Investigating NoSQL from a SQL Perspective

PETER SÖDERGREN
and BJÖRN ENGLUND

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2011

Investigating NoSQL from a SQL Perspective

P E T E R   S Ö D E R G R E N
a n d   B J Ö R N   E N G L U N D

Bachelor's Thesis in Engineering and Management (15 ECTS credits)
at the School of Industrial Engineering and Management
Royal Institute of Technology year 2011
Supervisor at CSC was Cristian Bogdan
Examiner was Stefan Arnborg

URL: www.csc.kth.se/utbildning/kandidatexjobb/teknikmanagement/2011/
sodergren_peter_OCH_englund_bjorn_K11103.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

This report investigates the current branches in the selection of databases available today. What do the new, non-relational databases have to offer? What differentiations can be found? What are their pros and cons?

First off we will cover more specifically what these benefits are, and in this first part of the report cover a broader spectrum of databases. Amongst others, we look at characteristics such as sharding and replication. As examples, we also look specifically at five databases to compare them with each other: Dynamo (Amazon S3), BigTable (Google App Engine), CouchDB, MongoDB and Neo4j.

In the second part we will run a performance test on three of the databases, namely CouchDB, MongoDB and Neo4j. We will for this test use a type of data that most accurately can be described as data found when running a social network, containing persons, blog posts, relations, events and comments.

The purpose of this investigation is to look into who could benefit from putting their SQL database in the attic, and to start using a non-relational database instead.

More than just explaining the different characteristics of non-relational databases, we present a table in the results section where we compare the five different databases with each other. We also have a section under the results chapter covering the general pro's and con's that can be found for using SQL vs NoSQL databases depending on what type of data you store.

The performance test results shows us that MongoDB was the fastest one, because it was the best suited database for our type of data and queries. Neo4j also performed good, especially in regards to execution speed as function of data size. CouchDB produced the slowest execution times since our data and queries suited the database poorly. This ment that CouchDB had to send large amounts of data to our program for external filtering and thus performed worse.

Sammanfattning

Denna rapport utreder de i dagsläget olika typerna av databaser som finns tillgängliga. Vad har de nya, icke-relationella databaserna att erbjuda? Vilka skillnader finns? Vad är deras respektive för- och nackdelar?

I den första delen av rapporten så kommer vi att mer specifikt undersöka vilka fördelarna är med hjälp av ett bredare spektrum av databaser. Bland annat så tittar vi på egenskaper så som sharding och replication. För vår jämförelse så har vi valt att fokusera på fem databaser: Dynamo (Amazon S3), BigTable (Google App Engine), CouchDB, MongoDB samt Neo4j.

I den andra delen kör vi ett prestandatest på tre av databaserna, nämligen CouchDB, MongoDB och Neo4j. För detta test använder vi en typ av data som bäst kan liknas med den data man stöter på för sociala nätverk. Vi använder oss av: personer, blogginlägg, relationer, evenemang samt kommentarer.

Syftet med denna utredning är att försöka få en bild över vem som faktiskt kan tänkas dra nytta av att lägga undan sin SQL-databas på vinden, och istället börja använda en icke-relationell databas istället.

Utöver att bara förklara skillnaderna sammanställer vi även de olika egenskaperna för de olika databaserna i en tabell för att göra jämförelsen mer lättöverskådlig. Vi förser även läsaren med ett avsnitt i resultat-delen som behandlar olika typer av data och deras lämplighet med olika typer av NoSQL-databaser. Prestandatesterna visade att MongoDB var snabbast eftersom den var bäst lämpad för den data och de frågor som vi testade med. Neo4J gav även goda resultat framför allt med avseende på skillnaden i tidsåtgång beroende på datastorlek. CouchDB visade sig ta längst tid av alla de NoSQL-databaserna vi testade på grund av att den inte lämpade sig för den data och de frågor vi valde. Detta gjorde att CouchDB behövde sända stora mängder data till programmet för extern filtrering, vilket fick längre tidsåtgång som följd.

Table of Contents

1 Introduction

1.1 Background

Databases are an important element in today's information society. They have been around almost as long as we have been having computers. The norm for databases over the years have been of a relational nature. One of the bigger reasons for this is that relational databases are well fit for storing data when you require the data to be consistent and error-handling to be swift and fully functional. These attributes put requirements on the databases, and it lowers performance, especially as the amount of data grows. Some companies ran into these performance issues, and decided to develop new models for how data is stored in a database, typically lowering consistency but increasing performance at high data levels.

Over recent years we have been seeing more and more alternative models for databases and how they work, so called non-relational (or NoSQL) databases.

