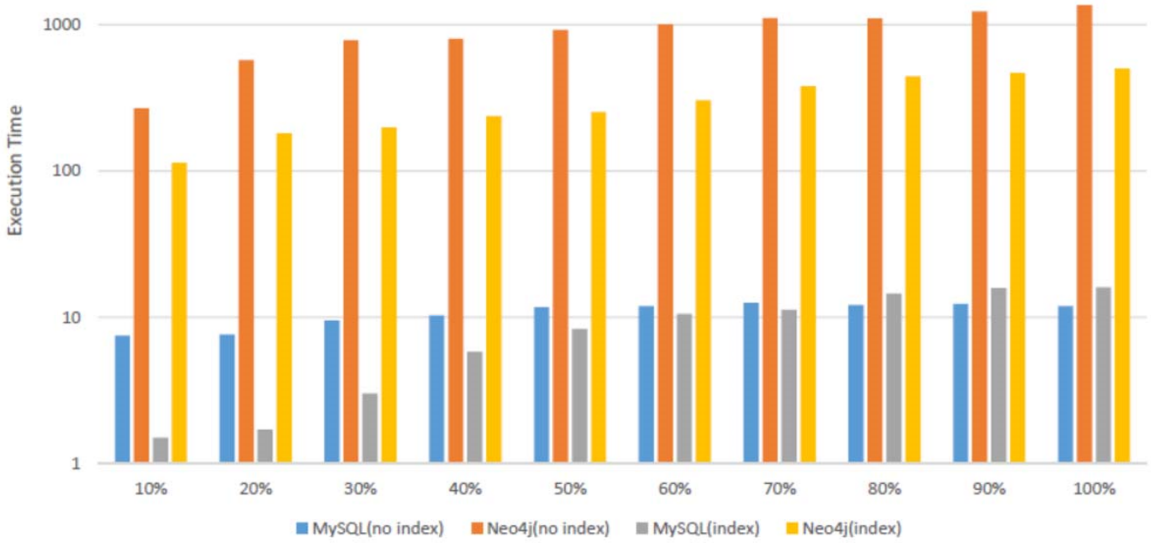

Figure 5.7 Execution time of selecting nodes with different selectivity factors on Integer with/without index.

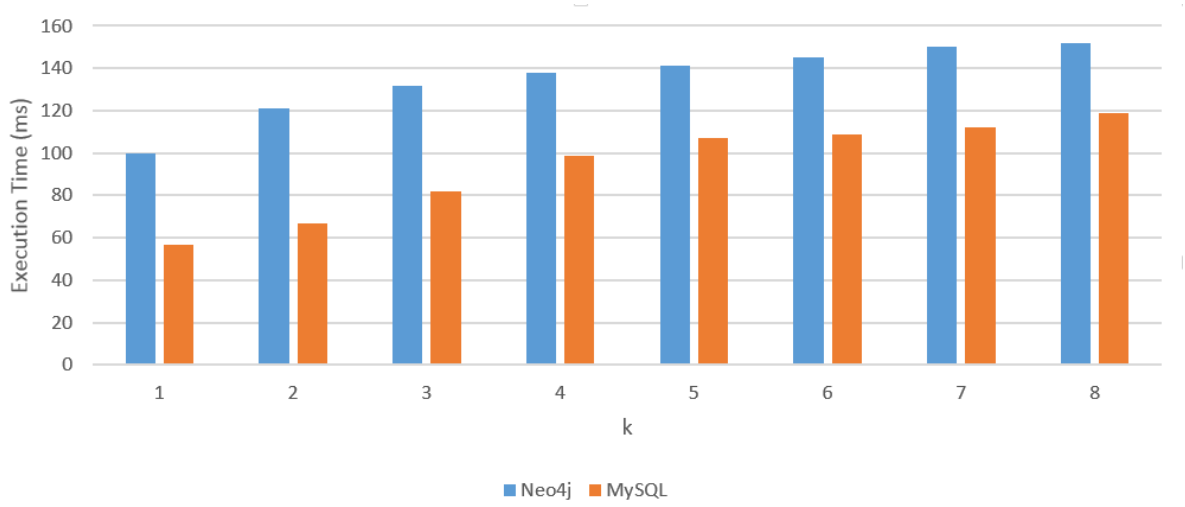

Figure 5.15 Execution time of finding k-hop neighbors of a given node

- Rank of neighbors. Table 5.5 shows the execution results of Ranking of Neighbors query. Again, the execution time of Neo4j is much greater than that of MySQL.

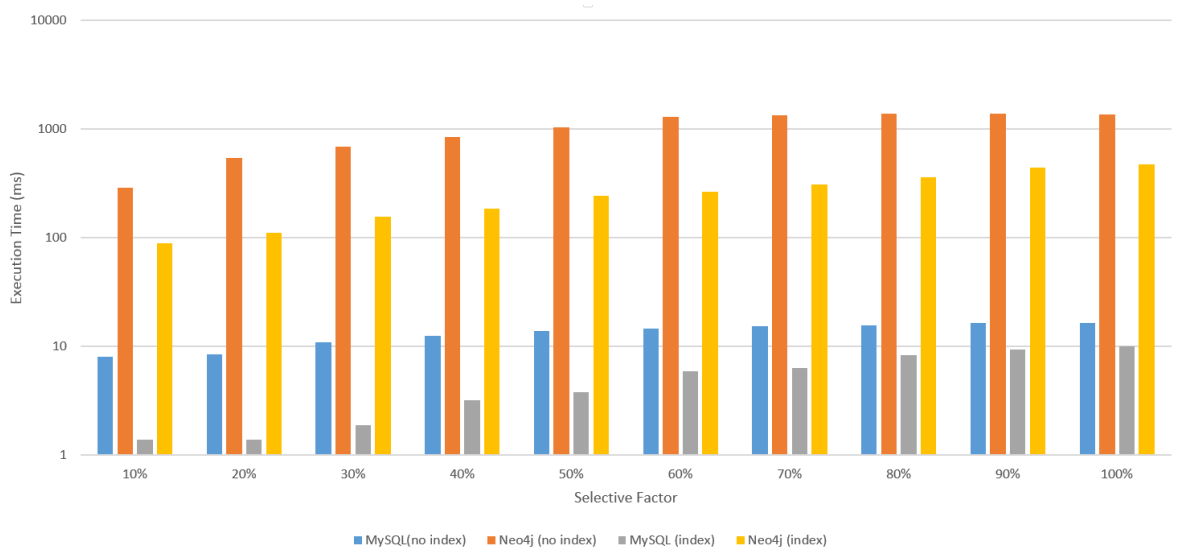

Figure 5.8 Execution time of selecting nodes with different selectivity factors on String with/without index.

Table 5.5 Execution Time of Ranking of Neighbors in random graph of 1M nodes

DBMS	Neo4j	MySQL
Execution Time (ms)	98	5.6

- Find a shortest path among a given pair of nodes 5.16. Similarly in the execution time in Random graph, the cache-warmup effect is obvious in Neo4j. However, the execution time of MySQL is greater than the first time execution time in Neo4j, which is different from the results in Chapter 5.1.6.

- Compute a degree centrality of a given edge type. Figure 5.17 shows that Neo4j takes much more time than MySQL.

- Find all nodes with k out-going degrees of a given edge type. Figure 5.18 shows that the execution times in Neo4j is about 20 times of those in MySQL.

