University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Computer Science	Spring semester, 2014 Restricted access
Writer: Stefan Andrew Aase	…… (Writer's signature)
Faculty supervisor: Tomasz Wiktor Wlodarczyk External supervisor(s): Odd Are Bjørkli	
Thesis title: **Adding Scalability and High Availability to Enterprise Performance Management Software.**	
Credits (ECTS): 30	
Key words: Distributed system, scalability, high availability, Apache Lucene, Neo4j, Elasticsearch	Pages: 45 + enclosure: 0 Stavanger, 12.06/2014 Date/year

Adding Scalability and High Availability to Enterprise Performance Management Software

Stefan Andrew Aase

Faculty of Science and Technology

University of Stavanger

June 2014

ABSTRACT

Corporater AS is a world leading provider of enterprise performance management software. The reason for businesses to use enterprise performance management systems so that the company can make better decisions based on the company's data.

This being the case, Corporater AS is experiencing an increasing demand for scalability and high availability. With the advantages of making Corporater EPM Suite a distributed system it would be possible to exploit the advantages of such a system.

The objective was to find a good solution, which makes Corporater EPM Suite less error prone and better suited for scalability than it is today. The solution must be fault tolerant while at the same time keeping the performance at an acceptable level for the end user.

This thesis creates two new implementations of Corporater EPM Suite. These two implementations replace the current Corporater EPM Suite using Lucene, with Elasticsearch and Neo4j. We have tested the performance of our two new implementations and compared them to the current Corporater EPM Suite using Lucene. By changing the architecture of the current Corporater EPM suite it is possible to create a scalable version by utilizing Elasticsearch or Neo4j.

ACKNOWLEDGEMENT

I would like to thank my supervisors Dr. Tomasz Wiktor Wlodarczyk and Odd Are Bjørkli for their valuable guidance. I would also like to thank Arne Rannestad and the development department at Corporater As for their support.

Last but not least, I would also like to thank my family and friends for their support through my thesis.

Stefan A. Aase
University of Stavanger

TABLE OF CONTENTS

ABBREVIATIONS

CRUD operations	-	Create, Read, Update, Delete operations
CS	-	Configuration Studio
EPM	-	Enterprise performance Management
JSON	-	JavaScript Object Notation
KPI	-	Key performance indicator
POC	-	Proof of concept
PK	-	Primary key
RAID	-	Redundant Array of Inexpensive disks

1 INTRODUCTION

Companies operate in different time zones so the companies' software solutions are in constant use. At the same time the leaders want to make the best possible decision. To do so they are dependent on their tools to help them make these decisions. What`s a good decision is found in the Company's business plan. From this the leadership of the company can decide what is best to achieve the company's strategic goals. Corporater EPM Suite helps them understand how their company is doing and can assist them in making the best possible decisions. Corporater is a growing company. Their existing customers are growing and they are getting increasingly larger companies as customers. Both of these groups have high demands for the performance of the software they use, which is the case with Corporater EPM Suite. Today Corporater EPM Suite is a single server application. It runs on one server and if the server under some circumstance should become unavailable, the application is not reachable until the server is available again. With this being the case it is not possible to horizontally scale out Corporater EPM Suite. It is also hard to ensure that Corporater EPM Suite is always up and running. With Corporater's customers located all over the world, doing business across countries, this is no longer acceptable.

As mentioned EPM Suite is a tool used to help the leaders in the company to make the correct decisions. For them to be able to do this Corporater EPM Suite must be configured to the company's needs. This is done by setting up the system to reflect the customer's goals. Based on their goals Strategic Initiatives, KPIs and business defined calculations can be set up. When this is done the correct people can be notified when an aspect of the company is not going as expected. How complex these functions are can vary and may include data for a large period of time. To be able to calculate them fast, the objects in the system must be indexed for fast searching and retrieval. It is possible users want to go back and look at how things were back in time. Because of this, the system may span over a very large period of time. There may also be a very detailed resolution of time in the system. Quarterly, days, hours, minutes. This makes it very expensive to pre calculate all the functions in the system.

With today's solution this is possible but the demands for scalability and high availability cannot be met. By creating a solution with the possibility to be scaled horizontally, these demands can be met and create a more stable Corporater EPM Suite. At the same time, it is important that the performance does not get worsened by the new functionality. The user experience is very important and must not be worsened by the new features. An optimal solution has the same or better performance than the current solution while, at the same time, adding the new required features to EPM Suite.

With this in mind, we have implemented two versions of EPM Suite. The first implementation is a distributed index to add better scalability to the system. This implementation uses Elasticsearch as the distributed index and replaces Apache Lucene which has been used in EPM Suite. The other implementation embeds a Neo4j graph database as the index in EPM Suite. By utilizing a graph as the index, we hope to gain performance by traversing the graph instead of searching. A drawback with the Neo4j graph is replication of data. Instead of sharding the data like Elasticsearch, Neo4j replicates the data over each instance in the cluster. This creates a solution with less scalability then the Elasticsearch implementation. With EPM Suite utilizing a distributed index, several application servers can speak to the index. This makes EPM Suite easier to scale out horizontally with the possibility of also having high availability. To verify the performance of the new index, we have compared the performance of our new implementations to the current Corporater EPM Suite with a Lucene index. To maintain the usability of the system, it is important that the final solution keeps the same performance as today's Corporater EPM Suite implementation.

2 RELATED WORK

The market provides many solutions for Enterprise Performance Management. A lot of them operate very similar to Corporater EPM Suite [1] [2]. They all take in data, execute calculations on them before showing them to the end user through a UI. The main difference between them and Corporater EPM Suite is how the metadata is created and when the calculations are done.

In other Enterprise Performance Management solutions, typically in data warehouses the metadata is created at design time. Data warehouse specialists model the data structure and create metadata based on this. Once the metadata has been created in the system, data from other systems can be imported and create objects based on the metadata. One of Corporater EPM Suites strengths and the reason they differ from other solutions is how this design time customization is not needed. Instead of having to create the data structures beforehand, design time, the metadata is already created. In Corporater EPM Suite the metadata objects like scorecard, perspective and KPI already exist in the system and the user can just go ahead and use them. It`s these kinds of objects that other solutions have to create before they can start. Runtime the user can setup the system as he/she likes, connect to the data sources and populate with data. With the metadata already created in the system, new aggregations can be added anytime runtime. There is also no need for data specialists. As the metadata is already available business intelligence consultants can setup the system.

Because of this difference it is hard for Corporater EPM Suite to pre aggregate the data like they do in e.g. data warehouses. As a result of this most of the calculations have to be executed runtime within the system. For this to happen in a timely manner Corporater utilize an index to retrieve the objects fast and minimize the time taken to do various calculations.

3 BACKGROUND

In this chapter we will first look at Corporater EPM Suits architecture. We will look at the different layers the application is built up of. After this we will go into depth of how Neo4j and Elasticsearch work. We will have a look at their architecture and how we can query them.

3.1 CORPORATER EPM SUITE

Corporater EPM Suite is an enterprise performance management software which helps the user execute their strategies. This is done by importing data from different data sources which is than evaluated in EPM Suite. How the data is evaluated is dependent on the user's goals and strategies and can be different from customer to customer.

3.1.1 Architecture

The architecture of EPM Suite is divided into three layers. Figure 1 shows these three layers. The top layer is the different clients that a user can use to interact with the system. Under this is the application layer that talks to the Lucene index and the Corpo store.

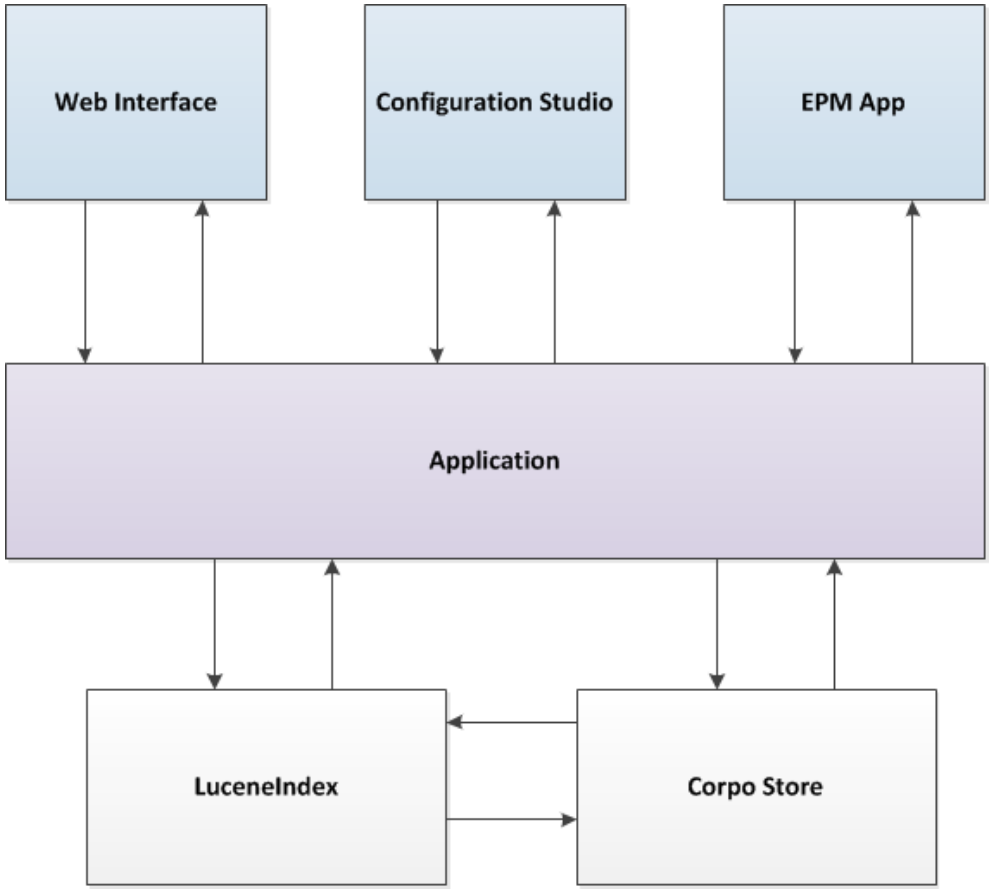

Figure 1 - EPM architecture

3.1.1.1 Clients

When using Corporater EPM Suite, the user has three ways of interacting with the system. The web interface, Configuration Studio or EPM App. The web interface is how the majority of user interact with the system and can be accessed from any web browser. From here the user can add, edit, or just look at the status of the system. Configuration studio, as the name dictates, is where the administrators of the system configure EPM Suite. The last way, EPM App, is a read only solution so that user can have a look at how everything is going from their iPhone or iPad.

3.1.1.2 Application layer

This is the server implementation of the system. Independent of which client the user is utilizing, they are all communicating to this layer. When a new business object is created in the web interface or in configuration studio, a request is sent to the server. From here, the application layer creates a commit pack that gets sent to the CorpoStore. A commit pack is an object that contains all the changes that have been done. Which property that has been changed, what value it has been changed to and on which objects

these changes has been done. If some objects have to be found later, the application layer uses the Lucene index to search for the relevant objects. Dependent on what kind of request the application layer has received, if the result from the Lucene index is not enough, the application layer can use the RIDs returned from Lucene to retrieve the complete object from the Corpo store.

3.1.1.3 CorpoStore

The CorpoStore is the non-volatile memory in Corporater EPM Suite. There are two types of Corpo stores that can be used, either embedded or an external database.

If the system is set up to use an external database, all objects will be stored in the database and only some information about it will be indexed in the LuceneIndex. When the application layer has changes to execute, it sends a commit pack to the CorpoStore. This will then send the necessary request to the database, when this has been completed, it will then update the index with these changes.

If the system is set up to use the embedded solution, the object itself is also stored in the index, together with the indexed information. The commit pack is sent straight to the LuceneIndex which indexes the information about it and stores the object itself in a separate field in the index.

3.1.1.4 Lucene Index

The backbone of Corporater EPM Suite is an Apache Lucene Index. The main purpose of the index is to search for different objects in the system but, as mentioned earlier it can also be used as an internal database. When the system is set up to use a relational database, it is only the most critical properties for an object that are added to the index. These are properties like name, rid, parent and more. The object itself is added to the database.

When a new object is created or an object property is edited, this change gets first sent to the CorpoStore to handle before the index receives the change. When a user requests something, the request first goes to the index. If it is just an index property that is being requested, the index does not ask the database for any information. If the index does not contain the requested information and the system is set up to use a database, it asks the database for this information. If the system is setup to use an embedded database, it obtains the object out of the index and obtains the requested information from there. As

this index contains references to all objects in the system, it is critical that the index is always up to date with the right references.