1.2 Problem description

What benefits and drawbacks do the different types of databases have? What type of data benefit from what type of database? Is scalability (performance at high volumes of data) the only upside with NoSQL databases?

1.3 Purpose of the report

The purpose is to investigate the database systesms which are commonly classified as NoSQL database systems and their aspects while also giving guidelines on what type of database you should use for maximum efficiency and flexibility depending on the type of data you want to store. In addition we will also try to establish the properties of each database system.

1.4 Delimitations

The intended size of this report implies that we cannot investigate and run performance tests on all the different types of data and databases. We will pick one type of data and test it with a selection of relevant databases.

1.5 Target group

The target group for the report is those who are interested in databases, and especially those who are looking for the best database for their program/business. The requirements needed to understand this report are not high, since the nature of the report is that we explain most of the technical terms that we use. Anyone who is used to working with computers should be able to follow without any greater difficulty.

2 Method

In order to understand who could benefit from transitioning from SQL to NoSQL databases, we will in this report have two foci:

1. A general search for information, covering the different attributes and aspects of different NoSQL databases and the different types of data that is supposed to be stored in them. We will compare the databases both to their SQL alternatives and each other.
2. A performance and usability test of MongoDB, Neo4j and CouchDB.

The first part is done by searching for and reading scientific articles covering the different types of databases and their pros and cons. We will then compile the information we gathered and present it in a summarizing manner in the report.

The second part is done by generating our own data and putting the data into the three chosen databases. We will then compare their speeds in handling this type of data that we stored in them, and also see if there are any differences in usability and/or flexibility regarding for example the complexity of queries.

In the final parts of this report we will give recommendations as to who can benefit from transitioning from a SQL to a NoSQL database. We will also present the results of the performance test, showing which one of the tested databases was the best suited one for the type of data that we chose to store.

2.1 Motivating the chosen methods

- *The time frame available:* The general information search will require a rather large amount of time. Combined with the performance test, there is no room for additional methods to be used in the report considering the time frame available and the intended size of this report.
- *The nature of the research:* The focus of the report is to give a picture of the database situation today and to cover what kinds of up- and downsides the relatively new NoSQL databases have, and in the end present a performance test executed by ourselves. The large amount and broad spectrum of information that needs to be gathered motivates a *general search* for what databases are available and what they offer, and also what already has been written on the subject. This is done by looking at specific databases and reading articles covering the subject.
  The performance test is made to compare some of the NoSQL databases available in how they can handle the type of data that we chose, giving some hands-on test results to give our recommendations on which one of the databases we tested that is the most suitable one for this type of data.
- *General search vs performance test:* We use two main methods in this report. The general search covers many different aspects of the NoSQL databases, whereas the performance test is limited to testing a very small portion of the things we have been covering in the more general compilation of scientific articles. The ultimate report would have included a test of all aspects covered in the general compilation, but that is not possible because of several reasons, one being that it would be completely out of scope of the intended size of this report. We still wanted to have a performance test, and hence it will in this report not be possible to cover more than just a few aspects of this subject.

3 State of the art and theory

3.1 Technical terms

In this section we will try to cover all technical terms used throughout this thesis in order to establish our interpretation of these in order to avoid any misunderstandings.

3.1.1 ACID

ACID stands for

- Atomicity
- Consistency
- Isolation
- Durability

ACID was first stated as an acronym in the 1980's to provide a standard for properties required to make a RDBMS run without exceptions or data loss even when errors in storage or network occurred.[15] These are all requirements which are usually implemented in standard RDBMS's. They are put in place to ensure that the database runs without any errors and as expected. Without ACID the database run the risk of becoming inconsistent. Most NoSQL databases exploit the fact that all of these properties does not have to be fulfilled or that they can be avoided by adding these properties outside of the database layer.

3.1.2 Atomicity

Atomicity is a property which means that each transaction should be atomic. If each transaction is atomic then a defined set of calls to the database is either all committed to the database or none of the calls are committed. This property ensures that should the database encounter an error in the middle of a running transaction then none of the calls made will be committed to the database permanently. To ensure these demands a standard RDBMS uses a log file where it writes all operations corresponding to each transaction and only when the last operation of a transaction has been written to disk the changes to the database are permanent. In that case, should an error occur to the database between the time of the last log write and the last write to the database then these will be written when the database is restarted and if the error occurred before the first write to the log then the database will be rolled back to the state it was in before the transaction.

3.1.3 Consistency

Consistency is a property which describes a database that will only go from one consistent state to another and that it will never at any time be at a state where its rules are broken or it is giving different processes a different value for the same key at different database nodes. This is quite hard to ensure and in standard RDBMS this is enforced and means that a change has to propagate to all the relevant database nodes before the value may be read again. It also means data the database system may never be in a state where the restrictions made to the data do not apply.

