San José State University

CmpE226 – Database Design

Date of Submission

Dec 7, 2013

Submitted By

Krishna Nambiar [008639605]
Rahul Mehta [008686574]
Gayathiri Krishnan [007075601]
Manish Belsare [008297146]

Contents

1. Introduction

Though the industry has been dominated by the relational databases over a long period of time, the latest trend in the data needs very different type of database approach. Almost 80% of today's generated data is unstructured and this increased the frustrations of the application developers in storing and processing that semi-structured data. Big Data, Big Users and Cloud Computing are the three interrelated driving factors, why the developers started turning towards the adoption of NoSQL technology.

Handling large amount of data which comes in form of volume, velocity and variety is a big challenge. Big Data and NoSQL technologies deals with data creation, storage, retrieval and most importantly performing analytics of the data in hand which helps discovering useful information, deriving conclusions and helps in decision making. Today organizations are using this huge data to acquire insights into their business. The Big Data landscape is dominated by two classes of technology: systems that provide operational capabilities for real-time, interactive workloads where data is primarily captured and stored; and systems that provide analytical capabilities for complex analysis that may touch most or all of the data. These classes of technology are complementary and frequently deployed together. Operational systems, such as the NoSQL databases focus on servicing highly concurrent requests while exhibiting low latency for responses operating on highly selective access criteria. The NoSQL technologies has various types of databases such as document-based (i.e MongoDB, CouchDb), Key-Value store (DynamoDB, Riak) , Graph(i.e Neo4J) and columnar (i.e Cassandra, Vertica) databases.

For this project we have decided to compare **Neo4j**, a graph based database and **Cassandra**, a columnar data store, and are aiming to create a unique Wikipedia exploration system.

2. Data Source and Analysis of Data

We are using the Wikipedia data dump provided by Wikipedia as our data source. Wikipedia offers free copies of the data to the interested users. All text content is multi-licensed under the Creative Commons Attribution-ShareAlike 3.0 License (CC-BY-SA) and the GNU Free Documentation License (GFDL). 2.1

Observations about downloaded data

1. Page information is provided as XML and MySQL dump files. The XML files contain most of the text and content edit information, while the MySQL dumps have the control data.
2. Individual XML files may be downloaded as compressed archived with a file size ranging from 150MB to 4.5GB. These files expand to independent XML files of sizes from 250MB to 12GB.
3. The MySQL information is provided as mysqldump outputs, with each file around 20GB in size. The file that we required contained 65million+ records in a single insert command.
4. After downloading and processing, our combined dataset was above 85GB in size.

3. Our Goal

We are aiming to visualize Wikipedia information to showcase connections among its pages, a graph based approach, which would help uncover distinct relationships among the different topics, or how one topic may be related to another seemingly unconnected subject.

For example, the title picture shows how Mumbai in India, leads us to Minnesota in the US.

This project could also be expanded in future to show other heuristics like the edits happening on pages, negative vs. positive edits, locations from where the edits happen etc.

4. NoSQL Databases Considered

4.1 Neo4j

Relational database stores data in multiple related tables whereas NoSQL has very different storage models. They are document-oriented, columnar-store, key-value, and graph models. A document-oriented model stores data in JSON format documents, whereas a graph database stores data as a graph, data that are internally connected with an undetermined number of relations among them. A graph database records data in nodes that are organized with relationships.

Neo4j is a popular open-source graph database implemented in java. Neo4j is limited only by physical hardware. That means though the complexity of data storage increases as the application evolves; it has only minimal effect on the system performance. Querying of database is performed through traversals, which can perform millions of traversals in a second. Neo4j is suitable for full enterprise deployment and it guarantees data consistency by enforcing that all operations modifying data occurs within one transaction. Neo4j can be used without any schema and that is why it is called as schema-optional graph database. It helps us to keep the data clean by using constraints. Using these constraints, rules can be specified for how the data should look like. Any data that looks like breaking these rules will be denied. Also Neo4j provides full transaction management called Two Phase Commit, with rollback support over all data sources to keep the data up-to-date. Data retrieval may be done through application APIs, REST web services or through Cypher query language.