3.1.2 Apache Lucene

Apache Lucene is an open source project developed by the Apache Software foundation [3]. It is a search engine library which delivers good performance [4]. Lucene works by adding documents to the index which later can be retrieved by doing Lucene queries against the index. Each document has a number of fields. When querying the index, Lucene looks at the fields in a document and scores each field on its relevance to the query. Each score is then added together to get a score for the document. If the total score for an object is high enough, the document is added to the result for that query [5]. Lucene supports several different scoring methods that can easily be plugged into Lucene [3].

3.2 NEO4J

Neo4j is an open source graph database developed by Neo Technology Inc. [6]. With a graph database, you are able to store the data in the graph and retrieve data by querying the graph. With data that has a clear connection, like parent and child, there is a great advantage of using graph databases. With a graph database and data with such a nature, these connections also show up in the graph, making it easy to retrieve data by just querying the graph. Neo4j has several ways of running it. You can either run it on a single machine or distributing it over a cluster. It can also be run as either a service of its own or be embedded as a part of another application. Communication with Neo4j can be done either through a rest API or one of many language specific APIs for languages like JAVA, RUBY, PHP, SPRING and many more [7].

3.2.1 Architecture

The architecture of Neo4Js graph database is dependent on how the user chooses to set up the graph database. When Neo4j is run on a single server, this server contains the graph and handles all requests. If the server should fail under some circumstance, the graph will become unavailable. Neo4j can also be run in a cluster to add the possibility of high availability. When Neo4j is run in a cluster, the graph is replicated over to x

number of slaves. If the master should fail at some point, the graph is available at the slaves and one of them can take over as master.

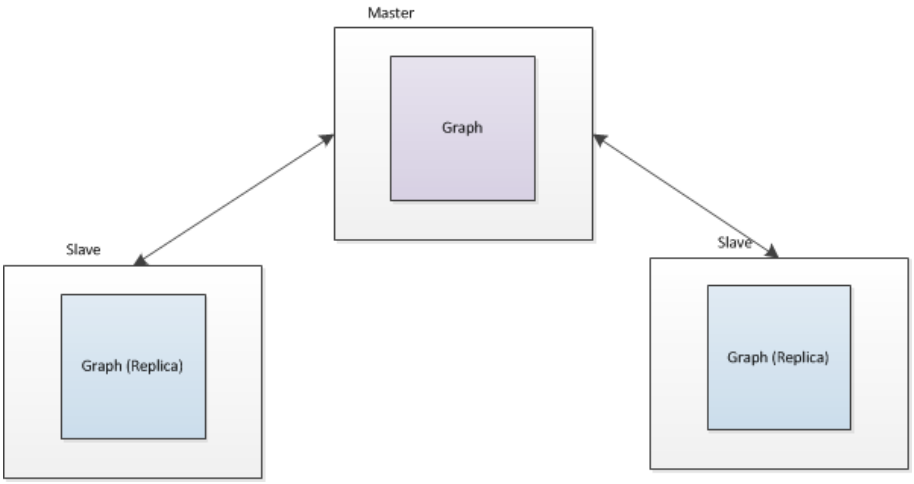

Figure 2 Neo4j replication

Figure 2 shows how the communication is between the master and the slaves. How often they communicate is decided by what type of communication it is (leader election, transactions) and how push/pull is configured for each node in the cluster. When a new transaction is committed, it is either pushed to the replicas or the replicas pulls the changes. Both of these variables can be configured. For example, if the cluster consists of three nodes, the master can push the change to one random slave and the other will pull changes at a given time interval. When a master receives a new commit he handles the commit and returns with a success or a failure. If a slave receives a new commit, it is sent to the master to handle. When this is done, the result is sent back to the slave who in turn returns it to the client. In leader election and agreement on a change Neo4j uses the paxos algorithm.

3.2.1.1 Paxos

Paxos [8] [9] is a consensus algorithm. Neo4j uses this algorithm for several things, cluster management, leader election and replication [10]. Each node in the paxos cluster has three components that are used, the proposer, acceptor and learner. The proposer's job is to propose values. The acceptor's job is to accept a value and lastly the learner is responsible for learning the value.

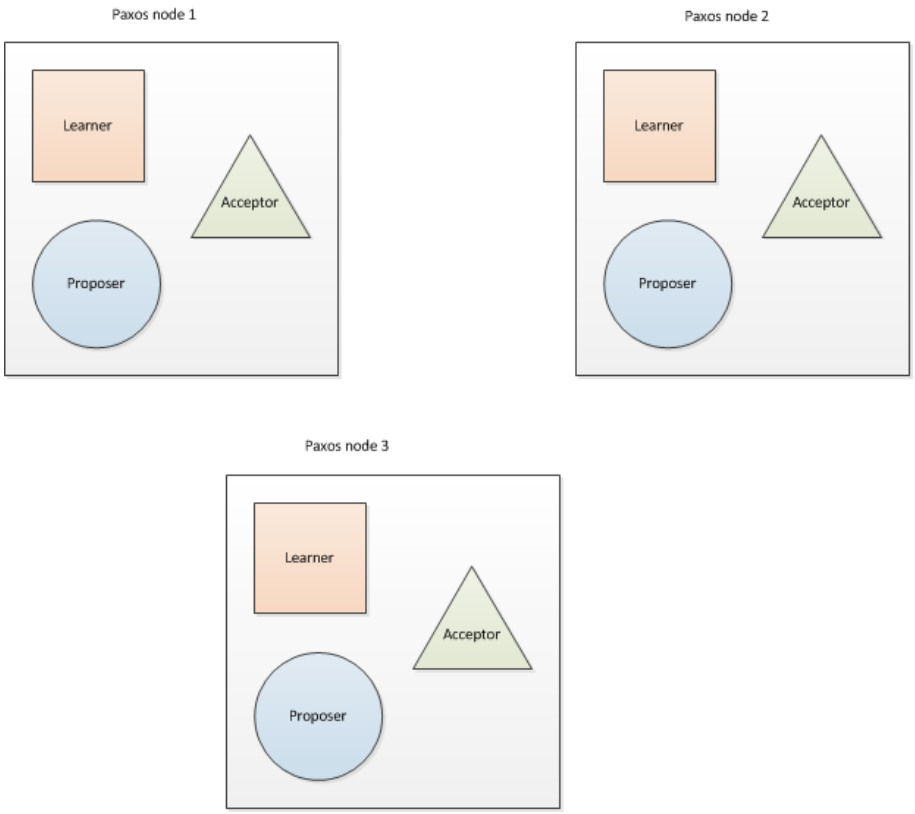

Figure 3 Paxos nodes

The proposer receives a propose message from someone. This can, for example, be a request for electing a new master. The proposer sends a prepare message to all the acceptors with this new value. If the acceptor receives a quorum of prepare messages, he promises to this value. If not, he sends a reject message.

$$n = number\ of\ nodes\ in\ cluster \quad (1)$$

$$quorum = \frac{2n}{2} + 1 \quad (2)$$

When an acceptor promises to a value, he is indicating that if he at some point in time, receives an accept message with this value he will accept it. When the proposer receives the promise message from the acceptor he sends an accept message in return. If the value in this message is equal to the one he has promised to he will return an accept message. If the proposer receives a quorum of accept messages, he will cancel the timeout and send a learn to the learner. Each new proposed value has an id. Because of this id, it is possible for paxos to handle messages that come in the wrong order and still

be able to create a result. As a result of this, when a learner receives a learn message he stores the value and checks the id of the message.

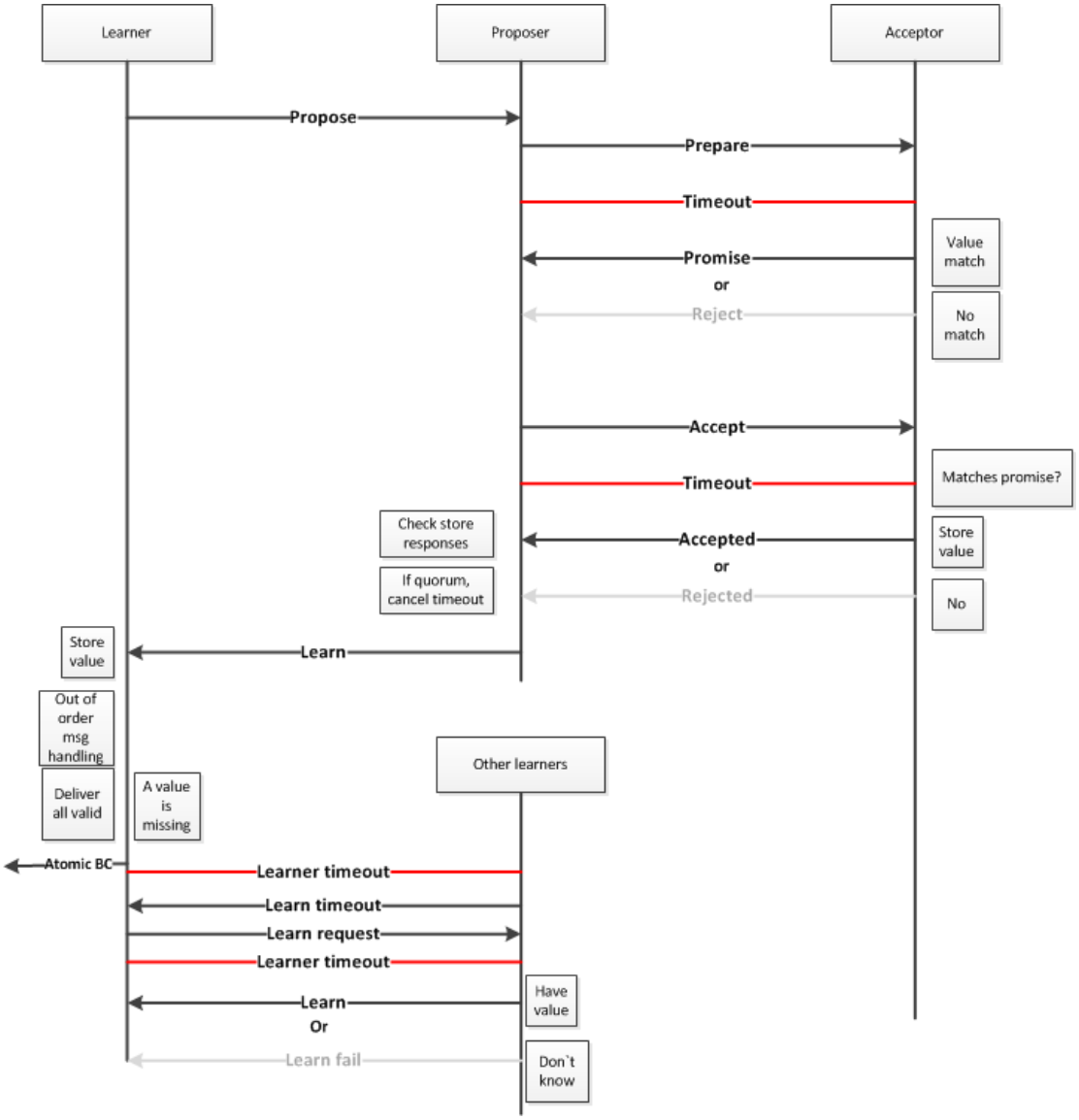

Figure 4 Paxos algorithm [11]

If the messages are in the correct order, the learner can deliver all the messages with an atomic broadcast. If not, the learner will simply wait to see if the missing messages are delayed. If the last learns have not been received by the time of the timeout, the learner will send a learn request to the other learners to see if any of them have learned the missing values. If any of the learners have the value he will send a learn with the missing value.

3.2.2 Query language

For querying the Neo4j graph there are two options. One can either do this using the Tinkerpop stack, more specific Tinkerpop Gremlin or Neo4j`s own Cypher queries. The Tinkerpop stack is a series of frameworks for working with graphs [12]. It has support for several types of graphs where Neo4j is one of them. Tinkerpop Gremlin is a graph traversal framework for traversing through the graph [12] [13].

The Neo4j Cypher query language is a query language written for the Neo4j graph database. When designing the query language they created it in such a way that it should be easy to write queries yet powerful. The query language has elements from other languages like SQL, SparQL and Python [14].

A cypher query is built up of two main components, MATCH and RETURN. It is also possible to add a WHERE clause to add more constraints on what is returned from the MATCH statement.