3.1.4 Isolation

Isolation is a property which states that each transaction should run in isolation of all other transactions and more specifically each transaction should execute in such a manner that all transactions that have been committed to the database would have yielded the same results as if they had been executed serially, meaning that no other transaction would begin before the previous one had committed. This property is very hard to implement and standard RDBMS's use the log, read and write locks, deadlock detection and rollbacks in order to ensure this property.

3.1.5 Durability

Durability means that any committed transactions must be preserved in the database even if a crash occurs. In standard RDBMS's this is done with a log which is committed before the acutal write to disk. In the event that the database crashes, the client redoes the committed transactions which have not had their updates committed to disk.

3.1.6 MapReduce

MapReduce is technique which was patented by Google in 2004 and described in a corresponding article. [22] MapReduce is a technique to fetch large amounts of data fast, by splitting up the fetching and filtering between severeal different computers in a network. The main idea behind map reduce is that each MapReduce operation uses two specified functions, map() and reduce(). These functions are then loaded onto each computer in the cluster and each function is then executed with the data as parameters.

The map function takes a key and a value as parameters and produces an intermediate key-value pair. When all intermediate key-value pairs have been produced they are combined and then each distinct key is sent to the reduce function with its corresponding iterator of values associated with the specific key.

```
map(String key,String value)          →   list(String key,String value)
reduce(String key, Iterator values)   →   list(String value)
```

By implementing mapreduce the developer can make fectches of data using arbitrary filters. Another benefit of using mapreduce is the possibility to ask the database questions which are no less restricted than the possibilities of the programming language in which the mapreduce pattern is implemented. The only potential problem with mapreduce regarding the limit of the question-span is the fact that mapreduce is constructed to execute each map and reduce function in isolation and thus makes it impossible to execute a function which needs the data to be processed in synchronization.

3.1.7 CAP-theorem

The CAP-theorem was introduced by Eric Brewer in 2000. [23] The CAP-theorem states that databases or web services can only support two out of three of the CAP properties which are.

- Consistency
- Availablility
- Partition tolerance

Consistency was covered in an earlier section [3.2.2.2], availability is the notion that each request is always answered with a result. Partion tolerence is a property that indicate that the service is able to remain functional and operating during a network partion e.g. when a database distributed over two nodes cannot successfully communicate between the nodes.

3.1.8 Replication

Replication is a method used to increase the speed of read operations of a database. This is done by creating copies - replicas - of the database, and spreading them out over different nodes. This means that you can distribute all read operations over several nodes, processing large amounts of requests simultaneously.

3.1.9 Sharding

Sharding is a method used to increase the speed of read and write operations of a database, and also to increase the capacity of it. This is done by dividing the data over many nodes. This way, read and write operations are quickly spread out over the different nodes, allowing many simultaneous operations on the database. In order to increase the capacity of the database, all you need to do is to add new nodes.

3.1.10 REST

REST stands for representational state transfer and is provided as an API to some DBMSs . In the NoSQL area this is usually implemented by using the HTTP protocol. By requesting a resource together with method and parameters you can retrieve or modify the resource.

3.1.11 DBMS, database

DBMS stands for database management system. This is the system that is used to interact with the underlying the data. Traditionally DBMS and the database do infers different things, the DBMS is the management system and the database is the data. However in this thesis these will be used synonomously and infer the complete entity, both data and management system.

3.1.12 RDBMS

Relational database management system in this thesis denotes a standard SQL DBMS.

3.1.13 JSON

JSON is an object representation which is aimed to be easily readible. It is popular due to its simple syntax and is very common in the NoSQL area.

3.2 Focus of research

As this report could be very extensive in size if we were to look at all available NoSQL solutions we have chosen to confine ourselves to three main categories of NoSQL solutions. These are:

- Key-value stores
- Document oriented databases
- Graph databases

These categories will be covered in a general sense and be evaluated based on the criteria above. As you will notice the different solutions adhering to the different categories will not always share the same characteristics even when from the same category. To bridge this issue we have chosen to also look more closely at a few specific databases which were chosen by their recognition and importance to the field. These are:

- Dynamo (Amazon S3)
- BigTable (Google App Engine)
- MongoDB
- CouchDB

- Neo4j

3.2.1 The different types of NoSQL

We have already mentioned that NoSQL databases can be structured into categories. These categories are based upon how data is structured in the databases and the functionality provided regarding how to query, add and change the data in the database. The types of NoSQL databases we have chosen to focus on are:

- Key-value stores
- Document oriented databases
- Graph databases

The different types of databases to include when talking about NoSQL differs and there are different opinions on what is a NoSQL database and what is not. We view NoSQL as databases which are unrelational in nature and which do at most include limited support for transactions hence giving us a wide range of different types of databases to choose from.[7] We decided upon the above stated types since they each provide interesting characteristics as well as being fairly comparable to one another as we soon will see. We have deliberately not considered object relational databases or horizontally scaling RDBMS since these are similar to standard RDBMS systems and thus not in our scope.[7] We have also chosen to neglect XML databases and other databases which do not fall under a specific category. This decision was made in order to make this report less extensive and more lucid. At last the reader will also note that we have left out column-databases. Column databases are a type of NoSQL database where the data is not stored row by row but rather column by column.[8] Column databases are widely popular since they have attracted big companies like Amazon (S3), Google (App Engine) and Facebook(Cassandra).[8] We have however chosen not to include this type of database because of their similarity to traditionally RDBMS's and complexity. Additionally the greatest database providers in this field, Google and Amazon, do not provide full access to their database system but does instead use proxies in form of Google App Engine and Amazon S3.[10] These databases will however be covered individually as much as possible.

3.2.1.1 Key-value stores

A key value store is the simplest form of NoSQL database. The main idea behind the Key-value store is that it should store objects.[9] These types of databases were created in response to the back-end in many web servers whose logic is based on an object oriented rather than a tabular approach.[26] The model in these types of databases is instead very simple. It stores every object in a single index with a unique key for each object. The values in the key-value stores is usually a serialized form of the object and provides no additional functionality. All lookups in the database are based on the index and no lookups can be made with restrictions based on the values stored in the database. Each value is usually only binary data with a few exceptions. As a result these type of databases are completely schema free. Amazon S3 and Scalaris are examples of well known key-value databases.[9] Google App Engine positions itself somewhere between the key-value stores and the document oriented databases explained next. It lacks support of integrated objects but supports multiple values per key like standard tabular databases however you can add more properties on the go. We will further in this report regard Google App Engine as a key-value store since it doesn't fully support integrated objects.

3.2.1.2 Document oriented databases

Document oriented databases do as the key-value store provide a soliution for storing objects, which in these cases are called documents. The main difference however is that in document oriented databases the values are given meaning and indexes can be defined on arbitrary properties in each document.[7] Most document oriented databases do also support integrated objects and arrays. Document oriented databases were created with the dynamic web in mind and thus use no schema. This does enable the application to add functionality while the database is running and does not pose the requirement for every document to include null values for all properties they do not contain. To make this possible, document oriented databases usually use a data format for transmitting objects between programs such as XML or JSON.[9] Popular document oriented databases are MongoDB[11] and CouchDB[12].

3.2.1.3 Graph databases

Graph databases do have many similarities with the previous types in that they have no schema. However in contrast with the other alternatives graph databases are made to fascilitate relations in the database layer.[9] Graph databases demand that the data is stored in a tree format. They also have some limited support for joins but are best used for their possibility to traverse nodes in the graph.[13] This makes them good for finding the shortest route or traversing relations with a specific goal or specific depth. N4j is an example of a well known graph database.[14]

3.3 State of the art, examples

In order to give a brief overview of the different NoSQL databases available today, we decided to breifly cover Amazon S3, Google App Engine, MongoDB, CouchDB and Neo4j in the following part of the report. The reason we specifically picked these five is that they are some of the more famous NoSQL databases on the market today.

3.3.1 Amazon S3

What is Amazon S3?

Amazon S3, also known as Amazon Simple Storage Service, is classified as "Infrastructure as a Service (IaaS)", and is a key-value storage system designed to make web-scale computing easier for developers. They offer a underlying database-related infrastructure for your applications. The idea is that you pay for the amount of data that you want to store per month, and in exchange they provide the storage space required and the database with all its functionality through a web interface. This makes the end user/the customer not have to worry about:

- How to store your data
- Whether the storing will be safe and secure
- Having enough storage available
- Paying the upfront costs of setting up your own storage solution
- Maintaining and scaling its storage servers

Features of Amazon S3

- **Scalable:** Amazon S3 was built to scale efficiently with the amount of data stored
- **Reliable:** High durability and availability is guaranteed
- **Fast:** Built to handle high-performance applications
- **Inexpensive:** Amazon S3 is built from inexpensive commodity hardware components, and the eventual failure of certain hardware must not affect the overall system.
- **Simple:** Aims to be easy to use for any application, anywhere.

Quick introduction to Amazon S3 database handling
The first part of setting up your own database with Amazon S3, where everything is done through the web interface, is to create a so called *Bucket*. Buckets are nothing more than a way to specify in which region of the world the data is to be stored. This feature is a way to remedy latency problems and in this way improve performance.

