Spring 2019

A platform for benchmarking Database Management Systems: CyDIW

Ishita Prakash

A platform for benchmarking Database Management Systems: CyDIW

by

Ishita Prakash

Major: Computer Science

Program of Study Committee:
Shashi Gadia, Major Professor
Robyn Lutz
Simanta Mitra

Iowa State University
Ames, Iowa
2019

ACKNOWLEDGEMENT

I would like to thank Professor Shashi Gadia, for providing the insight and expertise that helped me to complete the project. I thank Professor Robyn Lutz and Professor Simanta Mitra for assisting me through the work. I am grateful for their comments on the initial version of this paper.

I would also like to thank my friends for sharing their perspective and ideas, that improved the quality of my work.

TABLE OF CONTENTS

ABSTRACT

Relational databases have been popular since a very long time. They store data in a structured way providing optimisation and simplicity. Although the strict structure does not provide flexibility to the developer. It works on having primary key and foreign keys. Joins are created at runtime which eat a lot of memory and time. Graph databases involve nodes and edges. Each node represents an entity and each edge represents a relationship. So when an equivalent JOIN operation is run in graph databases, it saves the time from doing extensive searching. Although graph databases have started gaining a lot of popularity recently, everyone is interested in comparing these popular database management systems. We propose CyDIW (Cyclone Database Implementation Workbench) benchmarking, for measuring performance of MySQL (Relational Database Management System) and Neo4j(Graph Database Management System) in various aspects. CyDIW provides a one-click system to perform an extensive experiment to compare query runtimes. This benchmarking involves a graph model with multiple node types and CRUD operations. We discuss in details the process of creating Neo4j adapter for CyDIW system, which was successfully completed. After extensive study, we see that MySQL is faster than Neo4j for most of the CRUD operations.

CHAPTER 1. OVERVIEW

Graph databases have become very popular because of there flexibility and structure. Nodes and edges represent connections, which can very well be seen real-world systems like a social network, web network or transportation network. In a RDBMS(Relational DataBase Management System) the entities are linked through foreign keys of properly structured tables with individual keys, this makes it very difficult to relate it to a real-world entity [1]. In Fig. 1 and 2 we can see how the basic architecture of RDBMS and GDBMS differ.
We are using a graph model to perform our experiments. A graph model consists of nodes and edges. Each node and edge can have various properties associated to it.

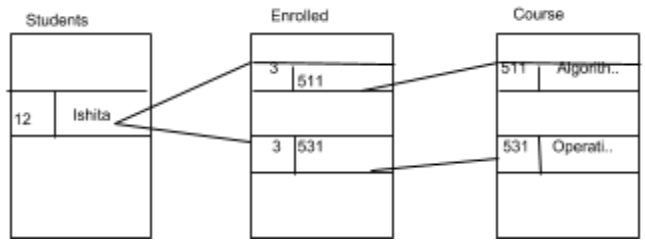

Fig. 1. RDBMS architecture

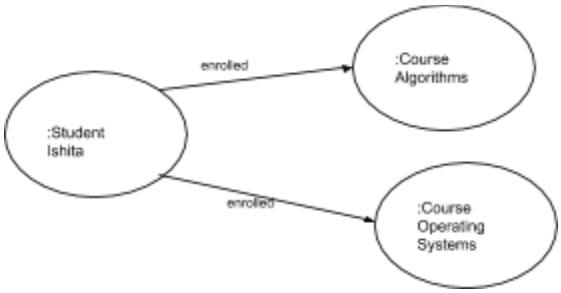

Fig. 2. GDBMS architecture

1.1 Benchmarks

We have discussed various advantages of graph databases over relational databases, but there are still studies going on to identify which has a better performance. We have based our study on a similar paper which attempts to do the same[1]. But we are proposing a system which will make it very easy to have a comparative analysis of any number of database engines, mainly focusing on MySQL and Neo4j in this paper.

CyDIW is a centralised system which provides interface to run any command based system as a client on it. We provide a one-click system to feed any number of queries to these two database system and see their performance graph. We will perform the benchmarking on the data size of 5k nodes.