4.2 Cassandra

Cassandra is a NoSQL column family representation implementation supporting the Big Table data model and using the architectural aspects introduced by Amazon Dynamo. Some of the strong points of Apache Cassandra is it is highly scalable and highly available with no single point of failure. It is column family design with very high write throughput and good read throughput.

Cassandra data model, data is placed in a two dimensional (2D) space within each column family. To retrieve data in a column family, users need two keys: row name and column name. Its data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key;

within a partition, rows are clustered by the remaining columns of the key. Other columns may be indexed separately from the primary key. It does not support joins or subqueries, except for batch analysis via Hadoop. Rather it supports collections in its data model.

5. Execution

5.1 System Specification

Node 1

Operating System	: Mac OS X Mavericks, 64bit
RAM	: 8GB DDR3
Processor	: 2.4 Ghz Intel Core i5
HDD	: Toshiba 500 GB, 5200RPM, 2 Hitachi 1TB External Drive on Firewire 800
Neo4j	: v2.0 M06 (Enterprise)

Node 2

Operating System	: Mac OS X Mountain Lion, 64bit
RAM	: 8GB DDR3
Processor	: 2.4 Ghz Intel Core i5
HDD	: Toshiba 500 GB, 5200RPM
Neo4j	: v2.0 M06 (Enterprise
Cassandra	:

Node 3

Operating System	: Windows 8, 64bit
RAM	: 8GB DDR3
Processor	: 2.6 Ghz Intel Core i7
HDD	: Intel 128 GB SSD
Neo4j	: v2.0 M06 (Enterprise)

5.2 Database Design

5.2.1 Neo4j Schema

Neo4j is a schema optional database. This means that each node in the database can be different from each other. In this project, we loaded each page as a single node with attributes like title, pageid, name space, contributor, comments and the page text. If a page was mentioned within a page, a relation was created among the two.

We used the new schema indexing, rather than the old auto indexing option, to index page title, page id and the page text. Neo4j does indexing based on an inbuilt Lucene engine. The use of schema based indexing gave us more control on the fields and the type of indexes being created.

As far as relation ships were considered, they had attributes like name space and a key (from page id + to page title). Index was created on the key.

5.2.2 Cassandra Schema

Cassandra does not have a fixed data schema. In Cassandra certain rows can have few items corresponding to the row-key whereas other rows can have millions of items. It allows l users change the schema at runtime. In our data model for Cassandra, we created two column-family :

A) A **wiki.page column-family,** which consists of the attributes title, namespace, redirect, username, comment and commentCount. The **pageid** was the row-key is the mentioned column-family. We also created a secondary index on the title attribute inside our schema. For **pageid 12** in our schema,

```
"12"-> {
    title="Anarchism",
    ns="Default",
    username="Eduen"
    redirect="Computer Accessibilty"
    commentCount= 1
    comment = "This is sourced"
     text ="....."
    }
```

B) A **wiki.links** column-family which consists of composite primary key of **(pageid,cols,linkid)**. This schema was created to decipher the links of particular page with other pages in wikipedia. The **cols** column represents the title of the linked page corresponding to that pageid. Thus using this schema we could find the page links across the whole dataset. In this columns family also pageid was the row-key.

5.3 Data Loading

The biggest challenge we faced was handling the large files in the input dataset. For XML files, we used a Java application implementing SAX parser to read the file as a stream, as the DOM based approach loads the entire file in to the RAM.

Similarly, for loading the mysql dumps, when we ran as it is, it did not finish even after running for 30 hours. So we used special software (Hex Fiend) that could open very large files (25GB), to edit the SQL inside the dump file. We removed all unwanted keys and indexes in the create statement, and introduced partitioning. Further, we also moved the MySQL data folder to an external drive. All these measures enabled us to load the complete file in about 4 hours. The reads from the table was also very fast, since it was indexed and portioned on the exact where clause we were using.