The next step is to add an object to the selected bucket. This is simply done by pressing the "Add files" button in the web interface, and a file selection dialog box will appear. From here, you navigate to the file you want to upload to the database and click "Open". After this, the file will be uploaded, and when the upload has been completed, you can add so called *meta data* - assisting info to the file if needed - and then you're done.

Amazon S3, by itself, does not offer an advanced query language towards the database. In order to access such features, you must turn to Amazon's other services, such as Amazon's SimpleDB.

3.3.2 Google App Engine

What is Google App Engine?
Google App Engine (GAE) is classified as "Platform as a Service (PaaS)", and likewise to Amazon S3, it is a "external" solution for data storage. Whereas Amazon S3 is a solution for the database infrastructure alone, the App Engine is a solution within which you, in addition to gaining access to a database infrastructure solution, also run the applications. The App Engine uses Google's key-value database BigTable. Using the App Engine is free up till the point that your application exceeds either 500 MB of storage or 5 million page views a month.

Features of Google App Engine

- Dynamic web serving, including full support for common web technologies
- Persistent storage with queries, sorting and transactions
- Automatic scaling and load balancing
- API's for authenticating users and sending email using Google Accounts
- A fully featured local development environment that simulates Google App Engine on your computer
- Task queries for performing work outside of the scope of a web request
- Scheduled tasks for triggering events as specified times and regular intervals

Quick introduction to Google App Engine database handling
The App Engine supports Python and Java, or any programming language using a JVM-based interpreter or compiler, such as JavaScript or Ruby.

Example in Java:

```
DatastoreService datastore =
      DatastoreServiceFactory.getDatastoreService();

Entity student = new Entity("Student");
student.setProperty("firstname", "Ellen");
student.setProperty("surname", "Johnson");
Date addedToDb = new Date();
student.setProperty("addedToDb", addedToDb);
student.setProperty("passedMathExam", false);

datastore.put(student);
```

3.3.3 CouchDB

What is CouchDB?

CouchDB is an open-source document-oriented database. Taken from CouchDB's official website [3], we find some quick info on what CouchDB is and what it is not:

What CouchDB is:

- A document database server, accessible via a RESTful JSON API.
- Ad-hoc and schema-free with a flat address space.
- Distributed, featuring robust, incremental replication with bi-directional conflict detection and management.
- Query-able and index-able, featuring a table oriented reporting engine that uses JavaScript as a query language.

What CouchDB is not:

- A relational database.
- A replacement for relational databases.
- An object-oriented database. Or more specifically, meant to function as a seamless persistence layer for an OO programming language.

Features of CouchDB

- **Document Storage:** CouchDB stores documents in their entirety.
- **ACID Semantics:** Although CouchDB is a non-relational database, it still offers the guarantees of ACID, which usually can only be seen in relational databases, to assure no conflicts will occur while having a high amount of concurrent readers and writers.
- **Map/Reduce Views and Indexes:** CouchDB uses Google's Map/Reduce methods to create so called *views*, which are a method of aggregating and reporting on the documents in the database. They are built dynamically and you can have as many different view representations of the same data as you like. CouchDB can then index those views for later use, and keep them updated as documents are added, removed or updated, in order to avoid going through the time-consuming process of creating a view.
- **Distributed Architecture with Replication:** With CouchDB you can have multiple replicas with copies of the same data, and you can then modify this data on the different replicas, and synchronize at a later time.
- **REST API:** Communication with the database, like performing a Create, Read, Update or Delete operation, can be done through the well known HTTP methods POST, GET, PUT and DELETE. This way, you can issue requests to the database from any environment that allows HTTP requests.

Quick introduction to CouchDB database handling

CouchDB supports many different programming languages for interaction with the database. They also have a web interface available, called *futon.*

A typical CouchDB JSON document may look like this:

```
{
      "_id": "ellen_",
      "_rev": "1-98321jhhjha78y31ah7a77dahdujjhak",
      "type": "Student",
      "firstname": "Ellen",
      "surname": "Johnson",
      "personalid": "861010-0123",
      "skills": ["Math", "Biology", "Physics"]
}
```

Example in Java:

```
Session s = new Session("localhost",5984);
Database db = s.getDatabase("studentsdb");

Document newdoc = new Document();
newdoc.put("firstname","George"); //same as JSON: { firstname: "George"; }
db.saveDocument(newdoc); //creates auto-generated id given by the database

Document doc = db.getDocument("ellen_johnson");
doc.put("personalid","861010-0123");
db.saveDocument(doc);
```

3.3.4 MongoDB

What is MongoDB?