5.2 Data generation and Execution Results for Kronecker Graph

We generated a Kronecker graph using the APGL in Python (42). We set the initial graph with 5 nodes and the number of iteration to 5. A Kronecker graph with 50,000 nodes and

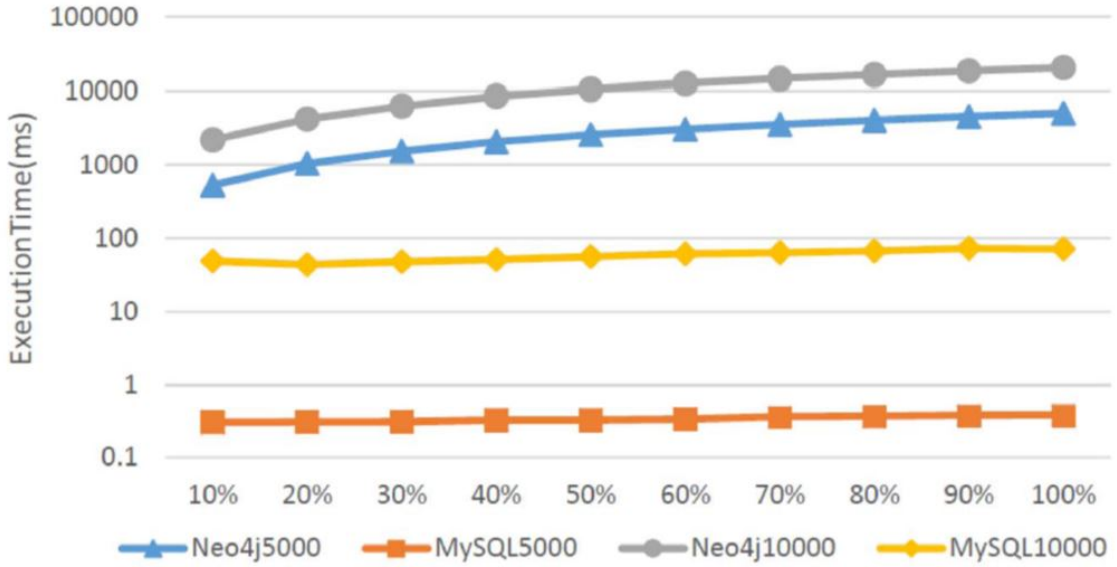

Figure 5.9 Execution time of selecting edges with different selectivity factors

Table 5.6 Execution Time of Ranking of Neighbors in a Kronecker Graph

DBMS	Neo4j	MySQL
Execution Time (ms)	10	3.48

322850 edges was generated. We then used the data graph generator to generate attributes for nodes and edges. We only showed the result of the categories related to graph structure in the Graph Structure Query workload as the major difference between Kronecker graphs and Random graphs are their structure.

- Find k-hop neighbors of a given node. Figure 5.19 shows that the execution time in Neo4j is larger than that in MySQL. We also noticed that there is no definite pattern of time with the increasing value of k.

- Rank of neighbors. Table 5.6 shows the execution results of ranking of neighbors query in a Kronecker graph.

- Find a shortest path among a given pair of nodes 5.20. Similarly in the execution time in Random graph, the cache-warmup effect is obvious in Neo4j.

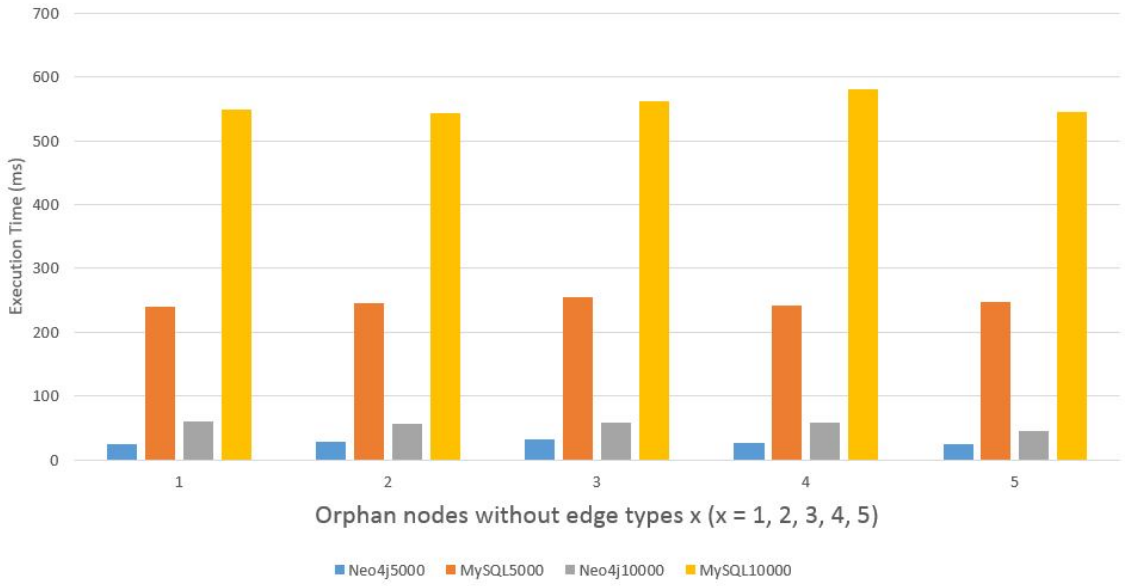

Figure 5.10 Execution time of finding orphan nodes in five different edge types.

- Compute a degree centrality of a given edge type. Figure 5.21 shows that Neo4j takes much more time than MySQL.
- Find all nodes with k out-going degrees of a given edge type. Figure 5.22 shows that the execution times in Neo4j is about 10 times of those in MySQL.

5.3 Data Generation and Execution Results for Graph Model with Multiple Node Types

5.3.1 Data Generation

For the interactive query suite in a complex graph model with multiple node types, we generated there data sets of 10,000, 100,000, 1,000,000 nodes. Table 5.7 shows the summary about these data sets. We applied the default parameters when generating these data. That is the Zipf factor for the distribution of the number nodes for each node type is 0.1. For each edge type, we used the default Zipf factor of 0.8 when generating a specific edge type. The correlation between the values of the attributes 'CorX' and 'CorY' was 0.9. Therefore, we

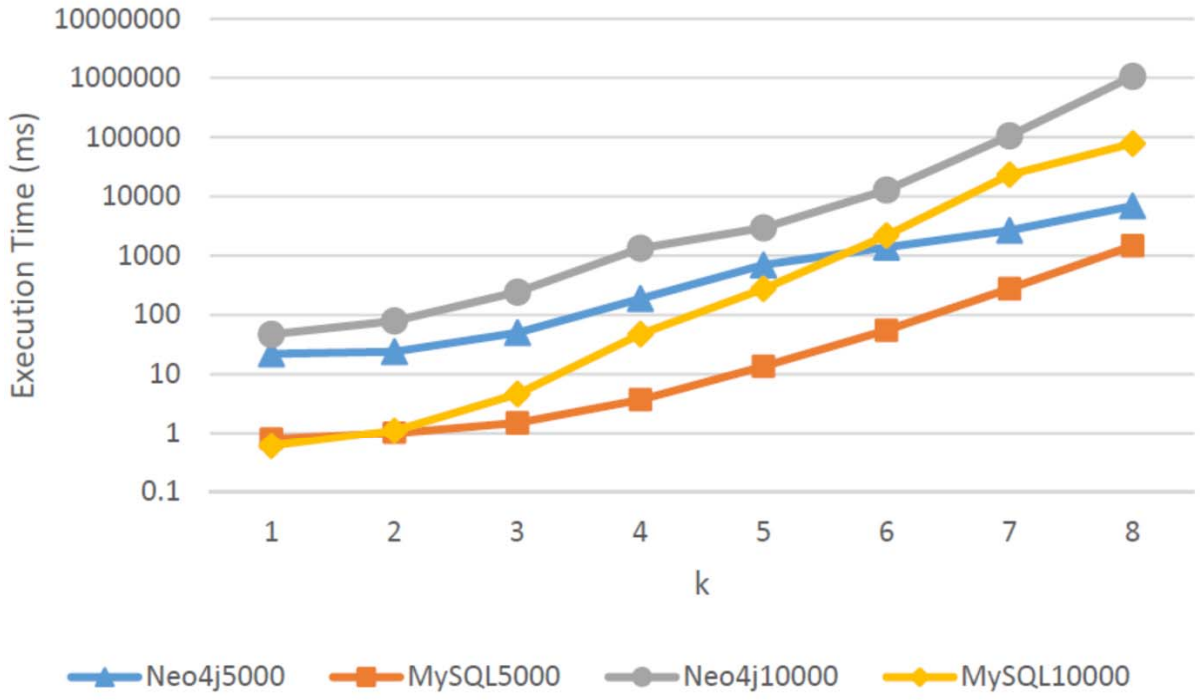

Figure 5.11 Execution time of finding k-hop neighbors of a given node

had a total of 6 databases: three in Neo4j and three in MySQL. We named them as "Neo4j-10K", "Neo4j-100K", "Neo4j-1000K", "MySQL-10K", "MySQL-100K" and "MySQL-1000K", respectively.