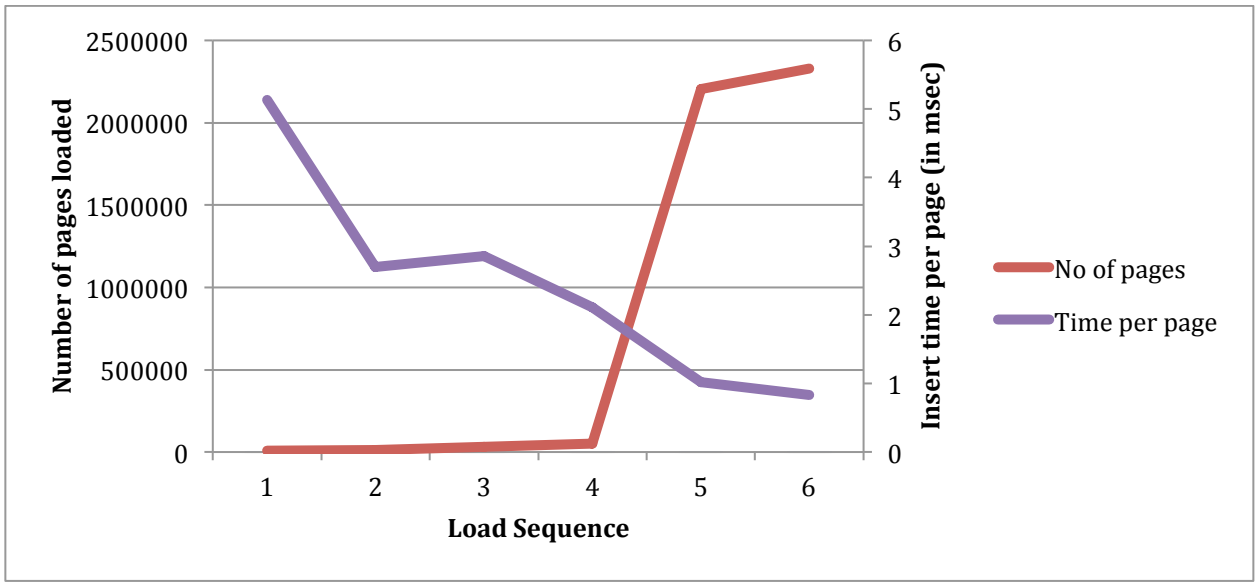
Similar code was used for inserting records into both Neo4j and Cassandra.

5.3.1 Neo4j

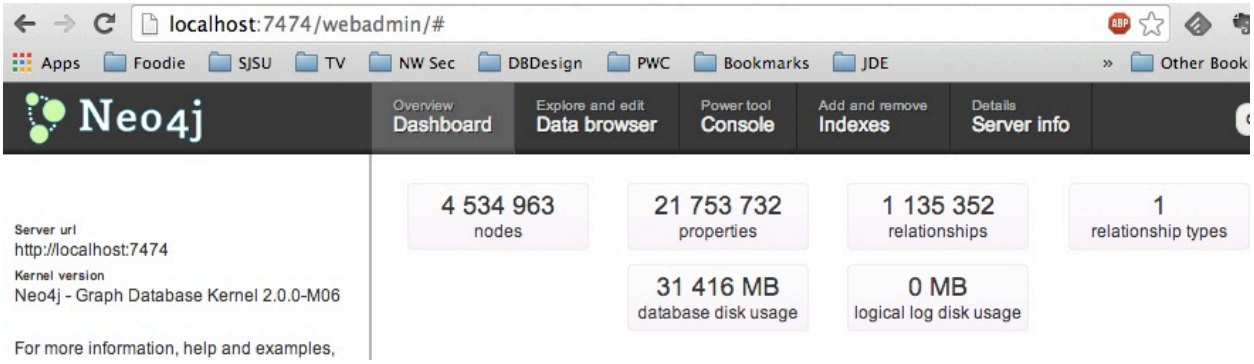
The initial approach of using individual inserts proved to be very slow. It took about 15 minutes to load a file with 7000 pages. Then we moved onto using the BatchInserter method. This enabled us to insert at very much faster pace. The batch inserter achieves this speed by doing away with transaction logs, at the cost of corrupting the database in case of a crash.

Keeping in mind that each page (node) was being indexed as it was inserted, an interesting thing that we noticed when loading the data was that, as more and more pages were being loaded, the insert time per page kept reducing.