MongoDB is a scalable, high-performance, open source, document-oriented database written in C++. They describe some of the features of MongoDB on the official website[2]:

Features of MongoDB

- **Document-Oriented Storage:**
  - Documents (objects) map nicely to programming language data types
  - Embedded documents and arrays reduce need for joins
  - Dynamically-typed (schemaless) for easy schema evolution
  - No joins and no multi-document transactions for high performance and easy scalability
- **Replication & High Availability:** MongoDB offers the use of Master-slave replication. This means that you can run "slave"-copies of the master database, and in this way increase the ability to read the database efficiently. MongoDB's replication also offers a automatic master failover system, electing a new master if the current one goes down.
- **High Performance:**
  - No joins and embedding makes reads and writes fast
  - Indexes including indexing of keys from embedded documents and arrays
  - Optional streaming writes (no acknowledgements)
- **Auto-Sharding:** Scale horizontally without compromising functionality. Sharding is a method used to divide and spread the database over different machines, making the database considerably quicker to work with, and especially read, as the amount of requests and amount of data stored grows.
- **Map/Reduce Views and Indexes:** MongoDB uses Google's Map/Reduce methods to create so called *views*, which are a method of aggregating and reporting on the documents in the database. They are built dynamically and you can have as many different view

representations of the same data as you like. CouchDB can then index those views for later use, and keep them updates as documents are added, removed or updated, in order to avoid going through the time-consuming process of creating a view.

- **Rich Query Language**

Quick introduction to MongoDB database handling

MongoDB supports a long list of programming languages, including all the most frequently used ones.

Example in Java:

```
BasicDBObject doc = new BasicDBObject();

doc.put("name", "Ellen");
doc.put("surname", "Johnson");

BasicDBObject info = new BasicDBObject();

info.put("weight", 58);
info.put("lenght", 171);
info.put("age", 24);

doc.put("info", info);

coll.insert(doc);
```

3.3.5 Neo4j

What is Neo4j?

Neo4j is an open-source object-oriented graph database. Graph databases use nodes for each object stored, and excell at expressing relationships between those nodes. An example of a typical application that can benefit from this type of database is social networks.

Features of Neo4j

- **Intuitive and flexible:** Neo4j is an intuitive and flexible graph model for data representation which emphasizes the use of nodes, relationships and properties instead of tables, rows and columns.
- **Disk-based:** Neo4j is a disk-based, native storage manager optimized for storing graph data with high performance and scalability.
- **High scalability:** Neo4j can handle billions of nodes, relationships and properties on a single machine, and can be sharded - split over multiple machines - to scale efficiently.
- **Traversal:** As a graph database, Neo4j offers the subsequent high-speed traversal benefits in the node space.
- **Simple and convenient object-oriented API**
- **ACID Semantics**
- **REST API**

Quick introduction to Neo4j database handling

Neo4j supports different programming languages for interaction with the database. They also have a web interface available.

Example in Java:

```
public enum MyRelationshipTypes implements RelationshipType{
      KNOWS
}

GraphDatabaseService graphDb = new EmbeddedGraphDatabase("var/graphdb");

Transaction tx = graphDb.beginTx();
try{
      Node firstNode = graphDb.createNode();
      Node secondNode = graphDb.createNode();
      Relationship relationship =
            firstNode.createRelationShipTo(secondNode,
            MyRelationshipTypes.KNOWS);
      firstNode.setProperty("firstname", "Ellen");
      secondNode.setProperty("firstname", "George");
      relationship.setProperty("message", " knows ");
}
finally{
      tx.finish();
}

System.out.print(firstNode.getProperty("firstname"));
System.out.print(relationship.getProperty("message"));
System.out.print(secondNode.getProperty("firstname"));

graphDb.shutdown();
```

The example above prints the following:

```
Ellen knows George
```

3.4 Differences when moving from RDBMS to NoSQL

When moving from or deciding whether to move from an RDBMS(Relational Database Management System) to a NoSQL solution it is imperative to look at key differences between these two categories. First and foremost it must be said that no NoSQL solution is another alike and as the word NoSQL has gained in popularity so has the number of implementations.[4] Intially NoSQL was a database. Now NoSQL has become a buzzword for databases which are faster or more scalable than traditional RDBMS's by the renouncement of key features. The term was barely written about before 2009 but has generated far more intrest since then.[5] The name NoSQL often leads to the misunderstanding that NoSQL databases becomes more fast or scalable simply by changing the query-language. However contrary to this misunderstanding a database with a grammar similar to SQL could be just as fast or scalable and many belive NoSQL should be read ”not only SQL” instead of the intutive ”no SQL”.[6]

This section is intended to give the reader a good outlook on what to expect when moving from an RDBMS to a NoSQL solution. It will begin by a short look on what has influenced the NoSQL movement and then move on to discussing the different types of NoSQL solutions. After the two initial subsections we will cover a few different key aspects where NoSQL differs from traditonal RDBMS. These aspects are:

- Atomicity
- Consistency
- Isolation
- Durability
- Queries
- Map-reduce
- CAP-theorem
- Unrelational characteristics
- Replication
- Sharding

3.4.1 Atomicity

Depending on which NoSQL solution you use the atomicity of the transactions you make differs. When using key-value stores it differs from solution to solution. They usually support atomicity on a single key, but whether they provide it for multiple keys depend on the provider. Amazon S3 offers full atomicity over several keys[10] while Google App Engine simply offers it over a single key.[16]

Document oriented databases do also have different solutions to this problem depending on how much scalability and speed they are willing to sacrifice for atomic operations. The atomicity that is offered is usually at least on a document level, this is true for MongoDB[17] while CouchDB offers full atomicity.

Regarding graph databases it is hard to define general atomicity level although many of them offer complete atomicity and this is true as well for Neo4j.

3.4.2 Consistency

When talking about consistency there are two distinct types of consistency. Strong consistency and eventual consistency.[10] Strong consistency enforces that the database is always consistent as shown above while eventual consistency has a time window where the database is not consistent but as the change is replicated across the servers the whole database system will eventually become consistent.

When it comes to consistency NoSQL solutions differ depending on where they position themselves in the CAP triangle which will be covered later. The consistency offered is also dependent on what server setup you are running. A master-slave replicating setup will be far more likely to provide strong consistency than a master-master replicating one. Master-master and master-slave are covered in the replication section. Google App Engine provides strong consistency while Amazon S3 only provides eventual consistency. The document oriented databases differ as well. MongoDB only provides eventual consistency while CouchDB provides strong consistency but only for master-slave replication. Nothing general can be said about graph databases either but neo4j enforces strong consistency.

3.4.3 Isolation

IThis property is very hard to implement and standard RDBMS's use the log, read and write locks, deadlock detection and rollbacks in order to ensure this property. This process is quite complex and deals with many exceptions and will not be covered in this report.

Rather we choose to focus on the way in which NoSQL solutions deal with the problems of isolation. The standard way of ensuring isolation is a very expensive solution and thus does not favour scalability or speed. Instead many NoSQL databases have looked toward other solutions. The first and easiest one to discard is isolation or only implement it across single key-value or a single document. This is easily implemented and not extremely costly. However if support for bigger transactions are to be implemented there is a need for another type of isolation ensurer and the solution most in used today are forms of multiversion concurrency control.[10]

Multiversion concurrency control

Multiversion concurrency control is a method which allows for multiple processes to access the same data without corrupting it.[10] There are basically two different methods for multiversion concurrency control, vector clocks and hash histories. These are methods for detection of conflicts. If a conflict is detected the database usually forwards the problem to the client to decide how to deal with the problem.

Vector clocks is a list consisting of key value pairs which are associated with each value in the database. For each update every process adds itself as key to this list and a incremented number based on the maximum number available as the value. By doing this the database can see which process updated this value at the latest time and in which order this value was updated.

Hash histories works similarly to the vector clock but instead of storing the process which updated the value it stores the hash value of the corresponding value. This means that updates with the same value will not cause a conflict and that the list grows with the amount of updates instead of the amount of processes.

Amazon S3 is an example of a NoSQL solution without processes running in isolation. It's the latest write that will count at all times. Google App Engine however uses MVCC and re-queues the transaction over again if a conflict is detected. MongoDB offers no MVCC support while CouchDB offers MVCC by storing old revisions of the document. However isolation is not ensured over several nodes.[18] At last neo4j provides complete support for isolation but uses locks like standard RDBMS's.[19]

3.4.4 Durability

Durability means that any committed transactions must be preserved in the database even if a crash occurs. In standard RDBMS's this is done with a log which is committed before the acutal write to disk. In the event that the database crashes, the client redoes the committed transactions which have not had their updates committed to disk.

Durability is quite easily implemented in most databases but for distributed databases the meaning of durability has changed. Durability in distributed databases now means how durable the database is in the event of a catastrophe. Amazon claims their S3 solution will offer durability even if one of their data centers were destroyed. Google App Engine offers customers to choose from a high replicating and thus more durable solution that is less consistent or a solution which does not replicate but only supports standard durability.[20]

Both MongoDB and CouchDB support durability. MongoDB does so through a log, called a journal and CouchDB by always flushing its information to the disk and simply and keeping older revisions. Neo4j does as well support standard durability.

3.4.5 Queries

As discussed before NoSQL does not really imply anything about the query language. However it is interesting to note that since most NoSQL databases support a far less rich feature set than standard RDBMS's they usually choose other forms of query language. In this section we will try to establish what type of queries the different databases support and what type of query languages and access methods they incorperate.