Table 5.7 Six database for multiple node type graph model used in benchmark

Database Name	Number of Nodes	Number of Edges	Configuration
Neo4j-10K	10,000	28,900	default
Neo4j-100K	100,000	289,000	default
Neo4j-1000K	1,000,000	2,890,000	default
MySQL-10K	10,000	28,900	default
MySQL-100K	100,000	289,000	default
MySQL-1000K	1,000,000	2,890,000	default

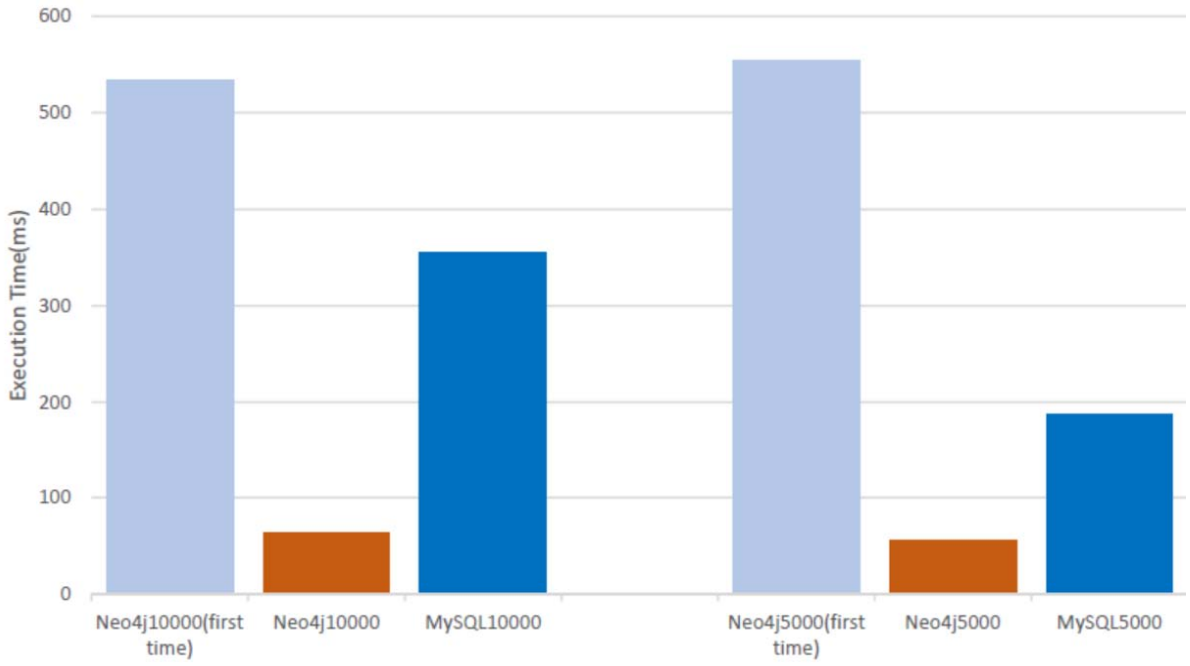

Figure 5.12 Execution time of the shortest path query

5.3.2 Execution Results for Graph Model with Multiple Node Types

To execute these queries, we provided a starting node first. There were several ways to provided the starting nodes. We executed these queries in two different ways according the way we selected the starting node.

As shown in Figure 5.23, we defined that the start node of type "Node1" was the one with ID = 1 in all the queries. We executed the same query continuously for 30 times and computed the average execution time. As shown in Figure 5.23, the performance of Neo4j is better than that of MySQL for the same data, except Q3. Q3 requires a "union" operation between n's friends and n's friends of friends, which is an expensive operation in MySQL.

The other execution method was choosing a set of 50 different start nodes of type "Node1", for instance, nodes of "Node1" with ID = 1, ID = 2,..., ID = 50. We executed queries with the set of nodes in a row and measured the total execution time. In this way, we could reduce the effect of the structure of the start node. As shown in Figure 5.24, Q3 still costs a longer time

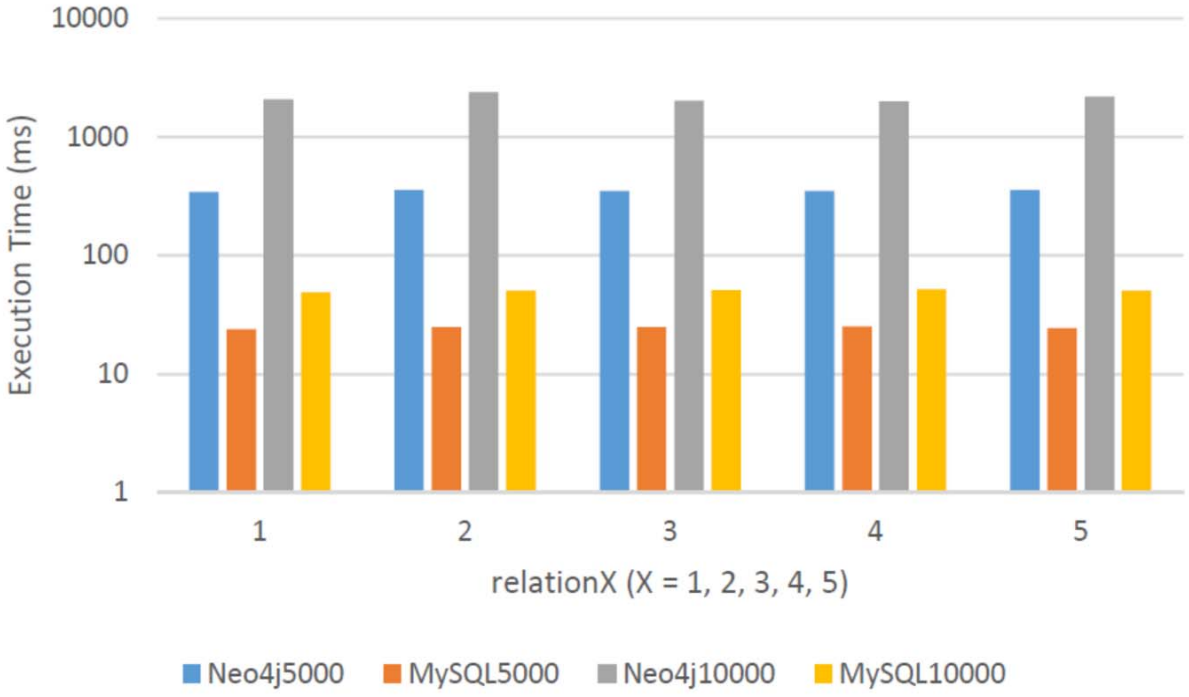

Figure 5.13 Execution time for computing a degree centrality

in MySQL than in Neo4j. And the difference between the execution time in Neo4j and that in MySQL increases a lot.

5.3.3 Selection with Two Linear Correlated Fields

We generated two correlated sequences which was stored as double-format in two properties of "Node1": 'CorX' and 'CorY', respectively. The values of these two properties was linear correlated with a correlation parameter of 0.9, as we mentioned in 5.3.1. Mean value of both 'CorX' and 'CorY' is 0. Standard deviation of 'CorX' is 1, while standard deviation of 'CorY' is 2.29 (45). To show the effect of two correlated properties on the execution time of selecting nodes, we selected nodes of "Node1" type by setting a certain range on both properties. As showed in Figure 5.25, the selection range on X-Axis is that we select Node1 with 'CorX' and 'CorY' less than the certain value. For example, "-1" means selecting all the nodes whose 'CorX' and 'CorY' is less than -1, while "-0.8" means selecting all the nodes whose 'CorX' and 'CorY' is less than -0.8, and so on.