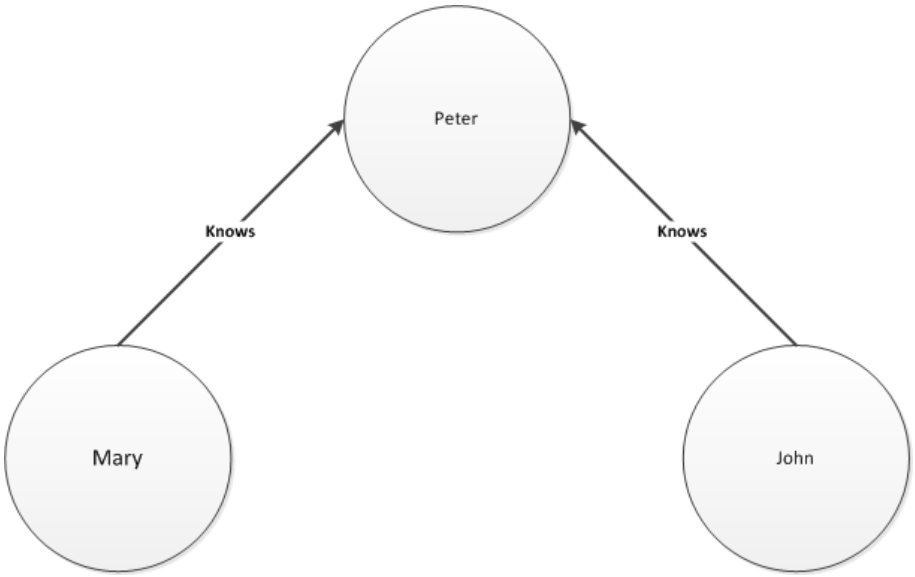

Figure 5 Simple graph

If we take Figure 5 as our graph, we can specify a query to retrieve everyone who "knows" Peter.

$$MATCH\ (n{:}\ Person) - [{:}\ KNOWS] -> (m{:}\ Person) \quad (3)$$
$$WHERE\ m.name = \text{“}Peter\text{”}$$
$$RETURN\ n$$

It is also possible to create vertexes and edges with the cypher query language.

$$CREATE\ (n\ \{name{:}\ \{\text{“}vertex\ 1\text{”}\}\}) \quad (4)$$

$$CREATE\ (n) - [r{:}\,KNOWS] -> (m) \quad (5)$$

Query 4 creates a vertex with the name “vertex 1”, while query 5 creates an edge *r* of type Knows from the vertex *n* to the vertex *m*. The arrowhead in the query determines the direction of the edge.

This is just a bit of what is possible with the cypher query language. It is also operators for removing edges, vertexes, mathematical operations, Boolean operations and more.

3.3 ELASTICSEARCH

Elasticsearch is an open source document store developed by Elasticsearch. It is built on top of Apache Lucene to extend Lucene to add features like clustering and high availability [15].

3.3.1 Architecture

Elasticsearch can be run alone or in a cluster. In both cases, Elasticsearch is run as a service beside the solution it is used in, but the configuration of Elasticsearch is a bit different. An Elasticsearch cluster consists of x number of instances in the cluster. From Figure 6 we can see how a cluster can look like. In this example the cluster consists of 3 Elasticsearch instances. Each cluster can contain several logical indexes which in turn can be divided into shards.

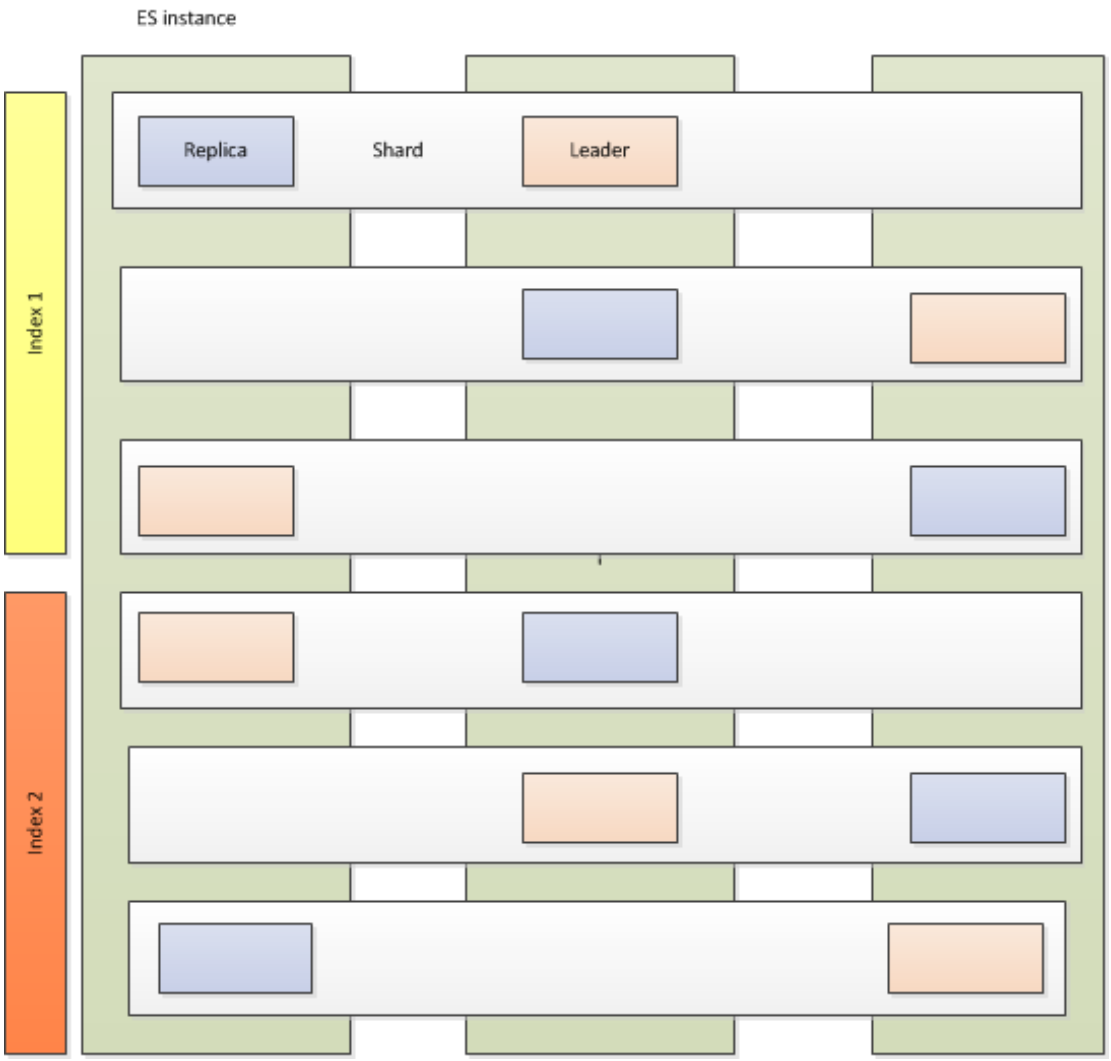

Figure 6 Elasticsearch cluster architecture

When looking at an index in Elasticsearch compared to Lucene, a complete Lucene index is one of the possible indexes in an Elasticsearch cluster. Each of these indexes can in turn be divided into shards. A shard is a part of the complete index. Each shard has a shard leader and x number of replicas. How many times the shard is replicated is configurable and up to the user. The same is the case for how many shards one index should be divided into. If we look at Figure 6 again we can see that the cluster contains two indexes, *Index 1* and *Index 2*. Each of these indexes are divided into 3 shards where each shard has one leader and one replication. Once an Elasticsearch cluster is set up the number of shards cannot be changed. An instance, on the other hand, can be added to the cluster at any time. When a new instance is added, the cluster will automatically re-allocate the shards.

3.3.2 Query language

For querying the Elasticsearch cluster one can either use the query DSL provided by Elasticsearch or one of the many APIs [16]. Elasticsearch provides APIs for JAVA, JavaScript, PHP and more [15] [17]. When using the provided APIs, you connect to the cluster by creating a client object. This object can then do CRUD operations on the cluster, as well as bulk operations.

If we take the Java APIs as an example, we can see how different operations are done through the API. When inserting an object into the index, every object is sent in JSON format. This conversion can either be done manually, convert the object into a map or use Elasticsearch JSON builder to convert the object.

For getting/deleting an object you simple create a *prepareGet*/*Delete* with the index name, object type and object id. This will either get or delete the object. By looking at the response it is possible to retrieve the object or see if the delete was successful.

The advantage of building on Lucene is the search capabilities that Lucene offer. Searches in Elasticsearch are able to interpret Lucene queries and these can therefore just be passed straight into Elasticsearch.

4 DESIGN AND IMPLEMENTATION

As we mentioned earlier, the backbone of EPM Suite is the main Lucene index. For the system to operate as expected, this must be up to date at all times. The Lucene layer in EPM Suite is today embedded within the server. To add scalability and high availability to EPM Suite, it is possible to replace Lucene with either Neo4j or Elasticsearch. We have made one implementation of each to see which gives the best performance with the added features of scalability and high availability. The first is a distributed Lucene index achieved by using Elasticsearch. The second is replacing the Lucene index with a Neo4j graph database and store the data in the graph. In this chapter we will have an in depth look at these two implementations. We will first have a look at the Elasticsearch implementation covering how it has been implemented into EPM Suite, querying Elasticsearch and lastly the communication between EPM Suite and Elasticsearch. After this we will cover the Neo4j implementation covering the same topics as with the Elasticsearch implementation.

Both implementations implement some base classes that EPM expects to be present. Since Elasticsearch and Neo4j have different ways of communications and queries this results in separate classes for both implementations. We will have a look at how both of these implementations are built up together with how they both communicate with EPM.

4.1 ELASTICSEARCH IMPLEMENTATION IN EPM

The Elasticsearch implementation into EPM consists of two base classes, ESCorpoStore and ESCorpoIndex. The first is the Elasticsearch instance of the CorpoStore, secondly is the Elasticsearch instance of the CorpoIndex. There has also been made two helper classes, one for converting system objects into documents for the index and one helper class for creating queries.

4.1.1 ESCorpoStore

The ESCorpoStore has the two method that all CorpoStores must have, add and getObject. The add method receives a commit pack from the application and adds this to the store. The getObject method is responsible for retrieving an object from the ESCorpoStore. When using the ESCorpoStore the location of storage is actually just the index itself. So all objects get stored in the index. If we look at figure 7 at line 157 we see the *storeData* variable. When this is true it indicates that the ESCorpoIndex is also used as storage and the data should also be stored in the index. The same is the case for the get method in the ESCorpoStore. When retrieving the object we go straight to the index and retrieve the object from the index.

4.1.2 ESCorpoIndex

The ESCorpoIndex is the distributed Elasticsearch index that replaces the Lucene index. It has the same functionality as the old Lucene index but it communicates with the Elasticsearch cluster instead of an embedded Lucene index. There are two main methods in the ESCorpoIndex, one for adding new elements to the index and one for extracting information out of the index.

```
146         BulkRequestBuilder bulkRequest = client.prepareBulk();
147
148         for (Rid rid : delKeys) {
149             bulkRequest.add(client.prepareDelete("index", "object", rid.toString()));
150         }
151
152         for (Map<String, Object> document : toAdd) {
153             bulkRequest.add(client.prepareIndex("index", "object", document.get(CorpoIndex.PK)
154                     .toString()).setSource(document));
155         }
156
157         if (storeData) {
158             for (Map.Entry<PropertyHandle, byte[]> entry : dirtyBlobs.entrySet()) {
159                 if (entry.getValue() == null) {
160                     continue;
161                 }
162                 Map<String, Object> objectMap = ESDocumentHelper.produceBlobPropertyDocument(
163                         entry.getKey(), entry.getValue());
164                 if(objectMap.get(CorpoIndex.PK) != null){
165                     bulkRequest.add(client.prepareIndex("index","object", objectMap.get(
166                             CorpoIndex.PK).toString()).setSource(objectMap));
167                 }else {
168                     bulkRequest.add(client.prepareIndex("index","object", objectMap.get(
169                             CorpoIndex.BLOB_PK_REF).toString()).setSource(objectMap));
170                 }
171             }
172         }
173         BulkResponse bulkResponses = bulkRequest.setRefresh(true).execute().actionGet();
```

Figure 7 ESCorpoIndex Update Index

In figure 7 we have the method that sends the changes to the index. Elasticsearch can handle bulk request to the index. In our update method, this is just what we are doing. We start by creating a bulk request before we go through the changes in the commit and add the changes to the bulk request. Once the bulk request contains all the changes, we go ahead and execute these. On line 171 we can see the execution of the bulk request returning us a *bulkRespons*. From the *bulkRespons* we can check if the changes where successfully executed on the index.

Elasticsearch can also handle single requests. Instead of adding the client request to the bulk request one would simply create an add or delete request and execute this single request. Just as the bulk request, these requests also return responses that can be used to check if the request was executed successfully.

```
122     <O> O extract(ESIndexQuery indexQuery, Aggregator<IndexEntry, O> aggregator) {
123         Query query = indexQuery.generate();
124         String queryString = query.toString();
125         SearchRequestBuilder searchRequest = client.prepareSearch().setQuery
126                 (QueryBuilders.queryString(queryString)).setSize(500000);
127         long sTime = System.currentTimeMillis();
128         SearchResponse searchResponse = searchRequest.execute().actionGet();
129         long eTime = System.currentTimeMillis();
130         LOG.log(Level.FINEST, "Query time {0} = {1}", new Object[]{new TimeDuration
131                 (sTime, eTime), queryString});
132         SearchHit[] hits = searchResponse.getHits().hits();
133         for(SearchHit hit : hits){
134             ESEntry entry = new ESEntry(client);
135             entry.setDoc(Long.valueOf(hit.getId()));
136             aggregator.add(entry);
137         }
```

Figure 8 ESCorpoIndex Extract method

Since Elasticsearch supports Lucene queries, we can use the already existing Lucene queries in the system. If we look at figure 8, we see that all that has to be done is create a search request and set the query string. In our case the query string is a generated Lucene query. Once the search request has a query string, we can execute it and the search result will contain all the results for that query. It is then possible to iterate through all the hits in the response to convert the hit into an object in the system.