CHAPTER 2. EXPERIMENT SETUP

2.1 GRAPH MODEL

Here we discuss the graph model which will be used to evaluate the performance of MySQL and Neo4j. We have used a complex graph model with seven node types and ten edge types as seen in Fig. 3.. This graph can represent a social network with different entities such as people, messages, mails, etc connected through different relationships.
The graph structure is pre modeled as seen in Fig. 3. For example a node of type node3 is connected to node2 through an edge of type edge7. Each node has seven attributes: UniqueIdx, UniqueNdx, Ten, StringIdx, StringNdx, CorX, and CorY. Each edge has three properties: Ten, StringIdx and StringNdx.

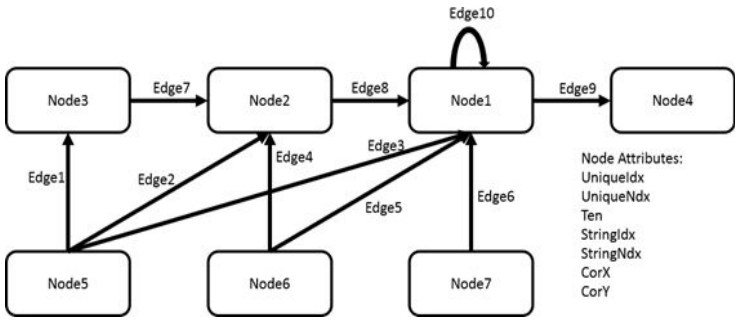

Fig. 3. Graph model used for the experiment

2.1.1 Data Generator

Each node has seven attributes. Amongst which UniqueIdx, UniqueNdx, Ten, StringIdx, StringNdx are randomly generated. CorX and CorY are correlation coefficient showing how strongly a pair of coefficient is related.

We use Graph Database driver to connect to Neo4j through a Java program implemented by us, which is used to generate the huge data. Figure shows a snippet of the code used for data generation. Fig. 4 shows how the graph database looks for node1.

```
private static List<String> generateGraph( Transaction tx )
{
    int id=26;
    Random random = new Random();
    for (int i=1;i<=500;i++) {
        //generate correlated
        Correlation cor = new Correlation(10, 100);
        double[][] correlation = cor.calc();
        int index = random.nextInt(100);
        String genNodes="CREATE (:node"+j+"{ID:"+ id+",uniqueIdx: "+id+",uniqueNdx: "
                +id+",ten: "+(random.nextInt(10) + 0)+",corX:"+ correlation[0][index]
                +",corY:"+correlation[1][index]+",stringIdx:"+ i+",stringNdx: "+i+"})";
        //creating edges
        String genEdges="MATCH (t1:node1 { ID:"+ (random.nextInt(24) + 1) +"}),(t2:node1{ ID:"
                + (random.nextInt(24) + 1) +"}) CREATE (t1)-[:edge" + 10 + "{ID:"+ id+",ten: "
                +(random.nextInt(10) + 0)+",stringIdx:"+ i+",stringNdx:"+i+"}]->(t2)";
        id++;
        long timeNow = System.nanoTime();
        tx.run(genNodes);
        tx.run(genEdges);
        long timeThen = System.nanoTime();
        long time_difference=timeThen-timeNow;
        System.out.println("Time Difference: "+time_difference+"ns");

    }
     return null;

}
```