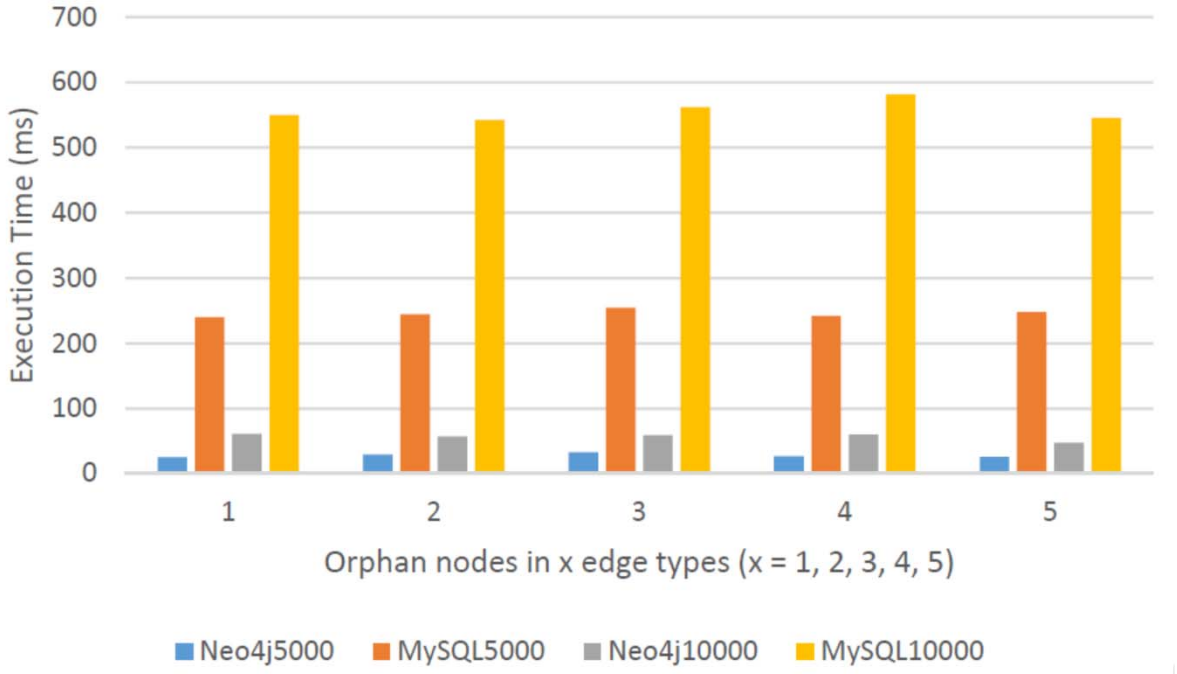

Figure 5.14 Execution time of finding all nodes with k out-going degree in edge type of "relation1"

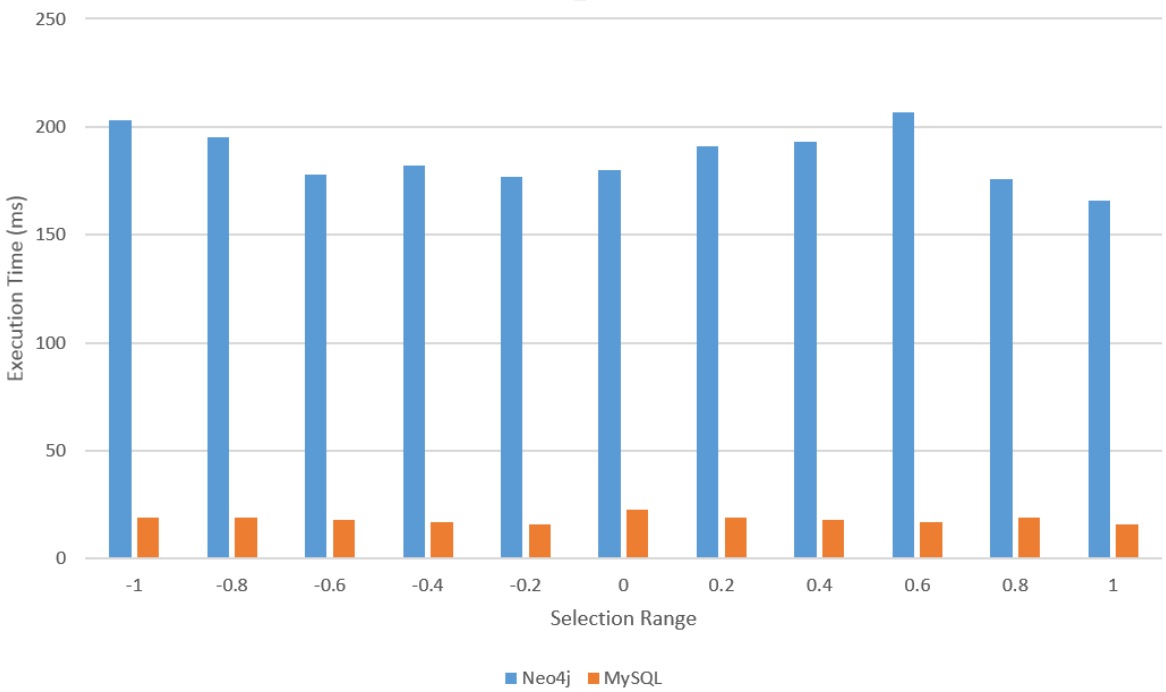

Figure 5.25 Execution time for selecting two correlated properties

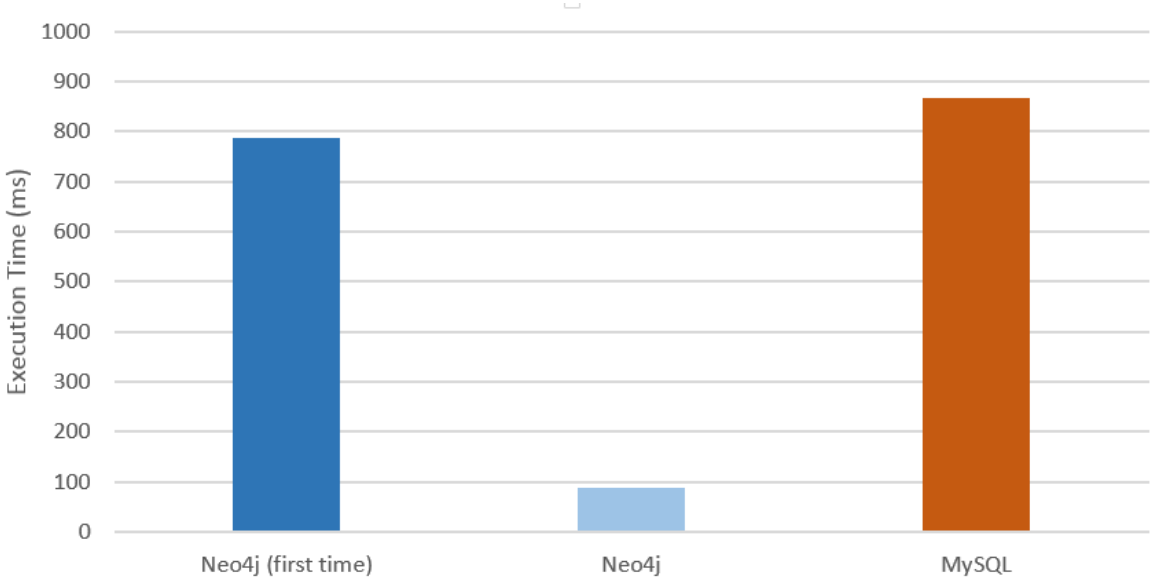

Figure 5.16 Execution time of shortest path in a random graph with 1M nodes

5.4 Cache Size and Performance

All the results showed above was under the default cache configuration in both MySQL and Neo4j. However, tuning the cache size in database could effect the performance of database management significantly. Therefore, to make database work as efficiently as possible, we also executed the queries we designed under different cache configurations.

However, the optimal cache size depends a lot on the database volume, database query types, database traffic and hardware. We have to define the data set we use and the queries at the beginning of our experiment. In the experiment, we showed the performance tuned by cache configuration on two data sets. One is the data set with 1M nodes of random graph we generated in Chapter 5.1.7. For this data set, we investigate the performance tuned by cache size for the query of shortest path. The reason we chose this query is not only because that it is the only query that Neo4j performed faster than MySQL did among all the queries we had studied in our benchmark, but also because that shortest path query itself is of a significant query for graph data. The other data set is the one with 1M nodes of multiple node types generated in Chapter 5.3.1. We chose Q3 as the query when tuning cache size because it was