4.1.3 Object to document conversion

To increase maintainability and readability of the code we have chosen to create a helper class for converting objects to Elasticsearch documents. The main method of this class is a process method that takes in a *StoreObject* and converts this into a map representation which in turn can be sent to the Elasticsearch index for indexing.

All model objects in the system extend an object called *StoreObject*. This is the base class and contains all the information about the object that we like to index.

The process method in turn calls a set of methods which returns a map with keys and values for the given *StoreObject*. Each method returning a map for a given part of the object.

```
167     private static final TypeProcessor<StoreObject, Map<String, Object>> FIELDPROC_AUDIT
168             = new TypeProcessor<StoreObject, Map<String, Object>>() {
169
170         @Override
171         public Map<String, Object> process(StoreObject in) throws DataException {
172             Map<String, Object> map = new HashMap<>();
173             map.put(CorpoIndex.LAST_MODIFIED_TIME_FIELD, in.getModified().getTime());
174             map.put(CorpoIndex.LAST_MODIFIED_BY_FIELD, in.getModified().getUser());
175             map.put(CorpoIndex.CREATED_TIME_FIELD, in.getCreated().getTime());
176             map.put(CorpoIndex.CREATED_BY_FIELD, in.getCreated().getUser());
177             return map;
178         }
179     };
```

Figure 9 ESDocumentHelper audit info

All the methods are of the same format as the FIELDPROC_AUDIT method shown in Figure 9. Each method takes in a *StoreObject*, creates a map and adds different info about this object to the map. All of these maps are added to one large map, which represents the object in document form.

4.1.4 Query generation

As with the conversion of objects, we have also made a helper class for creating Lucene queries to query the Elasticsearch cluster. By creating an ESIndexQuery object, one can use this object to create a query which can create a Lucene query from the given parameters. ESIndexQuery has methods for querying after text, strings, longs and ranges. Dependent of method, it takes in either a string or a list of string fields and the same for values. The field values are the fields in the index that we wish to compare our values too. If it is a list of fields we compare the values to all the fields, if it is the other way around we compare all the values to the given field. For text queries, we also

support wildcard searches. If a given text includes * or ? we create a wildcard search for the given field with the given value.

4.1.5 Elasticsearch service

Elasticsearch is a schema less document store meaning that it will create the schema based on the documents put into the index. Connecting together instances is also very easy as they find each other and create the cluster. Altogether very little setup is needed for the cluster to get up and running. How many shards the index should be divided into and how many times each shard should be replicated are the most important settings to configure.

4.1.6 Communication between EPM and Elasticsearch

Since Elasticsearch is run outside of EPM and not embedded like Lucene, we have to create a client object which we use to communicate with the Elasticsearch cluster. This is a client object provided by the Elasticsearch Java API.

```
59      public ESCorpoIndex(final SeqFile directory, final CorpoDataDir.WorkspaceDir wsdir
60      ) throws IOException {
61          super(wsdir);
62          DbConfig conf = wsdir.getDbConfig();
63          this.storeData = conf.getType().isDataStoredInIndex();
64          Settings settings = ImmutableSettings.builder()
65                  .put("discovery.zen.ping.multicast.enabled", "false")
66                  .put("discovery.zen.ping.unicast.hosts", "10.10.10.222:9300," +
67                          "10.10.10.65:9300,10.10.10.67:9300,10.10.10.174:9300," +
68                          "10.10.10.176:9300")
69                  .put("cluster.name", "testes")
70                  .build();
71          node = nodeBuilder().client(true).loadConfigSettings(false).settings(settings).node();
72          helper = new ESDocumentHelper(this.storeData);
73          client = node.client();
74          boolean indexExists = client.admin().indices().exists(new IndicesExistsRequest(indexName))
75                  .actionGet().isExists();
76          if(!indexExists){
77              CreateIndexRequestBuilder createIndexRequestBuilder = client.admin()
78                      .indices().prepareCreate(indexName);
79              createIndexRequestBuilder.execute().actionGet();
80          }
81          client.admin().cluster().prepareHealth().setWaitForGreenStatus().execute().actionGet();
82      }
```

Figure 10 ESCorpoIndex constructor

If we look at line 64 in figure 10, we can see that we first create a Settings object. This object contains all the settings we want the node to have. In our case, we have set the cluster to not use multicast and providing the locations of where the unicast hosts can be located. We are also providing the name of the cluster we would like to connect to. At line 74 we use a nodeBuilder provided by Elasticsearch to create a node which will communicate with the cluster. By calling the *client(true)* method on the nodeBuilder we

are telling Elasticsearch that this node will only be used as a client and will not hold any data. Once we have created the node we can use this node to give us a client object, which will be our communication point into the cluster. When using the client in this manner we have actually created a new node in the cluster that only handles client request. It is also possible to create a transport client, this will not get added into the cluster but communicates more from outside of the cluster. It implements a very simple load balancer by connecting to each node in the cluster in a round robin manner.

The positive side of connecting to the cluster with the client from the node, the request is sent to the node that should handle that request. This limits how many hops need to be done in the cluster to fulfil the request.

4.2 NEO4J IMPLEMENTATION IN EPM

Since both the Elasticsearch and Neo4j implementations implement the same base classes, both of the implementations have a lot of similarities. As a result of this, the Neo4j implementation is a lot like the Elasticsearch implementation but with some variations needed for Neo4j. The major difference between the Neo4j implementation and Elasticsearch implementation is that Neo4j is embedded into EPM instead of running beside EPM as a separate service.

4.2.1 Communication between EPM and Neo4j

Since Neo4j is embedded within EPM, we can connect to the graph without going over the network. Each EPM instance that starts up will have a Neo4J graph instance embedded into it. When the graph is created, it takes in a properties file that contains all the configuration information the graph needs. There are a lot of properties that can be set but the required properties are node id and host addresses. The first is the id of that node in the cluster. Each node in the cluster needs to have a unique id in the cluster. The second are the addresses of all the nodes in the cluster, including the node itself.

```
graphDB = new HighlyAvailableGraphDatabaseFactory()
        .newHighlyAvailableDatabaseBuilder(this.directory.getAbsolutePath())
        .loadPropertiesFromFile(configPath)
        .setConfig(GraphDatabaseSettings.node_keys_indexable,
                CorpoIndex.PK + "," + CorpoIndex.ALIAS_ID +
                        "," + CorpoIndex.ALIAS_SPACE +
                        "," + CorpoIndex.MODEL_FIELD +
                        "," + CorpoIndex.TYPE_FIELD +
                        "," + CorpoIndex.LINKEDTO_FIELD)
        .setConfig(GraphDatabaseSettings.node_auto_indexing, "true")
        .newGraphDatabase();
```

Figure 11 Neo4j graph construction

If we look at figure 11, we can see the construction of the Neo4j graph. The constructor takes in a file path where the graph is stored. Furthermore, it loads in the properties file and we configure some indexes we will need.

4.2.2 Neo4jCorpoStore

The Neo4j implementation of the CorpoStore also has the two base methods, add and get. The add method calls the add method of the index. We will cover this method in the next chapter. The get queries the graph directly for a given object.

```
index.extract(
        "MATCH (n:" + EPM_OBJECT + "{" + CorpoIndex.PK + ": {pk}}) " +
        "RETURN n", SeqFactory.<String, Object> map("pk", key.toLong()),
        agg);
```

Figure 12 Neo4jCorpoStore extract

If we look at figure 12 we can see the main part of the get method in the Neo4jCorpoStore class. Figure 12 shows the query used to get a given object out of the graph. We look for an EPM_OBJECT in the graph with the given pk. In return we are given the node we are searching for.

4.2.3 Neo4jCorpoIndex

The Neo4jCorpoIndex has the same two methods as the Elasticsearch implementation with some modifications to handle the graph. The extract method takes in a string query which is sent to the graph. This query is a cypher query which we generate based on the operation that we want to execute. How these queries are generated will be covered in the next chapter.

```
469     <RowType, O> O _extract(final String query, final Map<String, Object> params
470             , final Aggregator<RowType, O> aggregator) {
471         executeInTransaction(new InTransaction<Void>() {
472
473             @Override
474             public Void call(Transaction tx) {
475                 String cypherQuery = query;
476                 try {
477                     ExecutionResult rs;
478                     rs = engine.execute(cypherQuery, params);
479                     for (RowType item : asIterable(rs.<RowType> columnAs("n"))) {
480                         aggregator.add(item);
481                     }
482                     return null;
483                 } catch (CypherException e) {
484                     throw new RuntimeException(e);
485                 }
486             }
487         });
488
489         return aggregator.finish();
490     }
```

Figure 13 Neo4jCorpoindex extract

Once we have the generated cypher query we can execute this on the graph to get a *ResultSet* in return. Once we have the *ResultSet,* it is a simple job of converting this into objects.

As with the extract method, the add method is also a lot like the Elasticsearch implementation. Once the commit pack is received, each change is handled and added to the graph. How exactly, will be covered in more detail later.

```
263     @Override
264     public boolean _add(final CommitPack pack) {
265         boolean res = executeInTransaction(new InTransaction<Boolean>() {
266             @Override
267             public Boolean call(Transaction tx) {
268                 long start = System.currentTimeMillis();
269                 Neo4jDocumentHelper helper = new Neo4jDocumentHelper(Neo4jCorpoIndex.this);
270                 for (StoreObject so : pack.getAdded().values()) {
271                     helper.processNew(so);
272                 }
273
274                 for (StoreObject so : pack.getAdded().values()) {
275                     helper.process(so);
276                 }
277
278                 for (StoreObject so : pack.getChanged().values()) {
279                     helper.process(so);
280                 }
281
282                 if (storeData) {
283                     for (Map.Entry<PropertyHandle, byte[]> entry : pack.getBlobs().entrySet()) {
284                         if (entry.getValue() == null) {
285                             continue;
286                         }
287                         PropertyHandle key = entry.getKey();
288                         helper.produceBlobPropertyDocument(key, entry.getValue());
289                     }
290                 }
291                 helper.executeQueue();
292
293                 for (Map.Entry<Rid, ? extends StoreHeader> entry : pack.getRemoved().entrySet()) {
294                     StoreHeader so = entry.getValue();
295                     try {
296                         Node n = findByRid(so.getRid());
297                         Set<Node> seen = new HashSet<>();
298                         seen.add(n);
299                         deleteNode(n, seen);
300                     } catch(NodeNotFoundException e) {}
301                 }
302                 long end = System.currentTimeMillis();
303                 return true;
304             }
305         });
306         return res;
307     }
```

Figure 14 Neo4jCorpoIndex add

4.2.4 Query generation

For generating Cypher queries we have created a helper class. The helper class Neo4jIndexQuery contains several different methods for creating sub queries dependent on what we would like to achieve. Once all the sub queries are called we can call generate on the Neo4jIndexQuery object which will in turn create a Cypher query from our partial queries.

4.2.5 Object to graph conversion

For converting objects to the graph we have made a helper class. This is the helper object that can be seen from line 275 in Figure 14. It has some of the same functionality

as the Elasticsearch implementations helper class but instead of creating a map representation of the object, it creates vertexes and edges in the graph to represent the given object.

For the purpose of demonstration, say we have a KPI object with seven properties, two of which have multiple values for each of the properties. In the Lucene and Elasticsearch implementations each document will have seven properties each, with one value for that property. If it is one of the two properties with multiple values it will just contain a list and for the other a single value. In the Neo4j implementation this is a bit different. In Lucene and Elasticsearch, a document represents a single object. In Neo4j, a vertex is created to represent that single object. For each property an edge is created to a new vertex representing the property and its value. So if we go back to the KPI object giving it has these properties;

Name
Description
Status
Trend
Rid
Name_Localized
Description_Localized

Table 1 KPI object properties

The graph for the KPI object will look as follows.