Load Seq #	1	2	3	4	5	6
No of pages	7,126	11,502	30,590	51,338	2,204,522	2,330,440
Cumulative #	7,126	18,628	49,218	100,556	2,305,078	4,635,518
Time Taken (msec)	36,574	50,293	120,074	172,958	2,299,502	3,762,177
Time per page (msec)	5.132	2.699	2.852	2.111	1.019	0.829



The final count on our Neo4j instance was:

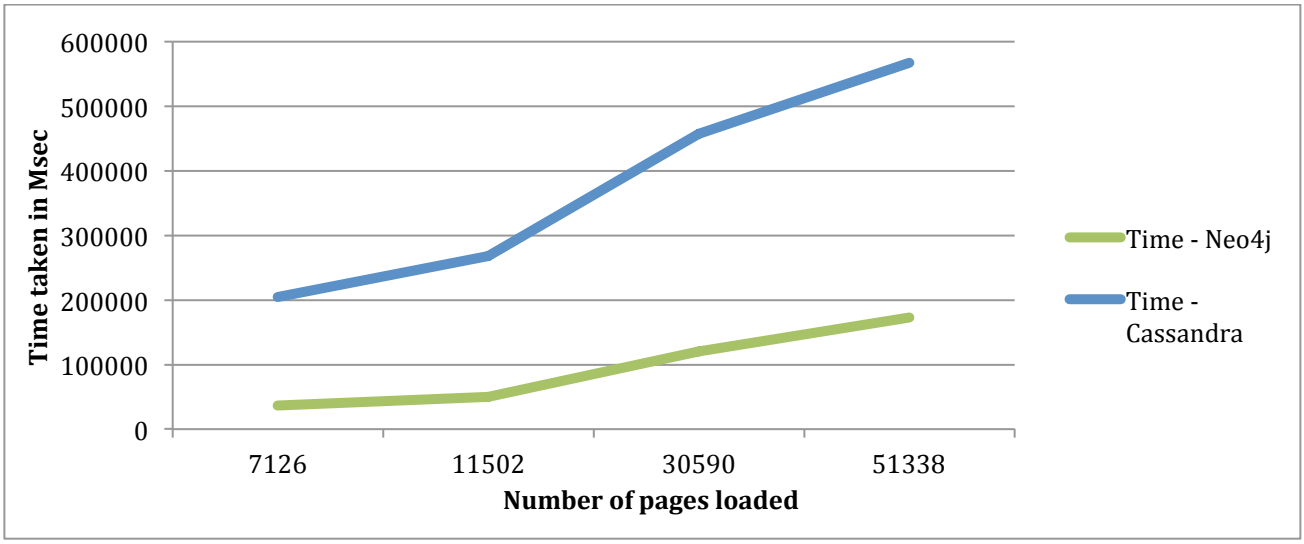


5.3.2 Cassandra

Our initial approach of loading the 6GB file was not successful as cassandra could not handle such large load while inserting into the database. So we often got java heap memory issue while inserting the pages into it. Thus, we built a small program which split the huge file into smaller chunks where each file consisted of 10,000 pages. This approach was very efficient.

Once we were able to split the files, we created a batch insert the data from each files. For this we created a small bash script which would automate the whole process and create a new java process for each file execution and kill the previous process.

```
#!/bin/bash
javac -d bin -sourcepath src -cp "lib/*" src/*.java
for dir in /Users/raul/Desktop/split1/*
do
    for file in "$dir"/*.txt
    do
     java -cp bin:lib/* ReadXMLUTF8FileSAX "$file"
     echo $file
     javapid=$!
     echo "$javapid"
     kill "$javapid"
    done
done
```



6. Visualizing Data using Neo4j

Since, our target was to browse Wikipedia visually, we thought of creating a web page where you could search for any Wikipedia page, and then it would represent that page and its link information visually. We also gave it the ability to traverse from page to page using these visual nodes.