Fig. 4. Data generation Java code

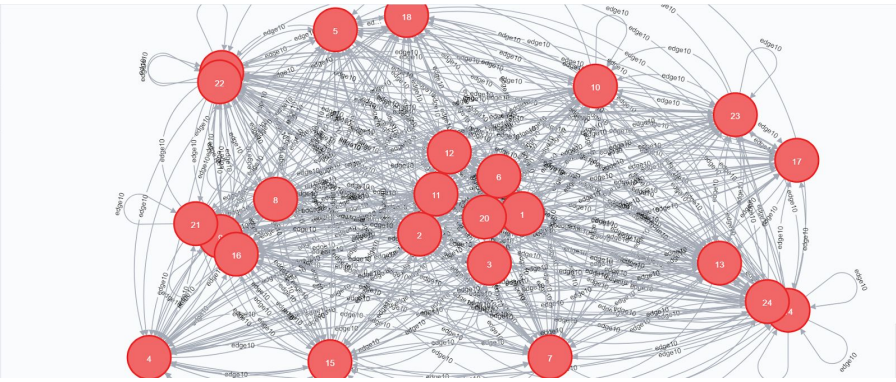

Fig. 5. Graph database as seen in Neo4j UI

For MySQL, we use bulk insertions. For instance: “INSERT INTO edge10 (AuniqueIdx,BuniqueIdx,ten,stringIdx) VALUES(32, 41, 15, 8),(76, 18, 11, 933),(53, 28, 20, 22),(54, 66, 75, 23),(24, 63, 58, 57),(61, 89, 90, 71),(55, 4, 66, 5),(44, 42, 95, 48),(90, 88, 45, 809),(17, 12, 85, 69),(7, 4, 41, 52),(5, 99, 17, 94),(84, 30, 46, 1),(3, 14, 37, 68),(94, 92, 54, 28),(10, 87, 98, 39),(80, 84, 25, 47),(10, 40, 65, 64),(90, 79, 38, 37),(91, 90, 69, 66),(24, 16, 41, 43),(31, 76, 46, 78),(25, 94, 88, 20),(40, 66, 75, 20),(37, 41, 24, 88);”

AuniqueIdx	BuniqueIdx	ten	stringIdx
3	14	37	68
5	99	17	94
7	4	41	52
10	40	65	64
10	87	98	39
17	12	85	69
24	16	41	43
24	63	58	57
25	94	88	20
31	76	46	78
32	41	15	8
37	41	24	88
40	66	75	20
44	42	95	48

Fig. 6. Relational database as seen in MySQL Workbench

2.1.2 Queries

We take a number of different kind of queries so that it touches a significant amount of data. These queries are borrowed from [2].

•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 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 3. PROPOSAL AND IMPLEMENTATION

We propose to use CyDIW for benchmarking the performance of two major database engines, MySQL and Neo4j. We will now discuss how we use CyDIW inbuilt functions and how we built an adapter for Neo4j to run as a client on CyDIW.
Note: SQL adapter is implemented beforehand, so we use the previously built adapter for this experiment.

3.1 CyDIW ARCHITECTURE

Each client running on CyDIW needs to provide an adapter which acts as a facilitator between CyDIW and the client system. CyDIW has an XML file named SystemConfig.xml which maintains the classpath information for CyDIW and client systems. With the help of this information, the system class loader loads all the client systems at the start of CyDIW. This keeps track of all the clients which are running. When a command is run, it will search for that particular client using the prefix which is basically a '$' symbol followed by a string, for eg: $neo4j is the prefix configured for Neo4j, as seen in the Fig. 7. *LibraryPath* defines the path of folder containing neo4j drivers. *ClientAdapter* gives the path to neo4j adapter. Details on CyDIW features can be found in [3].

```
<Client Name="neo" Prefix="$neo4j" Enabled="yes"
 ClassPath=""
 LibraryPath="cyclients\neo\lib"
   WorkspacePath=""
 ClientAdapter="cyclients.neo.Adapter.adapterNeo" />
```

Fig. 7. Neo4j configurations in SystemConfig.xml

3.1.1. Logging Runtime in CyDIW

Any client command in CyDIW has the option of using *OutputClause* and *LoggingClause*. We use *OutputClause* and *LoggingClause* with "run" command here.
"Run" command takes in $$prefix and $$query as the parameters. $$prefix is a variable pointing to the client we want to run and $$query is the variable holding the actual command we want to execute on that particular client.

OutputClause is used as “out>>OutputFile” which redirects the output to a particular file, which otherwise is printed on the output pane by default. Here we redirect out output to an xml file, which then is used for logging the time(discussed further in this section).

```
$CyDB:>run $$prefix[1] $$query[$$j] out>> ($$prefix[1])_query($$j).xml
```