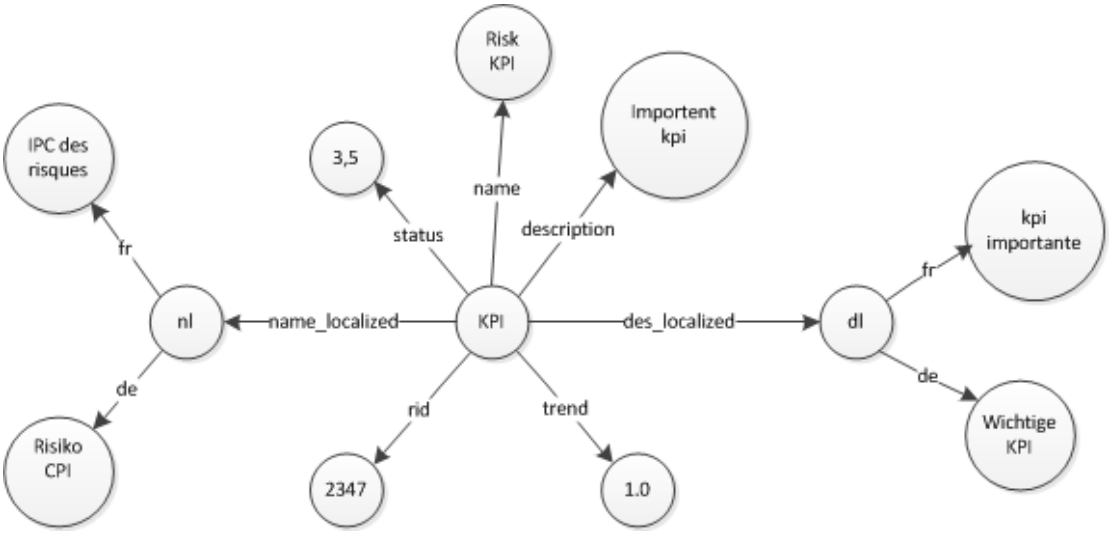

Figure 15 Example KPI graph

The properties and values in the graph are not important here. It is only for illustrative purposes.

If we look at Figure 15 we can see how the graph for the given KPI object will become. The conversion from object to graph elements can be expressed as;

$$edges = xp + \sum_{y-1}^{o} mp \tag{6}$$

$$vertises = edges + 1 \tag{7}$$

(6) expresses how many edges will get created based on the properties. The first addend *xp* is the number of single value properties. The second addend is the sum of all the multi value properties.

The first addend of (7) is (6). It is simply because for each edge there is a corresponding vertex. The last addend in (7) is the vertex that represents the object itself.

5 RESULTS AND EVALUATION

In this chapter we will test our two new implementations of EPM Suite. We will compare the performance of each implementation against Corporater EPM Suite with a Lucene Index. We will conduct tests that focus on both the read and write performance of each implementation. Lastly we will test the implementations with larger amount of data too see how they perform with large amounts of data in the system.

5.1 EXPERIMENT SETUP

When testing, 3 setups where used, one for Lucene, one for Neo4j and one for Elasticsearch. The Lucene implementation was running on one server connected with an SQL server. The Elasticsearch implementation had EPM running on the same server as with Lucene, an SQL server connected and 5 Elasticsearch instances to create the Elasticsearch cluster. Lastly the Neo4j implementation ran on the same server as earlier with an additional 4 servers to create the cluster.

For all of the tests, EPM Suite was run on a Windows 7 computer with 16GB RAM, Intel Xeon E31270 processor running at 3,4 GHz and 240 GB solid state drive.

Three of the nodes in the cluster consisted of virtual servers all running Windows Server 2008 R2. The physical servers that was running these three virtual servers has an Intel i7 3820 running at 3,67 GHz, 64GB RAM and a configurable three layer disk solution. The disk solution is set up with a RAM cache, and a SSD cache with a 10.000rpm hard drive as the last layer.

The last two are Windows Server 2003 running on in VMWare. The physical machine running VMWare had an Intel Xeon E5520 running at 2.27GHz with 128GB of RAM. For storage the server used a SAN. The SAN is a Dell Power Vault MD3600i set up in RAID 5 with 8 x 600GB HDD.

The SQL server used is the same for all tests. The SQL server has been run on a physical server equal to the three first virtual servers. The SQL version used is 2008 R2.

5.2 WRITE RESULTS

For testing write performance, we have done tests with different loads to see how each implementation write performance is. The first test is to add different types of objects to the system. Our experience is that adding objects to the system takes about the same time independent of object type. Because of this, we have only included adding a scorecard in the test results.

If we look at figure 16 we can see how long time each implementation took to add one single scorecard to the system. In the charts Lucene is listed as one node. This is actually the standard implementation of EPM Suite with Lucene embedded. This is the same for all tests.

We have started our testing with just one node in each cluster. This does not actually give us any of the desired features but it shows us how a very simple setup of each implementation performs against Lucene.

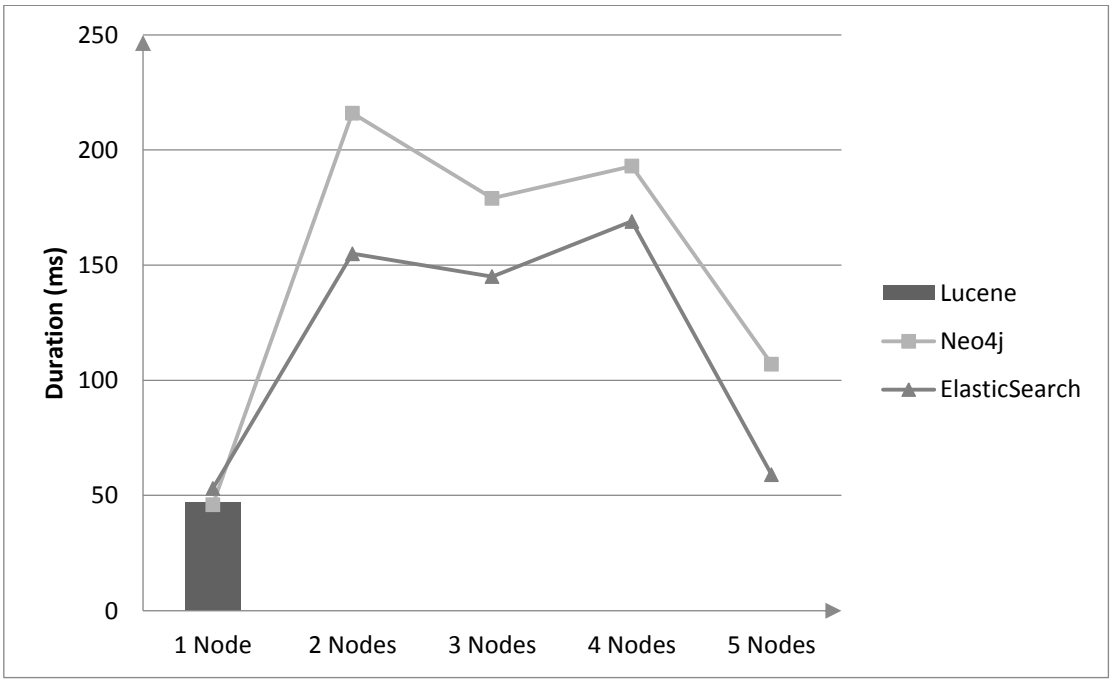

Figure 16 Add scorecard to EPM

From figure 16 we can see that each implementation with one node performs very similarly compared to the stock EPM implementation. If we keep adding nodes we see that the performance degrades quite a lot before is goes back up. The Elasticsearch implementation is almost at the same level of the stock EPM implementation with Lucene. Neo4js performance comes close but is still about 50ms slower than Lucene.

The first test only added one object to the index. To see how the index copes with larger write operations we create some dummy values which get added to the index. We achieved this by creating some node data. When importing data into the system this is attached to an object called node. This node can represent any data and any type of data (actual, budget, trend, etc). The node is than associated to an organization in the system. For our data generation we generated node data for 5 organizations, 5 nodes and 4 node types. This results in a much larger amount of data getting added to the index than just adding one scorecard, creating a larger load on the index.

If we look at figure 17 we can see how each implementation coped with value generation. Again, Lucene is the fastest to complete, only taking 1,3 seconds to complete. Both Neo4j and Elasticsearch take over two seconds more to complete the same action. When adding more nodes to the cluster Elasticsearch performance stays within the 3-4 second range before increasing after 4 nodes. Neo4js performance on the other hand degrades drastically with only 2 nodes in the cluster, taking over 6 seconds to complete the same action. This is the case until 5 nodes are in the Neo4j cluster when the performance of Neo4j comes back to the same level as Elasticsearch.

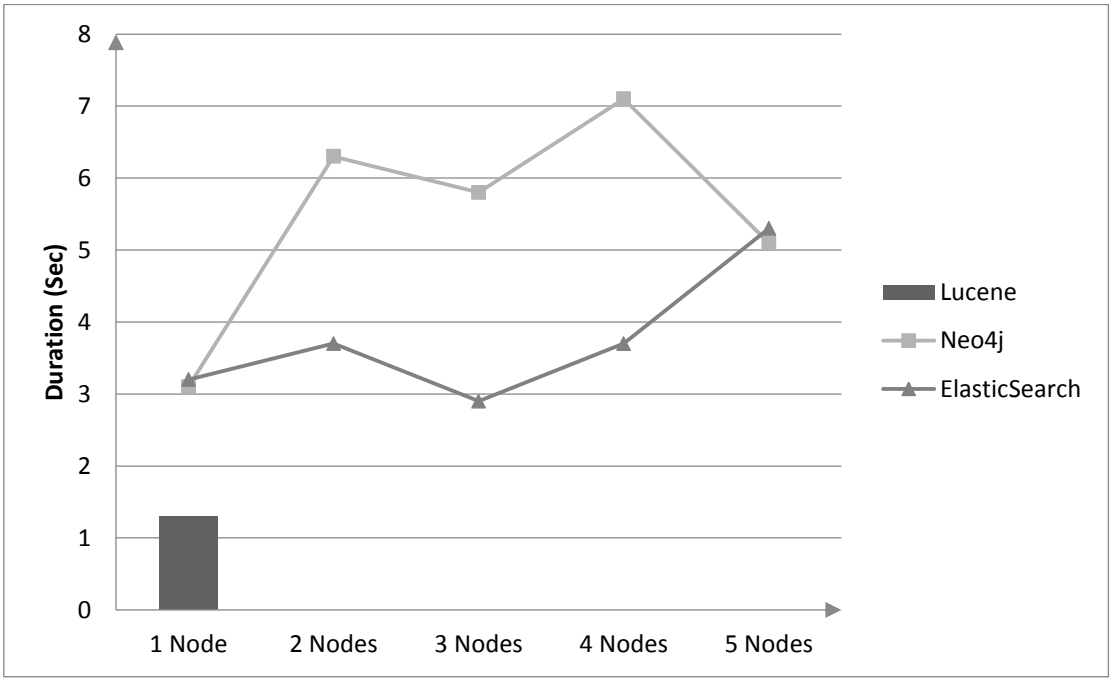

Figure 17 Generate node data

The last write test we performed was restoring a medium database into the system. This is a write heavy operation as the whole backup needs to be added to the system

and indexed. Unlike the previous write test this is a test that should take longer time to complete. This will give us an indication of how the system performs with larger writes over time. From figure 18 we can see the performance of each implementation. Here Neo4j starts off well only taking 46 seconds to complete the restore. Lucene completes the same operation in 38 seconds. Compared to these two, Elasticsearch used almost double the time to complete. More precisely, 96 seconds to restore the database. As we increase the size of the cluster Neo4js performance degrades before it comes back up as we have seen on earlier tests.

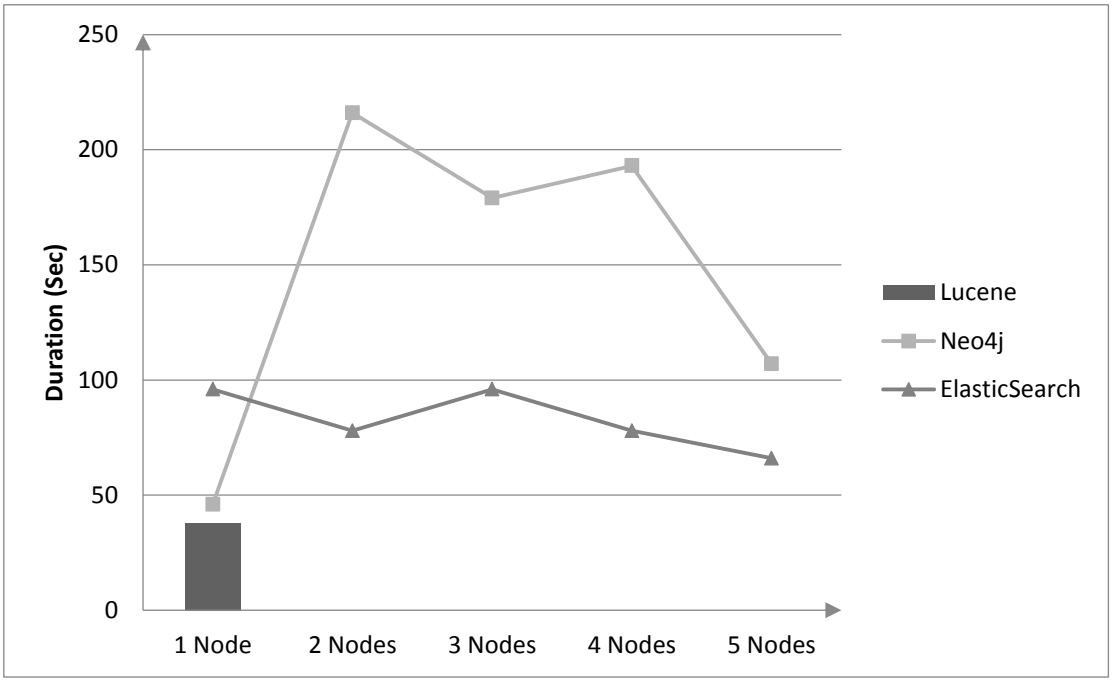

Figure 18 Restore medium db

Elasticsearchs` performance is more stable when adding more nodes to the cluster. Compared to Neo4j, Elasticsearchs` performance increase as the cluster grows larger. This is a bit interesting as we saw the opposite in the node data generation test.