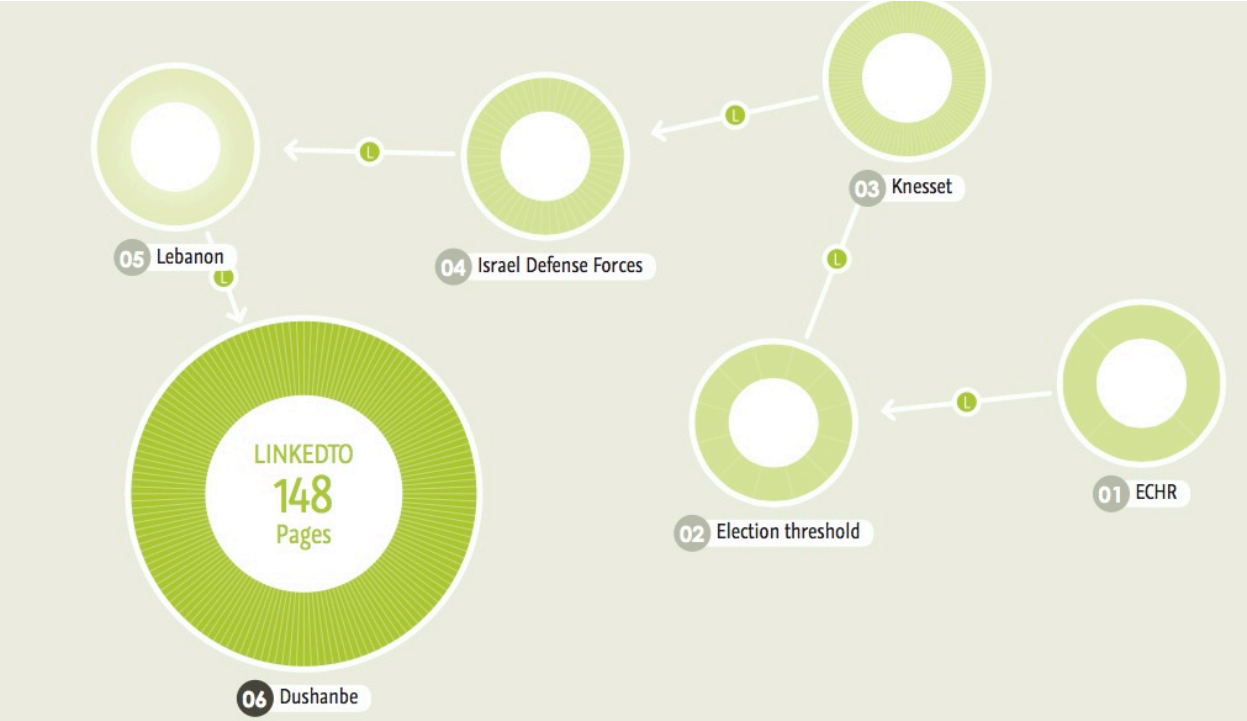
For this we used a combination of Ruby, Sinatra and Java scripting. The Neography & Neovigtor gem created by Max DeMarzi was used for connecting the ruby code to Neo4j. This gem abstracted the Neo4j REST API in a simple and efficient manner for our code.

Cypher queries were used to fetch data based on the page titles.

```
START me=node:Page(title='#{params[:id]}')
                  MATCH me -[r?]- related
                  RETURN me, r, related
```

This cypher query returns you all the nodes that the start node is connected to along with the relation ship information. The graph traversal is limited to a depth of 1, to reduce the amount of data returned. It also uses the "Page" index with a key of "title" to search for the nodes.

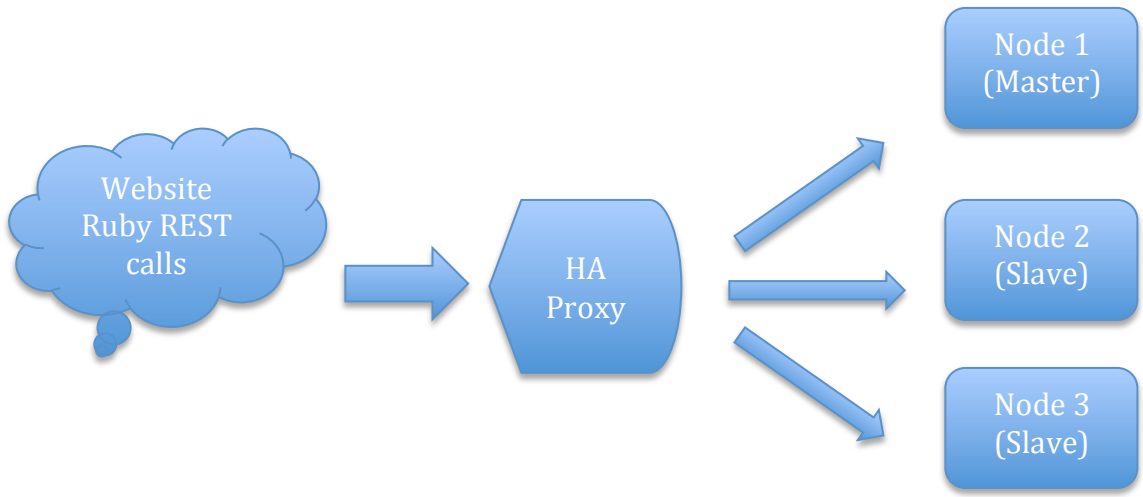
Processing.js was used to visualize and animate the results into the form shown in the below figure.