Fig 8. Use of out command in CyDIW

This xml file is used by the *LoggingClause*. If this file is already exists, it gets overridden. We call *LoggingClause* by “log(time | custom)>>LogTag LogFile”. *LogTag* is XML tag inside <>, “<query>” in our case. We are using the “time” option because we are interested in runtime. It simply records the elapsed time in milliseconds between dispatch and return of the command when it is executed. We initialise a log file by running “$CyDB:> createLog <root> benchmarkQ.xml;”. After running the queries *benchmarkQ.xml* looks like as shown below.

```
<?xml version="1.0"?>
- <root>
  - <loop1 var="$MySQL">
    - <loop2 var="Q6">
        <query>32</query>
        <query>14</query>
        <query>12</query>
      </loop2>
    </loop1>
  - <loop1 var="$neo4j">
    - <loop2 var="Q12">
        <query>172</query>
        <query>103</query>
        <query>71</query>
      </loop2>
    </loop1>
    #
</root>
```

Fig. 9. Example of a benchmarkQ.xml instance

3.1.2 Graphs in CyDIW

R is utilised to draw a bar graph from a set of 2D data stored in an XML file as discussed in section 3.1.1. The R client is already implemented in CyDIW by default on startup. The prefix for R is $R. We are running a batch to generate the plot as shown in Fig. 10.

```
// Step 5. If Project R has been installed, compute and display the performance graph
othewise display a sample graph;

$CyDB:> if ($$Project_R_Is_Installed == "Yes") {
$CyDB:> displayFile cyclients/r/workspace/R_code.txt;
$R:> CMD BATCH cyclients/r/workspace/R_code.txt;
// Display sample and computed plots. The computed plot Plot.pdf is mentioned in the R_code;
$CyDB:> displayPDF CyWorkspace/Plot.pdf;
} else {
// Display sample and computed plots. The computed plot Plot.pdf is mentioned in the R_code;
$CyDB:> displayPDF ComS363\Demos\Datasets\SamplePlot.pdf;
}
```

Fig. 10. Batch for plotting a 2D graph

```
args=(commandArgs(TRUE))

if(length(args)==0){
    print("No arguments supplied. Using default values")
    ##supply default values
    inputFile = 'CyWorkspace/benchmarkSubset.xml'
    outputFile = 'CyWorkspace/Plot.pdf'
    title = 'CyDIW Experimental Results'
    xlabel = 'Queries'
    ylabel = 'Execution Time (in Milliseconds)'
} else {
    for(i in 1:length(args)){
         eval(parse(text=args[[i]]))
    }
}

pdf(outputFile)

library(XML)
doc = xmlRoot(xmlTreeParse(inputFile))
```

Fig. 11. R code used to plot the graph

3.2 NEO4J ADAPTER FOR CYDIW

In this chapter we talk about the implementation of CyDIW benchmark. The major work was done in implementing the adapter of Neo4j for CyDIW. Neo4j is running as a client on CyDIW. I have implemented a Java program to generate the test dataset. Relational database generation is done using Bulk Insertion.

3.2.1 Neo4j Adapter

Neo4j has many integration tools for Java[4]. We installed Neo4j server on our local machine, which runs the bolt protocol using the neo4j drivers. The drivers are plugged in the CyDIW libraries. CyDIW then gives access to these drivers, mainly org.neo4j.driver each time neo4j connection is asked while running the commands.

There are mainly two Java programs running in order to run neo4j commands on CyDIW, *execQuery.java* and *adapterNeo.java* implemented by me. *execQuery.java* has four methods to perform functions namely *execQuery(constructor)*, *close*, *startSession* and *executeQuery*. The constructor takes neo4j server url, username, password and the query string as parameters and initiates the connection using GraphDatabase driver. Method close() closes the connection when called. Method startSession() starts a transaction and calls executeQuery with the query String. Lastly, executeQuery actually runs the query and returns a string containing key-value pair.

```
package cyclients.neo;

import org.neo4j.driver.v1.*;

public class execQuery implements AutoCloseable
{
    private final Driver driver;

    public execQuery( String uri, String user, String password,String query)
    {

        driver = GraphDatabase.driver( uri, AuthTokens.basic( "neo4j", "1234" ) );

    }

    public void close() throws Exception

    public List<String> startSession(String query )

  private static List<String> executeQuery( Transaction tx,String query )

    public static void main( String[] args ) throws Exception
}
```