5.3 READ RESULTS

The tests we have done so far have been write dominated. As both CS and the web client use a great deal of reads, it is also important that the read performance is adequate. To test this we have generated a report on all of the implementations. A report can consist of references to different objects in the system and also different functions. To perform the calculations and locate the objects the index is used to find them so we can generate the report. For our test report it finds all the KPIS in the organization structure in the

system and creates a PDF report with the names of all the KPIs. If we look at figure 19, we can see how long each implementation took to generate the given report. From the figure we can see that the Elasticsearch implementation completed the report generation faster than EPM Suite with Lucene and the Neo4j implementation.

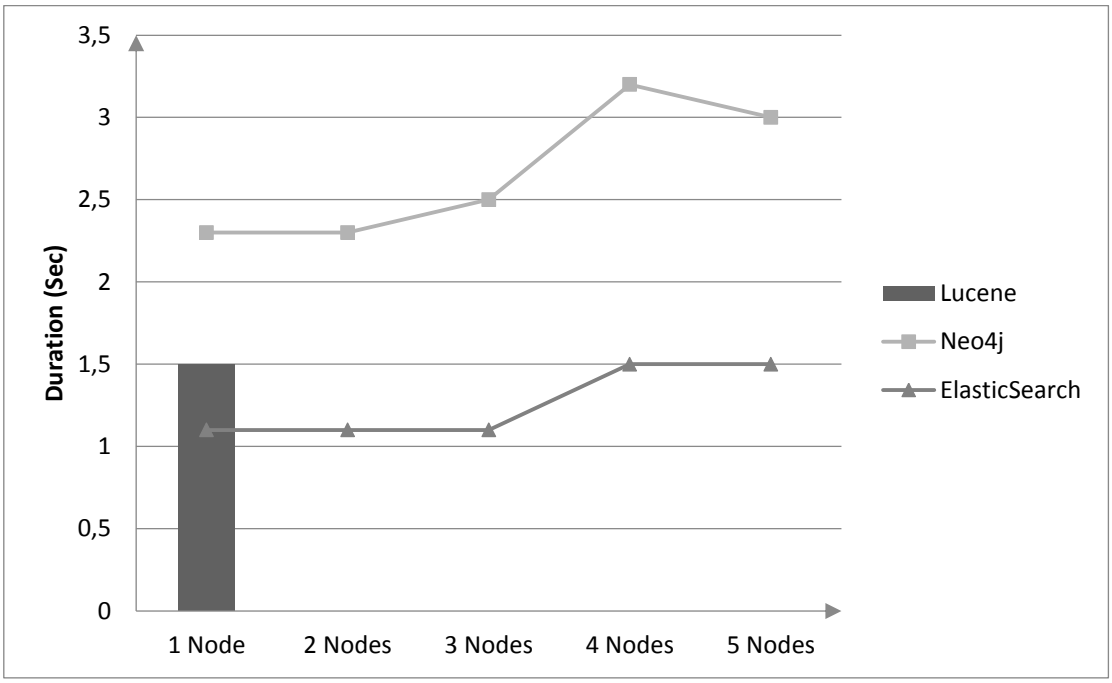

Figure 19 Generate test report

One interesting point is with a cluster of four or more nodes, the time taken to generate the report increases in both the Elasticsearch and the Neo4j implementation. With a cluster of 5, Elasticsearch completes the report generation in the same time as the Lucene implementation. Neo4j, uses 1.5 seconds more, completing the generation in 3 seconds.

5.4 LARGE DATASET RESULTS

In the tests we have conducted so far, the database consists of a small amount of data. To get a better indication of how each implementation copes with larger amounts of data we have restored in a large database. With the larger amount of data already in the system we will again do read and write test on each implementation to see if the larger amount of data has a significant impact on the performance.

For better results we should have tested with one of the largest databases but because of limited test resource tests were conducted with the largest database that our system

could handle. As the restore and start up process takes a long time for each setup we have only conducted these tests on the Lucene implementation and a 5 node cluster of Elasticsearch and Neo4j. For the following test the database has already been restored into the system.

Figure 20 shows the time taken to add a scorecard to the system. The Neo4j implementation executes this in the least amount of time. Lucene uses 14ms more at 44ms while Elasticsearch takes 97ms to add the new object. Lucene is as expected from the earlier add test. Elasticsearch takes a bit longer to complete, while the surprising result here is Neo4j. It completes the add faster with the large database in the system than with an almost empty system.

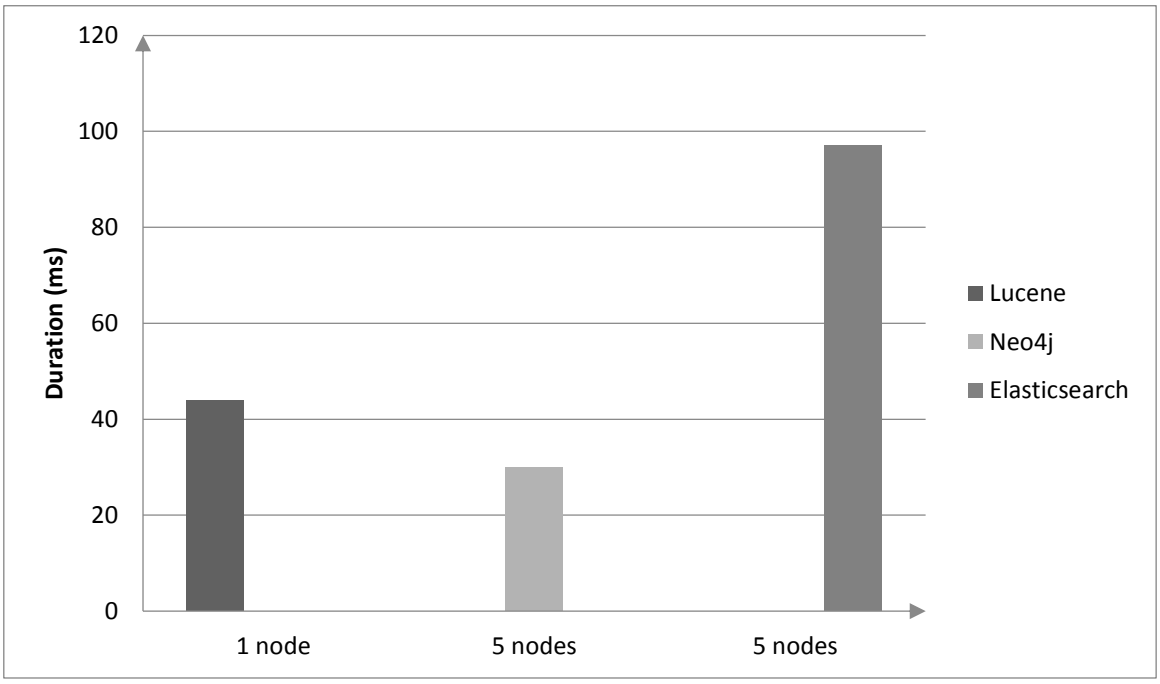

Figure 20 Add scorecard to large database

If we generate some node values as we did earlier we can see that the performance of each implementation has changed. By looking at figure 21 we can see that Elasticsearch is actually the one who performs the best. This is very interesting as in the previous test Elasticsearch was the one with the worst performance. Neo4j is the complete opposite as it under the demo value generation test is the one with the slowest performance.

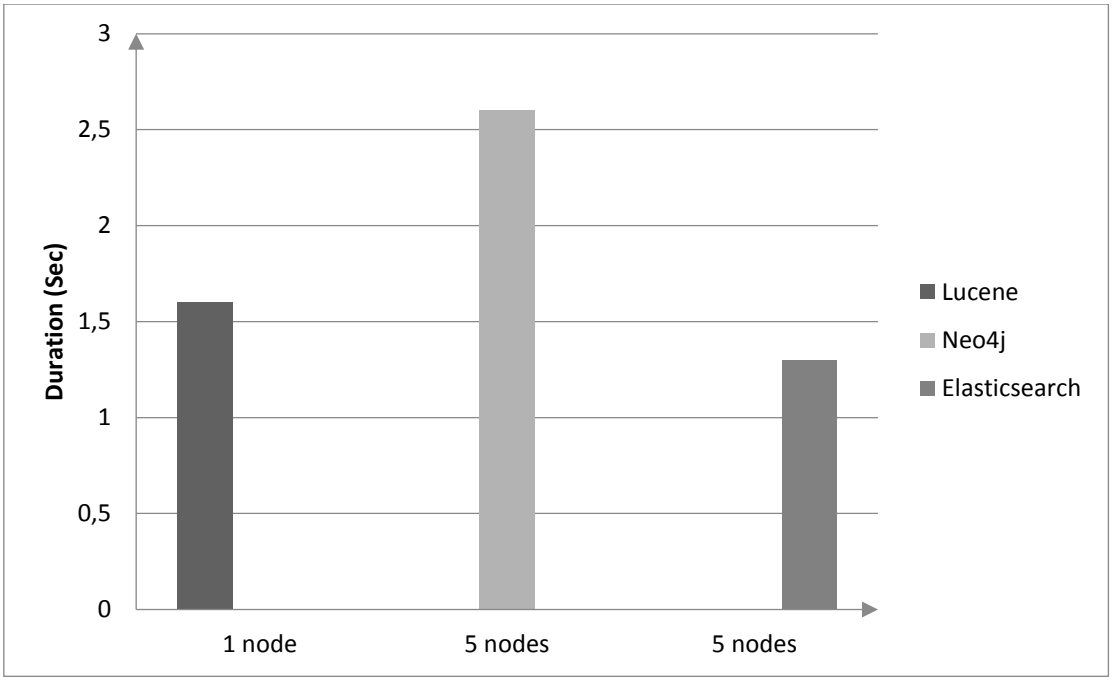

Figure 21 Generate demo values

In our last test we would like to see if the large amount of data affected the read performance of the system. To test this, we did a similar test as earlier by generating a report. The report generated here is a great deal more complex than the earlier report generated. This should result in a longer time to generate the report as more objects have to be located. In figure 22 we generate a report containing some different objects and some calculations. In this test the Lucene implementation was the one to generate the report in the shortest amount of time, then Elasticsearch and finally Neo4j. There is not a great difference between Lucene and Elasticsearch but the Neo4j implementation is quite a bit slower than the two others. This is the same as with the earlier report generation test.