6.3 Neo4j High Availability

We also implemented HA for Neo4j using 3 nodes. The latest version of Neo4j uses Paxos for it's HA implementation. We used a manually defined master with a write threshold of 0 to simplify and speed up our inserts. A write threshold of 0 means that master will commit the transaction without waiting for the slaves to commit. The slaves will be made consistent eventually.

To load balance the read requests from the website we used the HA Proxy load balancer to handle and redirect all our REST requests. We set the balancing factor to be the page title so that the caches on each node can be used effectively in case of repeated queries.



```
frontend http-in
    bind *:80
    default_backend neo4j-slaves

backend neo4j-slaves
    # To use select nodes based on the title in the query
    balance url_param title
    #If requests are to be sent only to slaves
    #option httpchk GET /db/manage/server/ha/slave
    server s1 192.168.1.123:7474 maxconn 32
    server s2 192.168.1.11:7474 maxconn 32
    server s3 192.168.1.26:7474 maxconn 32
```

7. Lessons Learned

1. We learnt methods to handle and process huge amount of data, and process large files.
2. Neo4j has an extremely good write speed, but read speeds may be slow depending on the numbers of nodes and traversals done.
3. Neo4j is very easy to implement and scale as it uses a schema less design. Implementing indexes were also efficient due to the use of Lucene.
4. We found that based on our schema the read performance was very good when compared to the write performance of Cassandra. We implemented on a single node but Cassandra works well on a distributed environment.
5. Our batch insert of a single file which consists of 10,000 pages took around 1.5 minutes thus to load the whole dataset took us around 5 hours. When compared with Neo4J on which the 3000 nodes were inserted per sec which is extremely fast.
6. We also realized that a better way to realize our use case would have been to use both the databases simultaneously to leverage the best parts of both. For example, use Neo4j to store page id, page title and linkage information due to its high speed traversals, and store the page id and page text in Cassandra. We could then link them up using the page id field.

8. Current Limitations and Future Plans

Limitations

1. We are not handling the full set of Wikipedia pages. The 4 million pages that we loaded forms just under 6% of the total pages available.
2. We are facing issues loading data into Cassandra.

Future Plans

1. Implement graph traversals to discover relations between two nodes.
2. Scale up the solution to handle all Wikipedia searches.
3. Use Neo4j to link up Wikipedia pages to other sources of information.

4. Implement clustering in Cassandra.

9. How to load and execute the demo

9.1 Data Sets

1. The Wikipedia page information in the XML file.
2. The MySQL dump has to be loaded onto the local MySQL instance.

9.2 Neo4j

1. Add all the jars in the Neo4j/lib folder into the project.
2. Update the XML file location in the Neo4jBatchInserter.java file.
3. Run the batch inserter.
4. Update MySQL connection information in the Neo4jRelationBatchInsert.java
5. Run the Relation batch inserter

9.3 Cassandra

1. Start the Cassandra database as a root user -- ./bin/Cassandra -f
2. Add all the jars into your build–path which is present inside the lib folder of the project.
3. Pass the file name as a command line argument into the ReadXMLUTF8FileSAX.java.

```
javac -d bin -sourcepath src -cp "lib/*" src/*.java
java -cp bin:lib/* ReadXMLUTF8FileSAX --filepath
```

4. Run the program to insert the data.
5. Run the PageLinksUtil.java to insert the data into the wiki.links column family which would create the links for different pages in wikipedia

9.4 Running the Website

1. Copy the Wikigraph folder to your local machine

```
cd Wikigraph
bundle
rackup
```

The website will be available at http://localhost:9292

10. Individual Contribution