Fig. 12. execQuery.java snippet

To register a new client in CyDIW, the client adapter is used, which is named *adapaterNeo* in this case. CyDIW provides a client interface and a client factory class to implement the adapter. These classes provide methods which can be overridden to perform certain function, such as *initialize*, *execute* and *getCustomLogData*. We are using only *execute()* here in this case. *execute()* takes the clientID and command as parameters. Integer clientID is an internal ID assigned to a client in CyDIW. This program implements an abstract class ClientFactory, one of the classes discussed above, to override the *execute()* method. This method takes in the

command and calls the constructor of *execQuery.java,* which returns the query results. The *execute()* method then displays the results on CyDIW interface.

```
public void execute(int clientID, String command) {
String s="";
String[] text = command.split(" ");
int len=command.length();
execQuery e=new execQuery("bolt://localhost:7687" ,"neo4j", "1234",command);
if(text[0].equals("connect")){
dbgui.addOutput("Connection successfull!");
dbgui.addOutput("Connection details: bolt://localhost:7687");
dbgui.addOutput("Username:neo4j, Password: **** ");
}
else if(text[0].equals("close")){
try{
e.close();
}
catch(Exception f){}
dbgui.addOutput("Connection closed successfully!");
}
else if(text[2].startsWith("MATCH"))
{
        dbgui.addOutput("Running the following neo4j query");

        String fin=command.substring(21,len);

        dbgui.addOutput(fin);
        List<String> a=e.startSession(fin);
    for(int i = 0; i < a.size(); i++) {
         s=s+a.get(i)+"\r\n";
        }
             dbgui.addOutput(s);
    }
}
```

Fig. 13. Neo4j adapter for CyDIW

CHAPTER 4. EXPERIMENTS AND RESULTS

4.1 QUERIES

Query 1

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;
```

Neo4j:

```
match (a:node1 {UniqueIdx : 1}) -[:edge10]-> (n:node1), (n)-[:edge9]->(b:node4)
where b.ten < 2
return n
order by b.ten;
```

Query 2

SQL:

```
select count(e4.BuinqueIdx), e4.BuniqueIdx
from edge8 e1
join node2 n2 ON n2.uniqueIdx = e1.AunqiueIdx
join node1 n1ON 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);
```

Neo4j:

```
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;
```

Query 3

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;
```

Neo4j:

```
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;
```

Query 4

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 byn3.ten;
```

Neo4j:

```
match (n3:node3)-[:edge7]->(n2:node2)-[r:edge8]->(n1:node1{UniqueIdx:1})
return n3.uniqueIdx
order by n3.ten;
```

Query 5

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;
```

Neo4j:

```
match(forum:node7)-[r2:edge6]->(friend:node1)<-[:edge10]-(person:node1{uniqueIdx:1})
return forum;
```

Query 6

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;
```

Neo4j:

```
match (n3:node3)-[:edge7]->(n2:node2)-[r:edge8]->(n1:node1 {uniqueIdx:1})
return n3
order by n3.ten;
```

4.2 STORING GRAPH DATA IN SQL

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*.

4.3 EXECUTION

The experiment was performed on a Windows PC with Processor Intel(R) Core(TM) i5-7440HQ CPU @ 2.80GHz, 2801 Mhz, 4 Core(s), 4 Logical Processor(s), Neo4j version 1.1.1.3 and MySQL 8.0. As discussed in section 3.1,1, CyDIW provides a set of inbuilt commands to log runtime for a set of queries. We have used the same commands for our experiments. It makes it really easy to run any number of queries for multiple times, calculate the average runtime, and automatically plot it on a graph(discussed in section 3.1.2), just on click of a button. Following are the results of running queries mentioned in section 4.1 on two engines, MySQL and Neo4j.

4.3.1 CyDIW Demo

We create a batch file for CyDIW which lets us run the entire experiment on one click. This file is saved as text file and can be opened from inside CyDIW to load all the commands. We will discuss major steps of the demo file in this section.
Fig. 14 shows a snippet of the demo file which includes the query execution and benchmarking part.

We start off by clearing old variables and initialising new ones. Here we assign MySQL to $prefix[1] and Neo4j to prefix[2]. Our queries are stored in queries[] array. We also create an XML based log file named *benchmarkQ.xml* to gather performance statistics.

In Step2, which is query execution and logging, we have two nested loops, one for iterating through the database engines i.e., MySQL and Neo4j and the other for iterating through the queries. In the inner loops, we have the first one to warm up the cache, to avoid caching issues(discussed in details later in section) and the second loop is for the actual benchmarking. We have already discussed the run and out commands in detail before (Section). The log times are stored in *benchmarkQ.xml*.