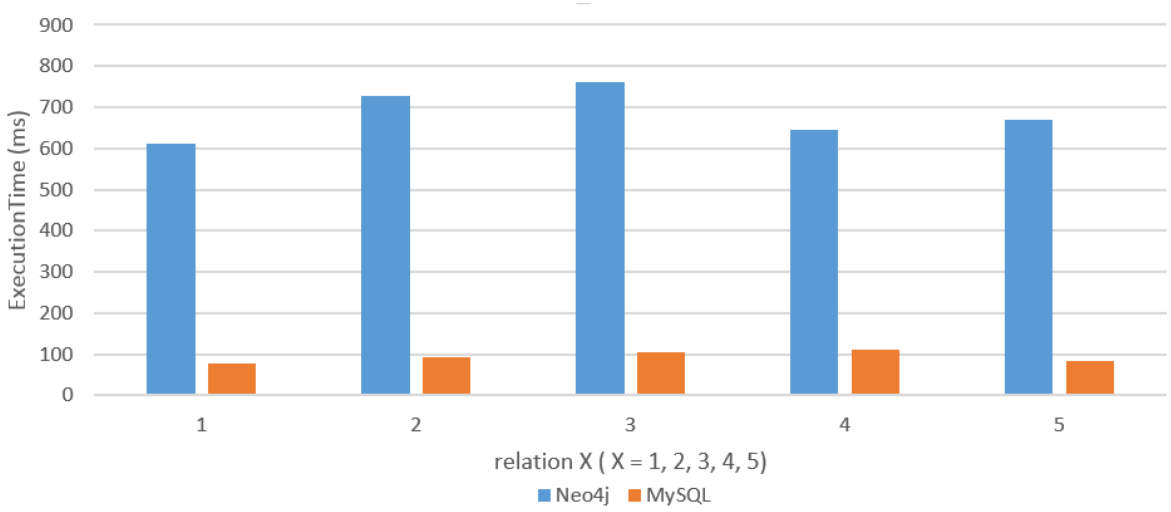

Figure 5.17 Execution time of Compute a degree centrality of five different edge types in a random graph with 1M nodes.

the only query of MySQL was slower than Neo4j. The purpose of the tuning cache size is to find an optimal cache size for these two queries in MySQL, and compare the performance of MySQL and Neo4j under their optimal cache configuration.

In MySQL, there are three parameters of query cache to tune the performance (3): query_cache_limit, query_cache_min_res_unit and query_cache_size. The query_cache_limit could control the maximum size of individual query results that can be cached. The default value is 1MB. The query_cache_min_res_unit could tune parameter to combat cache fragmentation and reduce prunes if that is an issue. The default value is 4K.

A typical query cache configuration is showed as following:

query_cache_type = 1

query_cache_limit = 1M

query_cache_min_res_unit = 4K

query_cache_size = 1M

In our experiments, we kept both of these values to be default. The only parameter we changed was the value of query_cache_size. As shown in Figure 5.26 and Figure 5.27, the

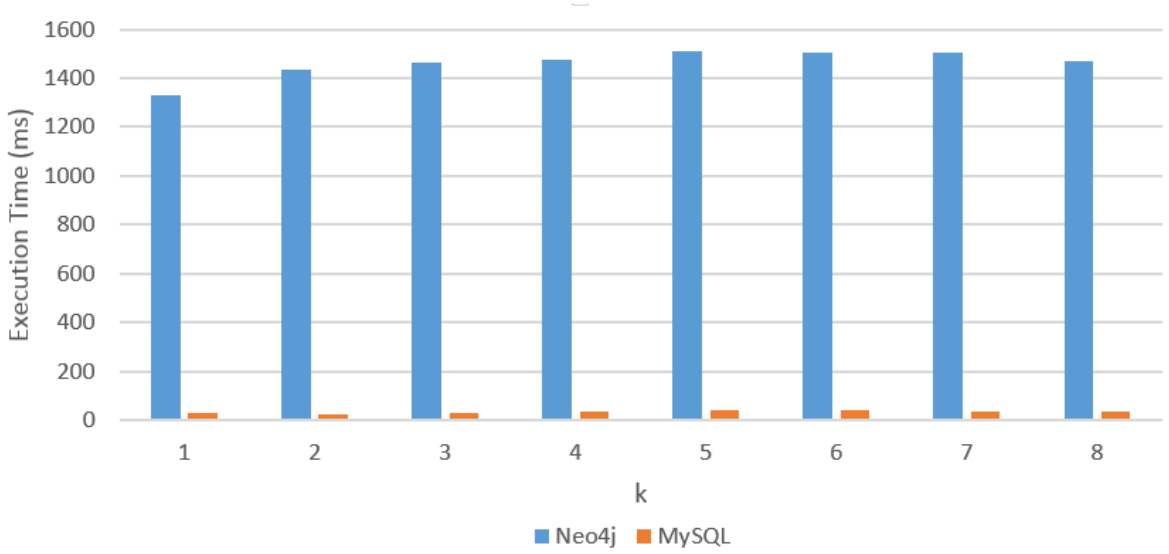

Figure 5.18 Execution time of find all nodes with k out-going degree in "relation1" in a random graph with 1M nodes

cache size varies from 1M to 1G. In Figure 5.26, the execution time for the shortest path query decreases significantly as the query cache size increased from 1M to 1G. When cache size reaches 500M, the execution time becomes smaller than that of Neo4j, which is 87 ms as shown in 5.16. In Figure 5.27, the execution time also decreases as the query cache size increases. When cache size reaches 100M, the execution time in MySQL becomes smaller than that in Neo4j showed in Figure 5.23.

Unlike the cache mechanism in MySQL, cache size in Neo4j is configured in a totally different way. The difference comes from how to store graph data. In Neo4j, there are three different types of memory to consider: OS memory, page cache and heap space. OS memory is the memory reserved for all activities that are not Neo4j related. The default value 1G is good enough for most cases (10). Page cache is used to cache Neo4j data as stored on disk. In our experiment, the default value is 4GB. Heap sizing is the size of available heap memory, which is an important aspect to tuning performance. The default value is 1GB. Generally speaking, it is beneficial to configure a large enough heap space to sustain concurrent operations.

In our experiments, we changed the value of heap size and kept the OS memory and page cache the default value. The initial heap size is 1GB. The RAM of the system we used is 8GB,

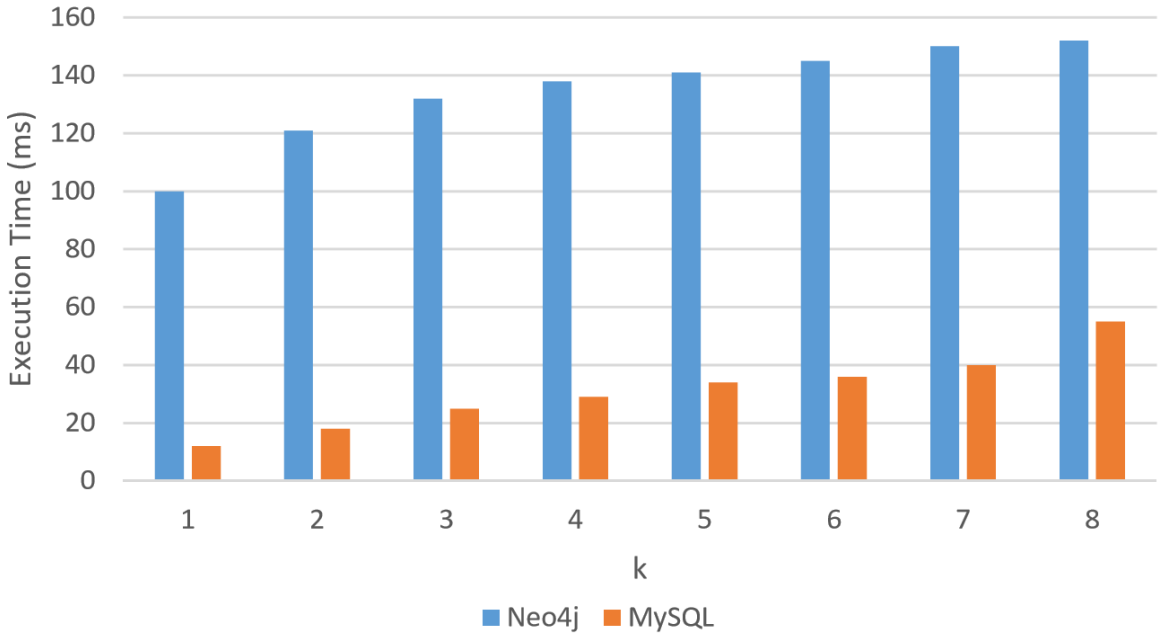

Figure 5.19 Execution time of finding k-hop neighbors of a given node in a Kronecker graph

the maximum heap size we could achieve was 3GB as we had to set 5GB RAM for the default setting of OS system and page cache. With the variation of heap size, we also investigated the shortest path query in 1M random graph and the Q3 in 1M multiple node graph.