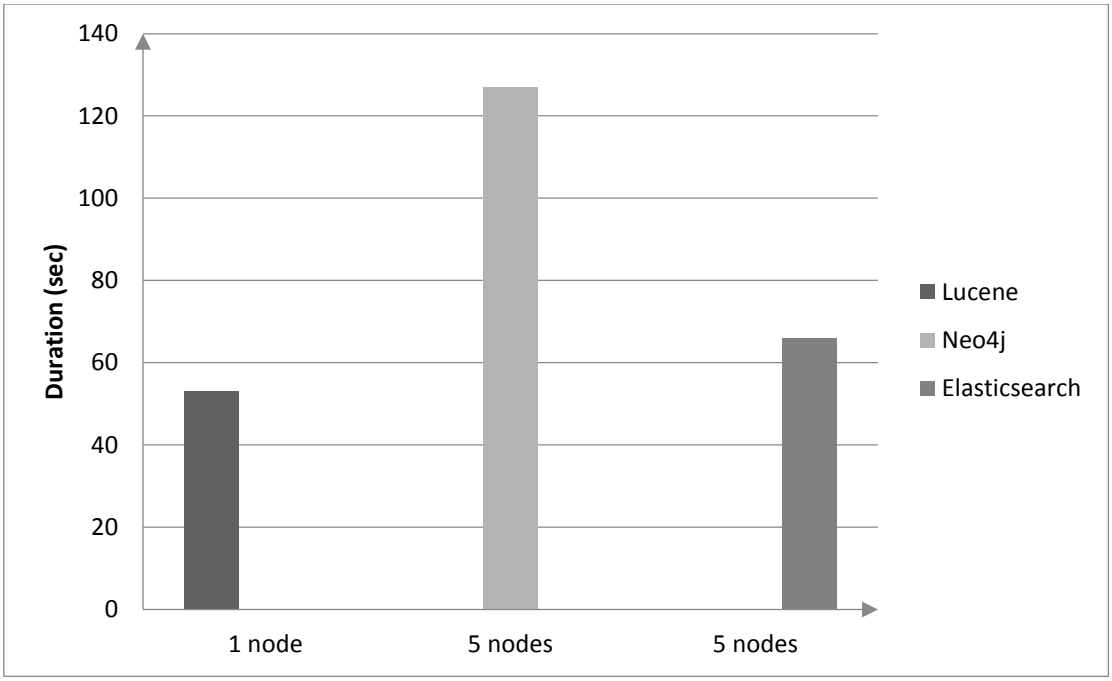

Figure 22 Generate report

In this chapter we have had a look at the read and write performance of each implementation compared to the current EPM Suite with Lucene. We saw that in some cases the performance of the new implementations actually was better while in others they were considerably slower. We also saw that increasing the size of the cluster over 4 nodes gave us in some scenarios increased performance of the cluster. In our last test with a larger amount of data already present in the system, the performance of each implementation did not decrease. Surprisingly, in some cases the system actually performed better with the larger amounts of data in the system than it did with almost no data

6 DISCUSSION

When comparing the Neo4j and Elasticsearch implementations of EPM to the Lucene implementation, the overall performance is not at the same level as the current EPM Suite Lucene implementation. In the initial test the performance of the Elasticsearch implementation were close to the performance of Lucene. Neo4js performance did get close but was still behind Elasticsearch in both writing and reading. When we increased the size of the cluster we saw an increase in the performance but it was still not on the same level as Lucene.

Comparing the results of the read tests, the read performance did not degrade despite the larger amount of data in the system. Lucene was still the one with the best performance but both Elasticsearch and Neo4j were as expected based on the first read test.

Despite the results of the read test, the performance changed drastically when retrieving large amounts of data. When starting up the server and navigating around in CS, both implementations took longer time than the Lucene implementation. Under start up all node data was retrieved and cached for fast calculations later. As a result of this, starting up the Elasticsearch implementation with the large dataset took 174 minutes. In comparison Lucene used 7.8 minutes to start the same database. In the Neo4j implementation we retrieved the nodes from the graph when needed. Because of this starting up EPM Suite with Neo4j took the same time as the Lucene implementation. But when the data later was needed the retrieval was extremely slow. So slow in fact, that the timeout of the webserver had to be extended to prevent it from closing the connections to the clients. The default timeout was 30 seconds but had to be extended to 4 minutes as some extractions took 3.5 minutes to complete. Because of this navigation in CS was perceived as very slow. This is a result of a lot of data being retrieved eagerly to prevent a delay on a simple operation like expanding a tree in CS. So when adding an object to the large dataset the actual add did not take extensive time to complete, but updating the UI after was the reason for the perceived slow system.

With the index being such a critical component of EPM Suite, the performance of it is critical for the system as a whole. This one index is responsible for all the objects in the system. In many cases 2/3 of the total amount of objects may be node data. This is data that not necessarily has to be updated as soon as it is added to the system. If the node data was extracted to a separate index, the main index could be reduced to only containing the metadata objects. This would give us two advantages. First, navigation in CS and on the web would be more responsive as the index is not containing a large amount of objects. Secondly, all the node calculations can be performed separately and made available when they have completed fully. This would result in the new calculations not being available as soon as they have been done, but it would result in a system that has a much more stable performance.

Another interesting perspective of the two implementations is how they divide the data over each instance in the cluster. As mentioned in the beginning of this thesis, Neo4j replicates the entire graph to each node in the cluster. This results in all the data replicated *x* number of times where *x* is the number of nodes in the cluster. In addition each instance in the cluster needs a large amount of RAM to handle the whole graph.

Elasticsearch has implemented sharding of the index. As this splits the data across the instances in the cluster, it offers better scalability than Neo4j. Also since each instance is responsible for a smaller amount of data, each instance does not need as much RAM as with Neo4j.

6.1 SIMPLE PROOF OF CONCEPT

From the results of testing our implementations of EPM Suite with Neo4j and Elasticsearch, the amount of data in one index is a challenge when retrieving large amounts of data from the index. In chapter 7 we suggested to split out the node data from the main index into a separate index. This could have its own life cycle, existing beside the main index. To give us an indication of how this implementation would perform we have created a simple proof of concept.

In our proof of concept, we have achieved this alternative by removing all the node data from the system. This is a very crude solution, but will give us an indication of how the system will perform when this data is no longer present in the main index.

We have used two different databases *DB 1* and *DB 2*, one medium sized and one large. In both of these databases all node data was removed.

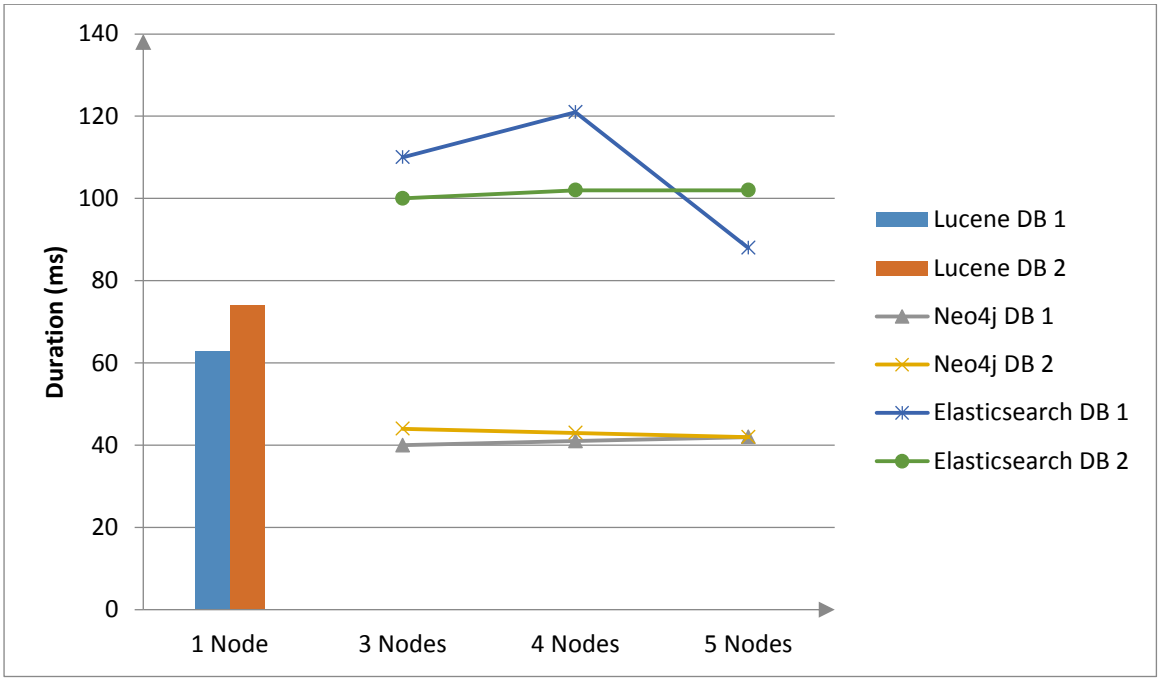

Figure 23 – POC Add scorecard

If we look at figure 23 we can see how much time each implementation used to add an object to the system. While Elasticsearch used more time than Lucene to add the scorecard, Neo4j executed the add in less time. It is also interesting to see that the time Neo4j uses increases with the smallest database, while it decreases with the largest. Elasticsearch has an increase in time before it drops back down again. With the larger database Elasticsearch has a very slight increase in time. In both Neo4j and Elasticsearch, this increase is not really a negative aspect as it is so small. For each added node to the cluster, the time only increased with 2ms. It was also somewhat expected as, when the cluster grows, the data is spread across more nodes and more network traffic is necessary.

For a large write test we have done the same test as earlier by generating some node data. In comparison to the node data we have removed, this is a small amount of data. It

is also not connected to any calculations so it will not affect the performance of the system.

In figure 24 we generate the node data which is then added to the system. Again, Lucene is the one to execute the add operation fastest. Neo4j has a slight increase in time as the cluster gets larger while Elasticsearch has the opposite effect. The large database results of Elasticsearch are a bit surprising here compared to the previous write test. In this test, the write performance had an increase in performance as the cluster size increased while in the previous test, it had a decrease.

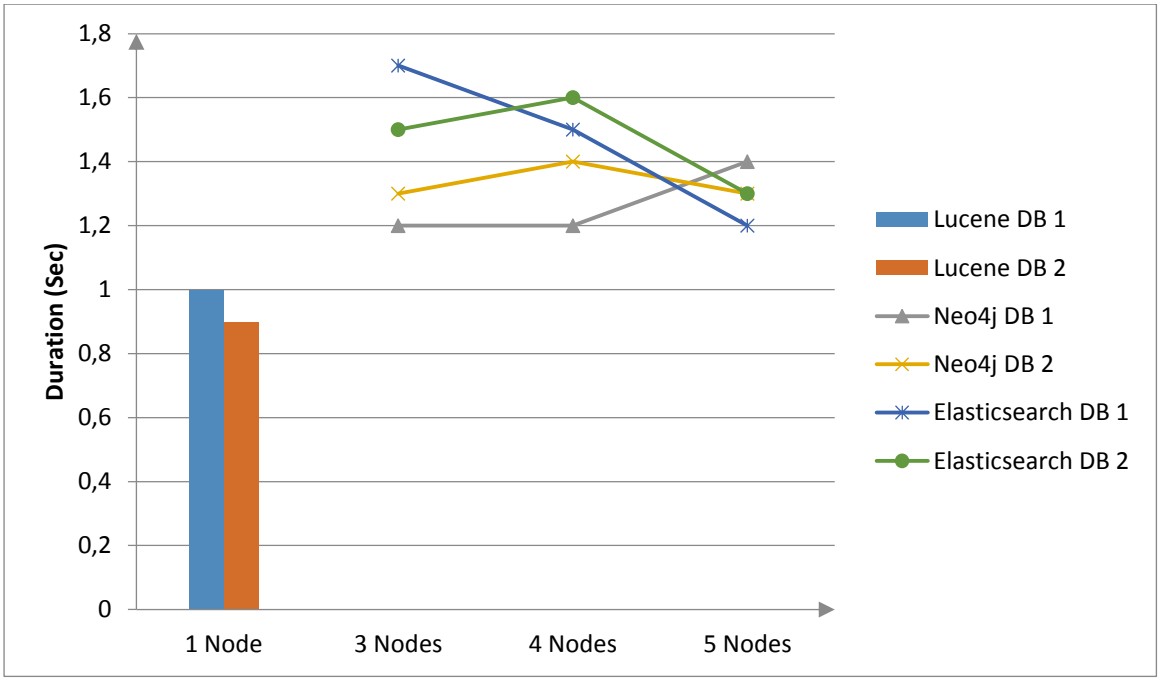

Figure 24 – POC Generate node data

We can also see that the total size of the data in the system increases the time to execute an add operation. The positive results is that with Elasticsearch this can be counteracted by increasing the cluster size. For the same amount of data, by increasing the cluster size, the same write operation will complete faster with a larger cluster.

For our final test we have tested the read performance. To test this, we generated a report. As the data structure of each database was different we have tested one report on each database. These two cannot be directly compared but does give an indication of the performance compared to Lucene and the cluster size.

In figure 25 we have generated one report in *DB 1* and one in *DB 2.* If we look at the time taken to generate the report in DB 1, we can see that both take almost the same time as Lucene. When increasing the cluster size, the Neo4j implementation has a slight increase in time while Elasticsearch has a slight decrease in time.