```
// Step 2. Query execution and logging of benchmark statistics;
$CyDB:> foreach $$i in (1) log time >> <loop1 var="($$prefix[$$i])"> benchmarkQ.xml
{
        // Inner loop to iterate through XMark queries;
      $CyDB:> foreach $$j in (1) log time >> <loop2 var="Q($$j)"> benchmarkQ.xml
     {

                //warm up 3 times;
                $CyDB:> foreach $$k in [1,3]
                {
                        $CyDB:>run $$prefix[1] $$query[$$j];
                }

                 // execute each query 10 times and log performance;
                $CyDB:> foreach $$k in [1,10]
               {

                                $CyDB:>run $$prefix[1] $$query[$$j] out>> ($$prefix[1])_query($$j).xml log time >> <query> benchmarkQ.xml;

                }


        }
}
```

Fig. 14. Batch for executing the queries and logging runtime

In Step 3 we display the log file for user to look at the actual run times. Step 4 calculates *benchmarkSubset.xml* for recording the average execution time for each engine and each query.

```
// Step 4. Calculate and display benchmarkSubset.xml for recording the average execution time for each engine and query;
$Saxon:> for $e in doc("CyWorkspace/benchmarkQ.xml")//loop1
return <loop1> {
   $e/@var,
   for $f in $e/loop2
   return <loop2> {
     $f/@var,
     let $g :=  $f/query/text()
       return <avg> {avg($g)} </avg>
  } </loop2>
} </loop1>
out >> benchmarkSubset.xml;
```

Fig. 15. Batch for calculating average runtime

Finally, the *benchmarkSubset.xml* is passed to R code which uses it to generate the plot. This functionality has been discussed in details earlier in section. The plotted graph automatically pops up for the user to see the results.

4.3.2 Plotted Results

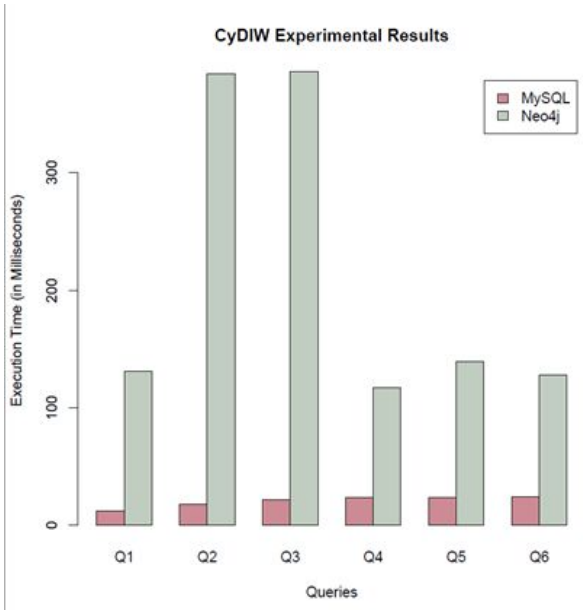

Fig. 16. Results without warm up cache

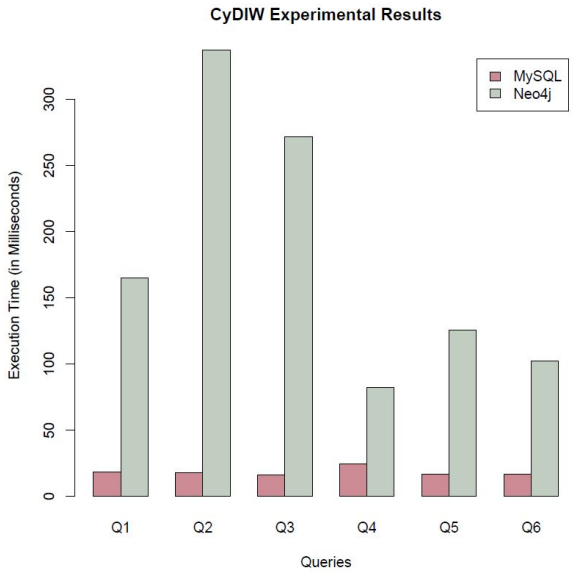

Fig.17. Results with warm up cache

4.4 PERFORMANCE AND CACHE

We were concerned about the caching issue as both Neo4j and MySQL have cache memories when running queries. Our experiment involves running a query multiple times to get an average execution time, so there is definite caching.
Neo4j has a two-layered cache architecture. cache_type and dbms.pagecache.memory. The default page cache size is set to 512 Megabytes. We could disable both these cache, but then we would only be measuring speed of our IO subsystem, since every query would access the disk every time[5].
Similarly in MySQL, InnoDB buffer pool contains indexes and data accessed frequently. The default buffer pool size is set 128 Megabytes[6].

So, we use "warm up caches" methodology to avoid the hindrance of cache in our experimental results. Warming up the cache saves us from having huge difference between the first run and the 2nd run. We run the queries 3 times each before actual logging starts. CyDIW batch is as shown below.