As shown in Figure 5.28, for shortest path query, the execution time decreases rapidly as the heap size increases from 1GB to 2GB. The execution time does not change from 2GB to 3GB. It indicates that 2GB was good enough for shortest path query. Compare to Figure 5.26, we notice that the execution time in MySQL with a 500MB query cache size still is slower than that in Neo4j with a 3GB heap size. As showed in Figure 5.29, the change of heap size does not change the execution time significantly, which indicates that 1GB heap size was good enough for Q3. We also notice that the execution time in MySQL with the optimal query cache size is almost the same as that in Neo4j. In summary, we found the optimal cache size for both MySQL and Neo4j and the execution time in MySQL is greater than that in Neo4j for shortest path query,while for Q3, both of them perform the same under the optimal cache size.

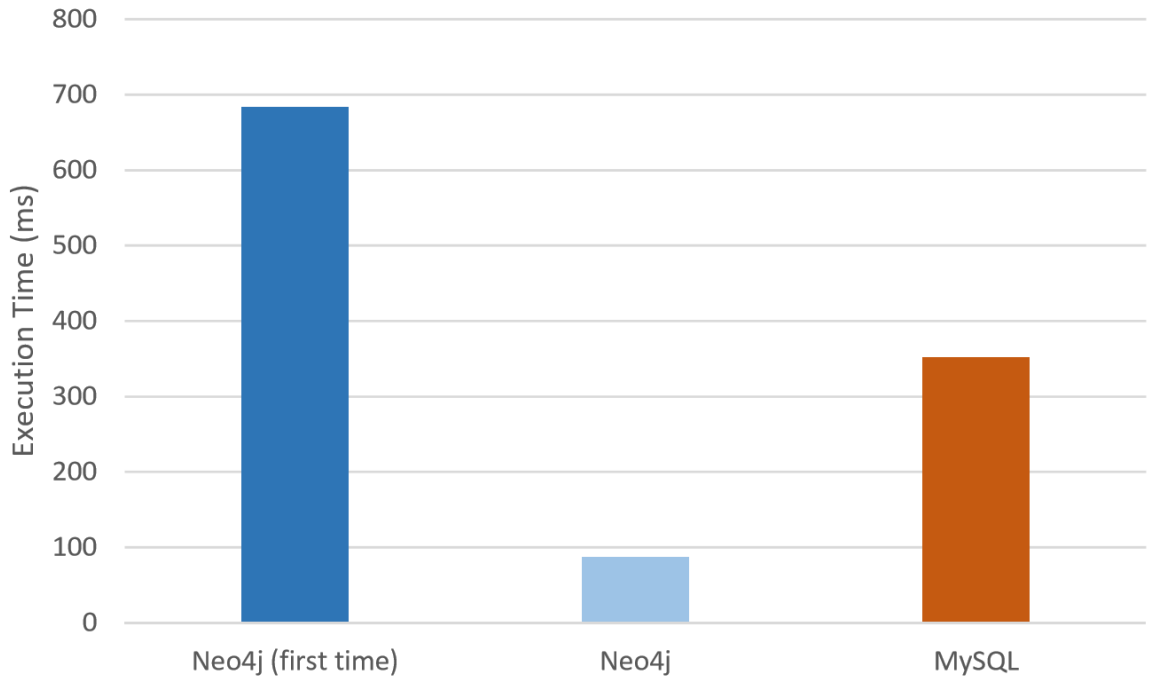

Figure 5.20 Execution time of shortest path in a Kronecker graph

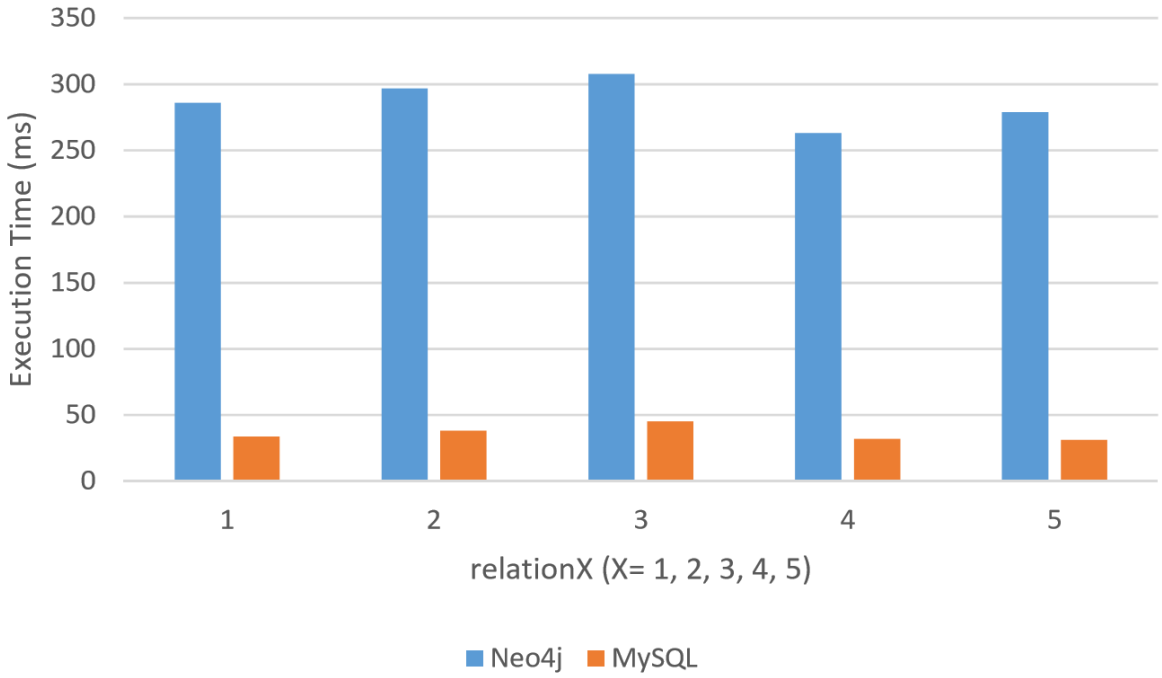

Figure 5.21 Execution time of Compute a degree centrality of five different edge types in a Kronecker graph.

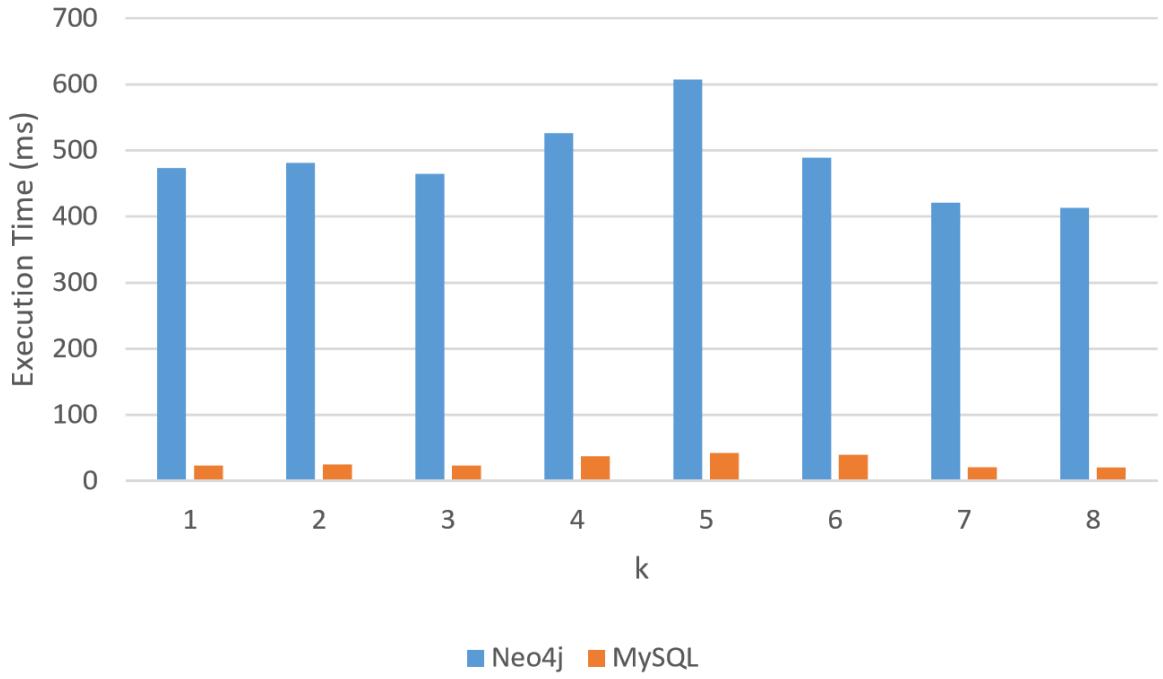

Figure 5.22 Execution time of find all nodes with k out-going degree in "relation1" in a Kronecker graph