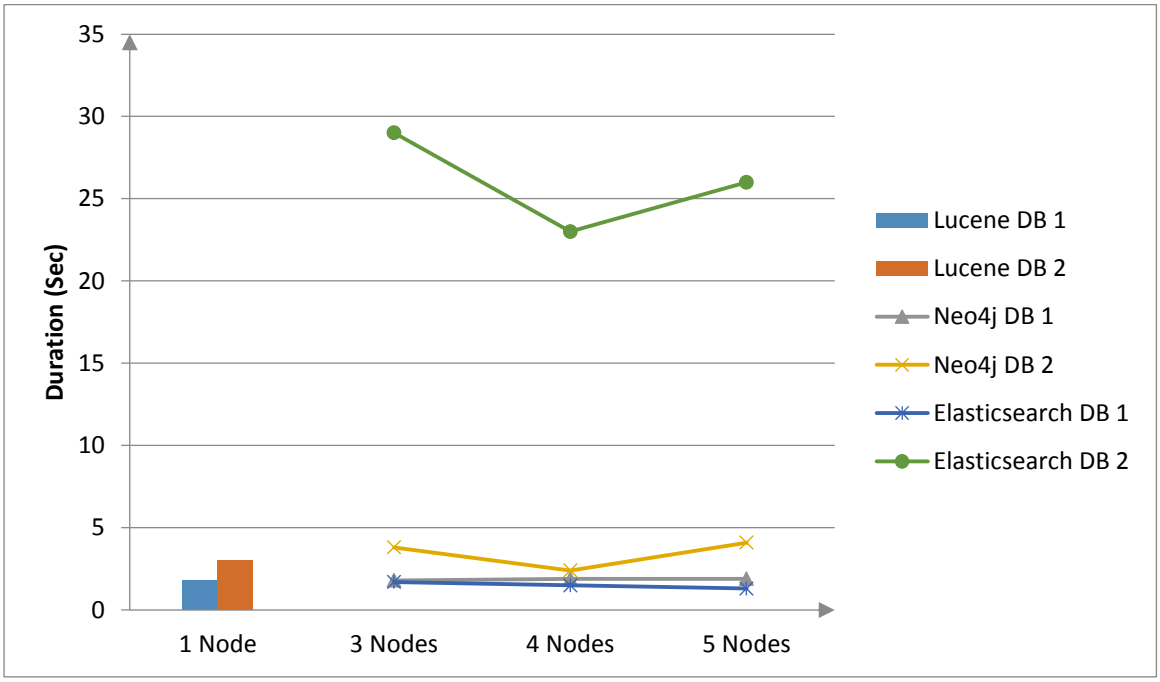

Figure 25 – POC Generate report

When looking at the performance of the report generated in DB 2, the Neo4j performance has a slight increase in time with a larger cluster size. Elasticsearch uses a lot more time to generate the same report compared to both Lucene and Neo4j. If we look more at the graph, we can see that by adding more nodes to the cluster, Elasticsearch has a decrease in time taken to generate the report. The cluster size has the opposite effect on Neo4j, more nodes in the cluster increases the time to generate the report. If we compare these results to Lucene, we can see that Neo4j has the best performance generating the report. Elasticsearch, on the other hand, has a better scalability as an increase in cluster size also increases the performance.

From figure 25 we can see a slight indication that Elasticsearch read performance increases by adding more nodes to the cluster. To see if this is correct, we have also tested changing the number of shards the cluster is divided into. In figure 26 we have the Elasticsearch results from DB 2 from our previous test plus a new database. This new database contains double the amount of data than in DB 2 but the index is also divided into double the amount of shards. The first observation is that DB 3 uses a lot more time to complete than DB 2. This is because of the added amount of objects in the database. As a result of this, the report has becomes larger and more objects are included into the report.

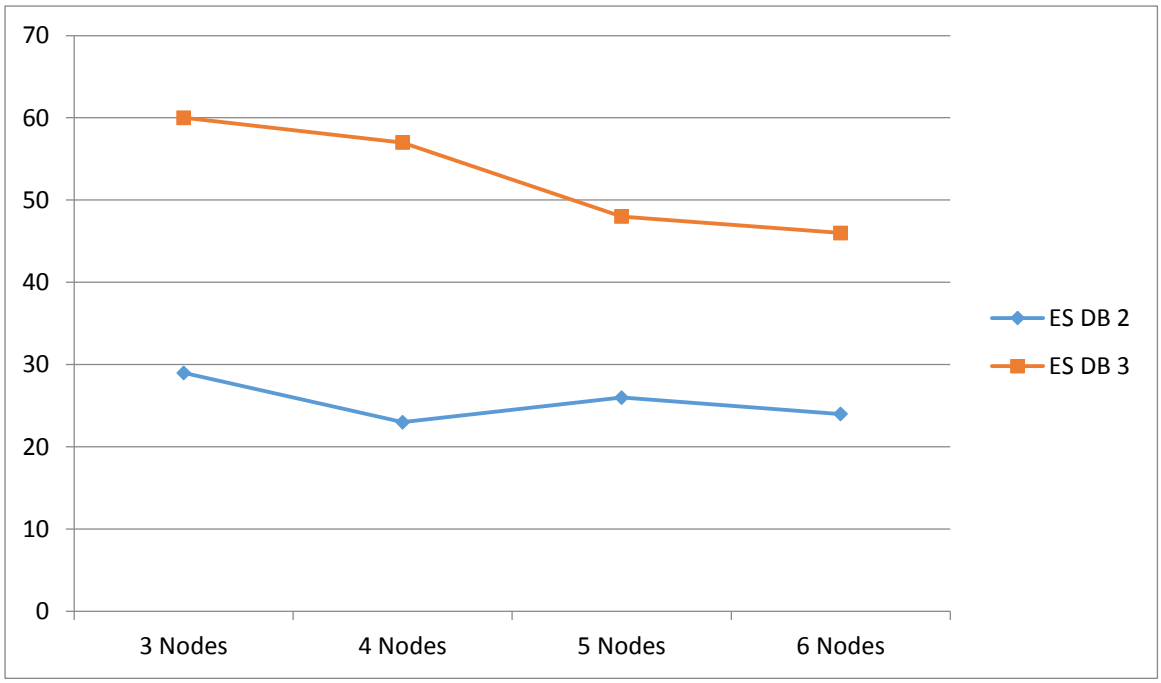

Figure 26 POC Generate report with different amount of shards

This is the reason for the gap between DB 2 and DB 3. The important observation here is how the performance increases. When using DB 2 with the added amount of shards, the test results indicated not enough data. Because of this we have had to increase the amount of data. By doubling the amount of shards the index is divided into, we can see that there is a larger increase in performance when more nodes is added to the cluster. With the added amount of shards, each shard is smaller. As a result, the time taken to search through each shard decreases. It is also to spread the shards more evenly across the whole cluster. With shards being spread across the whole cluster, each node can be better utilized resulting in better performance.

The gap between Elasticsearch and Neo4j/Lucene seen in figure 25, is most likely a result of two things. First being the wrong amount of shards and secondly, Elasticsearch uses more time to locate the objects. When profiling the Elasticsearch implementation, most of the time was used on searching for objects. By increasing the number of shards the index is divided into, we could utilize the index better. For searching for objects we are using Lucene queries. We could also try to optimize these and see if this would result in better search results.

7 CONCLUSION

In this thesis we have created two new implementations of EPM Suite which we have compared to Corporater EPM Suite.

With growing customers and higher demands of Corporater EPM Suite, it was necessary to create an EPM Suite with better scalability and the possibility for high availability. With the current Corporater EPM Suite using a Lucene index as the main index and backbone in the system, the current EPM Suite did not deliver the desired features that customers were demanding of Corporater EPM Suite. As Corporater EPM Suite differs in the way they do calculations, they do not have the same challenges that other Enterprise Performance Management systems have.

In this thesis we have developed two implementations of EPM Suite which have the possibility of distributing the index over several servers. This has been achieved by replacing the Lucene index in Corporater EPM Suite with Elasticsearch in one implementation and a Neo4j graph in the other. With the possibility of distributing the index in the system, we have evaluated the performance of our two new implementations against the current Corporater EPM Suit. From our initial read and write tests, we saw that increasing the cluster size also gave an increase in performance in both of our implementations. With five nodes in the cluster, the Elasticsearch implementation performed at almost the same level as the current Corporater EPM Suite. When we increased the amount of data in the system, the read and write tests performed at the same level as with the small data tests. The surprising result was the large amount of time taken to retrieve larger amounts of data from the two new implementations. Because of how EPM Suite is implemented, a large amount of data is retrieved eagerly to improve the response when navigation in CS and on the Web. As a result of this, when adding an object to the system, it seemed like the add action used a long time to execute. This was in fact not the case. The add itself executed within the expected time. The reason for the perceived slow responsiveness was the get action performed to update the UI after the add action.

Based on these test results we suggested a solution of moving the node data into a separate index which could have its own life cycle. We created a simple POC to give us a better indication if this was a viable alternative.

From our results and testing we can see that, by using Elasticsearch, we can achieve the added features of high availability and are also able to get a scalable solution with sharding of the index. If we were to use Neo4j as our new index, scaling would not be achieved in a good way. Neo4j does not shard the data and increasing the cluster size, in most cases, also increases execution time. If it is possible to increase the read performance of Elasticsearch in EPM Suite, using this instead of Lucene, would result in the possibility of running a clustered EPM Suite. By using Elasticsearch we would also achieve good scalability as the index would be sharded across all the nodes in the Elasticsearch cluster. If it is not possible to increase the read performance of Elasticsearch, Neo4j would result in a better performing distributed EPM Suite. On the negative side, using Neo4j would not create a good scalable solution as Neo4j does not shard the graph among nodes in the cluster.

For an optimal solution we would need to change how EPM Suite is implemented to be able to achieve a solution that performs well with large amounts of data. With the present implementation of EPM Suite, too large amount of data is contained in one index. In some cases, most of this data is data that could be extracted into a separate index. With this data in a separate index, calculations could be performed separately without disturbing the main index. This would also result in a more responsive navigation in CS and the web as it would contain a much smaller amount of data.

7.1 FURTHER WORK

In our implementations of EPM Suite and in the current Corporater EPM Suite, all the data is indexed in one index. This can be split into two separate indexes. To do this, we would need to change the structure of Corporater EPM Suite to support one index for metadata objects and one for node data. In this thesis, we created a simple POC to give us an indication if this was a valid solution. Further work would be to create a more complete implementation using Elasticsearch as the index, extracting the node data into a separate index. More time would also have to be spent on optimizing the queries sent to Elasticsearch to increase the read performance.

When this has been achieved, it is possible to do calculations on the node data independently to the rest of the system. If the system should need to update the node calculations, this can be started separate from the main index. Once the calculations have been completed they can be made available to the rest of the system. The result of this is a system with a much more responsive navigation but, the new data will take some more time before it is available for the user.

8 BIBLIOGRAPHY

[1] S. I. Inc., "Performance Management | SAS," SAS Institute Inc., [Online]. Available: http://www.sas.com/no_no/software/performance-management.html. [Accessed 28 05 2014].

[2] Q. S. Oyj, "Process Management and Performance Management Products - QPR Software," [Online]. Available: http://www.qpr.com/products/products-overview.htm. [Accessed 28 05 2014].

[3] A. S. Foundation. [Online]. Available: http://lucene.apache.org/core/. [Accessed 28 04 2014].

[4] C. Z. X. W. Yinan Jing, "An Empirical Study on Performance Comparison of Lucene and Relational Database," International Conference on Communication Software and Networks, Shanghai, China, 2009.

[5] A. Lucene, "Apache Lucene - Scoring," Apache Lucene, 25 12 2012. [Online]. Available: http://lucene.apache.org/core/3_6_2/scoring.html. [Accessed 04 29 2014].

[6] I. Neo Technology, "Neo4j - The world`s leading graph database," Neo Technology, Inc., [Online]. Available: http://www.neo4j.org/. [Accessed 28 04 2013].

[7] N. T. Inc., "Neo4j - The world`s leading Graph Database - Develop," Neo Technology Inc., [Online]. Available: http://www.neo4j.org/develop. [Accessed 28 05 2014].

[8] L. Lamport, "Paxos Made Simple," 2001.

[9] R. v. Renesse, "Paxos Made Moderatly Complex," Cornell University, 2011.

[10] M. Hunger, "FOSDEM 2013 - An in depth discussion of the Neo4j HA architecture," 20 02 2013. [Online]. Available:

https://www.youtube.com/watch?v=vLg18Yv0SVE. [Accessed 29 04 2014].

[11] M. Hunger, "New Neo4j Auto HA Cluster," 6 02 2013. [Online]. Available: http://www.slideshare.net/jexp/new-neo4j-auto-ha-cluster?utm_source=slideshow02&utm_medium=ssemail&utm_campaign=share_slideshow_loggedout. [Accessed 2014 04 29].

[12] "TinkerPop," [Online]. Available: http://www.tinkerpop.com/. [Accessed 2014 05 28].

[13] TinkerPop, "Home tinkerpop/gremlin Wiki - GitHub," 14 04 2014. [Online]. Available: https://github.com/tinkerpop/gremlin/wiki. [Accessed 29 04 2014].

[14] neotechnology, "7.1. What is Cypher? - The Neo4j Manual v2.0.2," [Online]. Available: http://docs.neo4j.org/chunked/stable/cypher-introduction.html. [Accessed 29 04 2014].

[15] Elasticsearch, "what is elasticsearch?," Elasticsearch, [Online]. Available: http://www.elasticsearch.org/overview/elasticsearch/. [Accessed 14 04 29].

[16] Elasticsearch, "Query DSL," Elasticsearch, [Online]. Available: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl.html. [Accessed 28 05 2014].

[17] Elasticsearch, "Guide," Elasticsearch, [Online]. Available: http://www.elasticsearch.org/guide/. [Accessed 05 28 2014].