```
//warm up 3 time
 $CyDB:> foreach $$k in [1,3]
        {
            $CyDB:>run $$prefix[1] $$query[$$j];
         }
```

We performed the experiment once without warm up cache and once with it, results shown in Figure. We do not notice a huge difference in performance between the two set ups. This is probably because we are calculating average run time for plotting the graph. That cover ups the huge time difference between the first and the second run. In Fig. we can see that when we run the experiment without warm up, the first run takes a lot of time. When we do the warm up before actual logging, the time difference decreases.

```
- <loop1 var="$neo4j">
   - <loop2 var="Q7">
        <query>168</query>
        <query>119</query>
        <query>106</query>
     </loop2>
```

Fig.18. Runtimes without warm up cache

```
- <loop1 var="$neo4j">
   - <loop2 var="Q7">
        <query>123</query>
        <query>111</query>
        <query>101</query>
     </loop2>
```

Fig.19. Runtime with warm up cache

As shown in Fig, MySQL performs better than Neo4j. Our results are in line with the results shown in [1]. One thing to observe in Query 4 is that the difference in MySQL and Neo4j runtime is noticeably lesser. This is because Query4 includes Union which is an expensive operation for MySQL. But even so, MySQL performs better than Neo4j. While [1] does an extensive study on the performance, we focus our work on CyDIW as a platform to perform the experiments.

CHAPTER 5. SUMMARY

We are proposing a benchmarking system which makes it very easy to analyse and compare performance of two popular database engines. CyDIW benchmarking provides quick and convenient way to plug in the engines and the queries, and get the results plotted in a 2D graph. The benchmarking is reliable as we tackled the cache memory issue and we run a query multiple times to get an average run time. Also, the run time does not include the time spent on connecting to the database server, as that is done only once in the very beginning before the logging starts. We have used a complex graph model which very well represents a real world social networking system. Our proposed system makes it convenient for the user to benchmark engines, as he/she does not have to worry about underlying drivers, connections or servers running in the background. The user can give as many queries as needed to see the engine performance. We dedicated a major part of our time in developing Neo4j adapter for CyDIW, and it was successfully completed. Future work would include adding adapters for other new engines which are emerging and see how they perform as compared to the traditional ones. We would also like to have a better system which would allow the user to enter his/her server urls, username, password and the queries in an interactive manner.

REFERENCES

[1] https://neo4j.com/developer/graph-db-vs-rdbms/

[2] Benchmarking graph databases with cyclone benchmark, Yuanyuan Tang, Iowa State University Ames, Iowa, 2016

[3] A Lightweight Workbench for Database Benchmarking, Experimentation, and Implementation, Xinyuan Zhao ; Shashi K. Gadia, 20 September 2012

[4] https://neo4j.com/developer/java/

[5] https://neo4j.com/developer/kb/warm-the-cache-to-improve-performance-from-cold-start/

[6] https://dev.mysql.com/doc/refman/5.7/en/query-cache.html