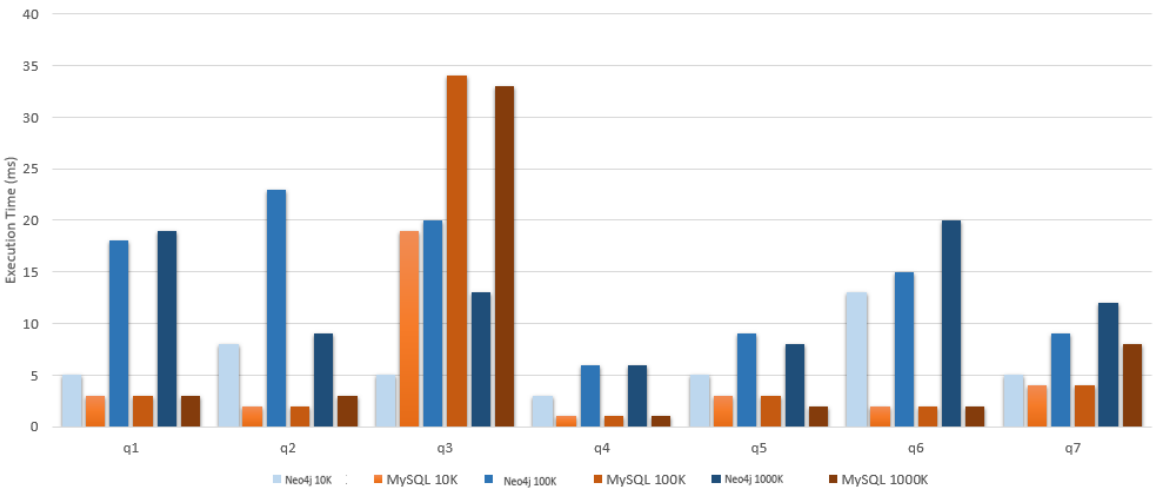

Figure 5.23 Average execution time of 7 queries in 6 different databases for 30 times

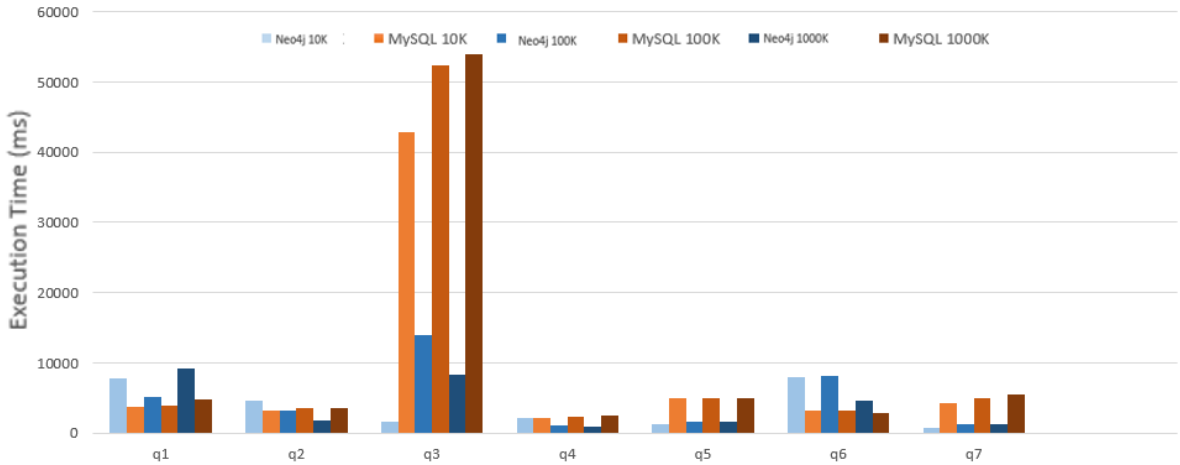

Figure 5.24 Total execution time of 7 queries in 6 different databases for 50 different starting nodes

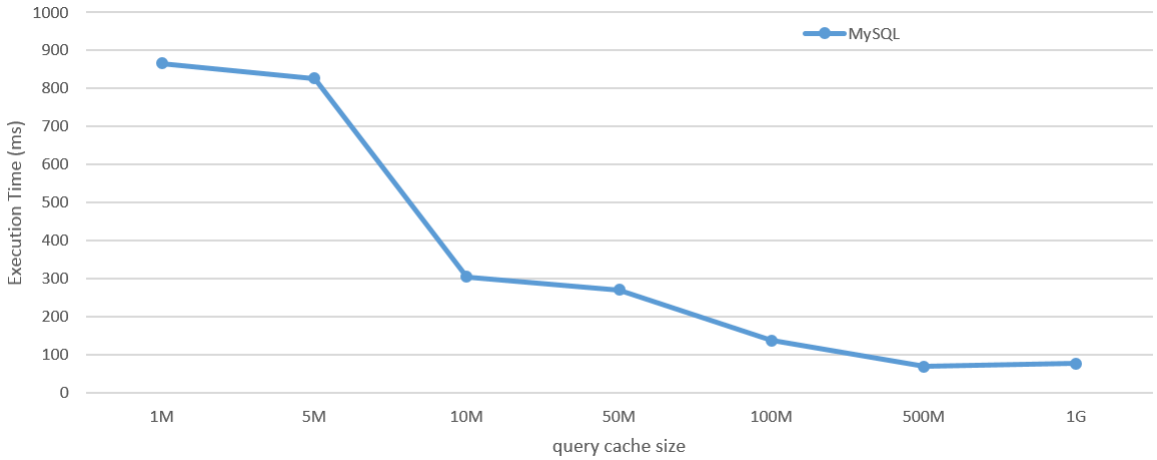

Figure 5.26 Execution time of shortest path under different cache sizes in MySQL

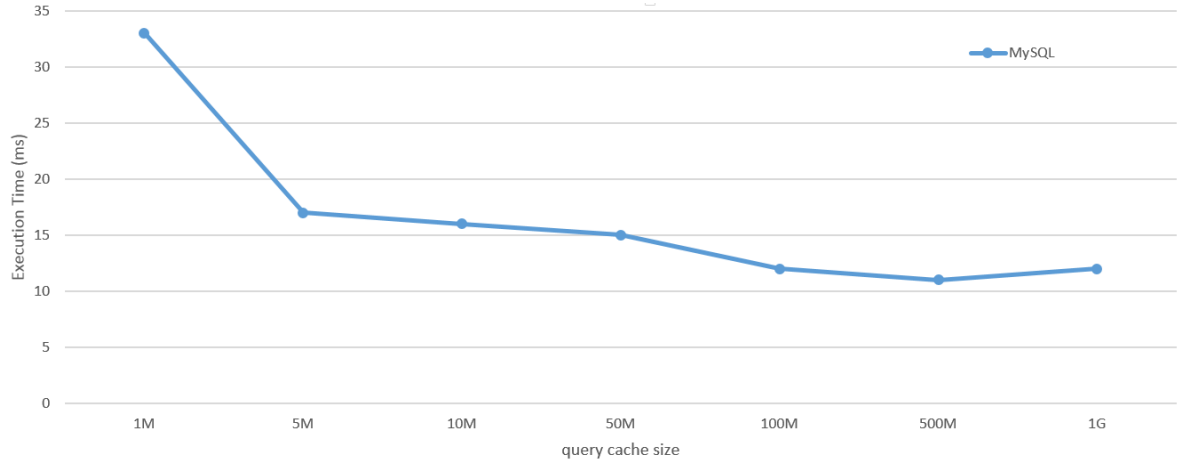

Figure 5.27 Execution time of q3 under different cache sizes in MySQL

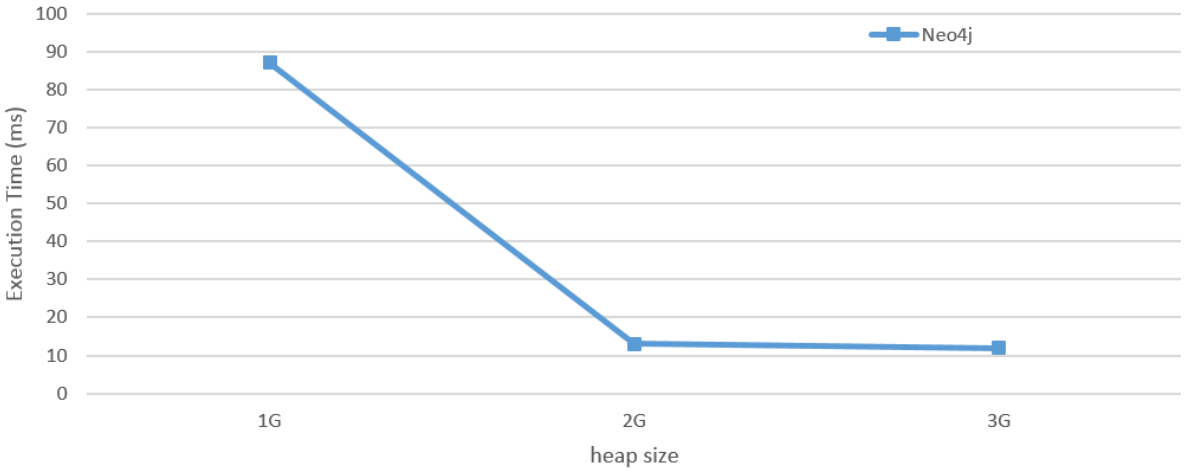

Figure 5.28 Execution time of the shortest path under different heap sizes in Neo4j

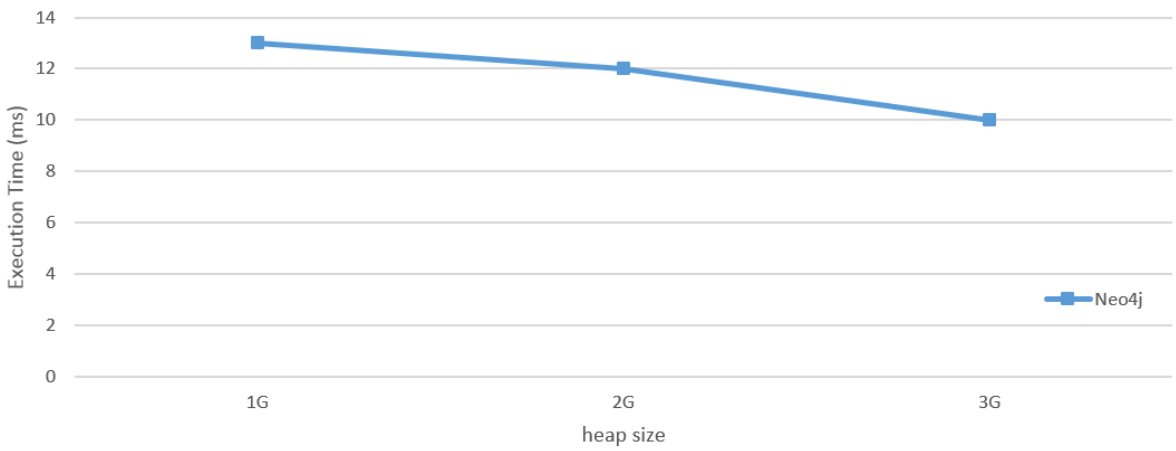

Figure 5.29 Execution time of the shortest path under different heap sizes in Neo4j

CHAPTER 6. SUMMARY AND DISCUSSION

We have described our Cyclone benchmark for graph databases. It comes with two different graph data models: simple and complex models. The simple model has only one node type, but allows us to study the effect of the graph structure. The complex model has multiple node types. The unique feature of this benchmark is that it lends itself for detailed studies of scalability with the ability to specify course grain and fine grain selectivity factors using node and edge attribute values. The benchmark allows for studies of the impact of indexing both on an integer attribute and on a string attribute as well as correlation among attribute values. Given a graph with edges and nodes, our attribute graph generator annotates the edges and nodes with attributes and attribute values. It is flexible for benchmarking random graphs, Kronecker graphs, and power-law graphs. The benchmark includes extensive workload including graph structure queries (e.g., computing a degree centrality and finding a shortest path between a pair of nodes) and data related operations involving attribute values. Future work includes support for benchmarking of concurrent users and benchmarking the transaction capability of the graph database management systems.

BIBLIOGRAPHY

[1] https://www.oracle.com/database/index.html

[2] https://www.microsoft.com/en-us/server-cloud/products/sql-server/

[3] https://www.mysql.com/

[4] http://www.ibm.com/analytics/us/en/technology/db2/

[5] B. A. Eckman and P. G. Brown. *Graph data management for molecular and cell biology* IBM J. Res. Dev., 50(6):545560, (2006).

[6] Allegrograph rdfstore web 3.0s database.

[7] Objectivity Inc. Infinitegraph.

[8] Ibm system g: Graph database overview.

[9] http://www.hypergraphdb.org/

[10] http://neo4j.com/

[11] https://github.com/thinkaurelius/titan

[12] http://spark.apache.org/graphx/

[13] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. *Pregel : A system for large – scale graph processing.* In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 10, 135146, (2010)

[14] Jim Webber Ian Robinson and Emil Eifrm. *In Graph Databases* 2nd ed. OReilly Media.

[15] Distributed graph database.

[16] Alekh Jindal. *Benchmarking graph databases*, (2013).

[17] Charu Tyagi Shalini Batra. *Comparative analysis of relational and graph databases*. International Journal of Soft Computing and Engineering (IJSCE), 2:509512, (2012).

[18] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn 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 SE 10, 42:16, (2010).

[19] Sotirios Beis, Symeon Papadopoulos, Yiannis Kompatsiaris*Benchmarking Graph Databases on the Problem of Community Detection* New Trends in Database and Information Systems II, 3-14, (2015)

[20] Draen Odobai Mario Miler, Damir Medak. *The shortest path algorithm performance comparison in graph and relational database on a transportation network* Scientific Journal on Traffic and Transportation Research, (2014).

[21] V. Vansteenberghe S. Jouili. *An empirical comparison of graph databases*. 708 715, (2013).

[22] John Gilbert Jeremy Kepner David Koester Eugene Loh Kamesh Madduri Bill Mann Theresa Meuse Eric Robinson David A. Bader, John Feo. *Hpc scalable graph analysis benchmark*, (2009).

[23] Linked data benchmark.

[24] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. *A performance evaluation of open source graph databases* In Proceedings of the First Workshop on Parallel Programming for Analytics Applications, PPAA 14, 1118, (2014).

[25] M. Dayarathna and T. Suzumura. *Xgdbench : A benchmarking platform for graph stores in exascale clouds*. In Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on, 363370, (2012).

[26] Khaled Ammar and M. Tamer Ozsu. *Wgb : Towards a universal graph benchmark*. In Advancing Big Data Benchmarks: Proceedings of the 2013 Workshop Series on Big Data Benchmarking, (2013)

[27] M. Ciglan, A. Averbuch, and L. Hluchy. *Benchmarking traversal operations over graph databases*. In Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference, 186 189, (2012).

[28] Draen Odobai Mario Miler, Damir Medak. *The shortest path algorithm performance comparison in graph and relational database on a transportation network*. Scientific Journal on Traffic and Transportation Research, (2014).

[29] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. *Linkbench : A database benchmark based on the facebook social graph*. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 13, 11851196, (2013).

[30] Objectivity Inc. *A performance and distributed performance benchmark of infinitegraph and a leading open source graph database using synthetic data*, (2012).

[31] David J DeWitt. *The wisconsin benchmark : Past, present, and future*, (1993).

[32] http://www.tpc.org/tpcc/. $TPC - C$ *benchmark*.

[33] Bristlecone performance test tools.

[34] Open source development lab database test suite.

[35] http://ldbcouncil.org/

[36] www.graphanalysis.org/index.html

[37] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. $R - MAT$: *A Recursive Model for Graph Mining*, 43, 442446.

[38] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. *Bigbench : Towards an industry standard benchmark for big data analytics* In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 13, 11971208, (2013).

[39] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, K. Zhan, Xiaona Li, and Bizhu Qiu. *Bigdatabench : A big data benchmark suite from internet services.* In 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 488499, (2014).

[40] P ERDdS and A RWI. On random graphs i. Publ. Math. Debrecen, 6:290297, (1959).

[41] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. *Kronecker graphs : An approach to modeling networks.* J. Mach. Learn. Res., 11:9851042, (2010).

[42] Another python graph library 0.8.1.

[43] George K. Zipf *Human Behavior and the Principle of Least Effort*. Addison-Wesley (1949).

[44] George K. Zipf *The Psychobiology of Language*. Houghton-Mifflin (1935).

[45] https://www.uvm.edu/ dhowell/StatPages/More_Stuff/CorrGen.html