AmazonS3 and Google App Engine support storage of objects in their databases however the key difference is that Amazon S3 is completely unaware of the contents of its objects while Google App Engine supports searches based on the objects properties. However Google does not support searches that deal with arrays or objects inside of another object. Amazon has two API:s to access its data, REST(HTTP) and SOAP(XML) which have been covered earlier. These API makes it possible to traverse buckets(directories) as well as store and delete objects(files). The REST API also makes Amazon S3 usable as an effective CDN(Content Delivery Network), a secondary server for static content on a website. Google App Engine uses a language for queries which they call GQL. GQL is quite similar to SQL in syntax but does only support a few of the standard SQL features. It can constrain selections by conditions but does only support logical AND operations for these conditions. Google App Engine can also be used to order the result and only output a subset of the selection. The standard of most key-value stores is to only support limited to none selection conditions but as stated earlier Google App Engine positions itself somewhere between a

MongoDB and CouchDB feature a richer feature set than key-value stores which is something that they share with most document oriented databases. MongoDB support selections based on several conditions and can use both, logical and & logical or, operations with the exception of nested logical or operations.[21] MongoDB also has support for grouping and aggregation functions. CouchDB on the other hand has a radically different approach. It uses map-reduce functions for all its selection. Map-reduce will be further explained in the next section.

Finally Neo4j which serves as our example of a graph database provides a completely different api. Neo4j is built for traversing nodes and thus provides an API for defining recursive functions. These traversing functions need to know to which relations they should traverse and issue a recursive call as well as when to stop traversing. Neo4J offers API's for java, python, ruby and a REST API for their Neo4j server.

3.4.6 MapReduce

```
map(String key,String value)           →   list(String key,String value)
reduce(String key, Iterator values)    →   list(String value)
```

By implementing mapreduce the developer can make fectches of data using arbitrary filters. Another benefit of using mapreduce is the possibility to ask the database questions which are no less restricted than the possibilities of the programming language in which the mapreduce pattern is implemented. The only potential problem with mapreduce regarding the limit of the question-span is the fact that mapreduce is constructed to execute each map and reduce function in isolation and thus makes it impossible to execute a function which needs the data to be processed in synchronization.

Mapreduce is not traditionally found in standard RDBMSs since standard RDBMSs use the restrictions of SQL to ensure all the properties of the database. Mapreduce is however very usual in many NoSQL-solutions where the mapreduce operation can run asynchronous with other operations, either by rejecting the isolation property or by using a work-around solution.

3.4.7 CAP-theorem

Standard RDBMSs are usually positioned to be consistent and available while having lower tolerence for network partitions. This is contrasted by most NoSQL-solutions which tend to sacrifice either consistency or availability in order to gain partion tolerence. The properties of the databases examined throughout this report will be stated below[23][24]

- Amazon S3 - Availability, Partion tolerence
- Google App Engine - Consistency, Partion tolerence
- MongoDB – Consistency, Partion tolerence
- CouchDB - Availability, Partion tolerence
- Neo4j – Information not available

3.4.8 Unrelational characterstics

In SQL there is a built in support for relations between different rows in different tables. This relational support manifests itself in the form of JOIN-operations and foreign keys. In a SQL-implemententation it is thus possible to fetch data by from a virtual table, created by a JOIN-operation. This JOIN operation combines two sets of table rows with one another by creating a new virtual table consisting of all possible combinations of a row in Table A and a row in Table B that honours the restrictions specified together with the JOIN-operation. In addition virtual tables can also be combined with other virtual tables. The ability to choose a foreign key poses a restriction to a certain table so that a row will not be allowed to be added if its foreign key columns have no direct corresponding row in another table sepecifed together with the foreign key.

These operations are crucial to standard RDBMSs and while they are an integral part they certainly are a great liability aswell. This becomes even more true the more the database grows since a bigger database means more rows and thus more rows to scan when joining tables.

3.4.9 Replication

As mentioned in section 3.1 replication increases the simultaneous read capacity to the database by replicating the data. Howerver there's also this problem with hardware failing. Earlier we've been trying to prevent failures by buying expensive hardware[25]. With replication you don't have to spend excessive resources on preventing failures, but rather if one of the nodes fail, then you have at least one other node that has the same data that can replace the one that failed.

The downside with replication is when you want to write to the database. If anything needs to be written to the database, then you need to write that to every node that is supposed to store that piece of data. It can be done in two ways: master-slave scheme or master-master/multi-master scheme. Master-slave scheme means that you first write to one node, the master, and then its state is transferred to the replicas, the slaves. This way of doing it insures availability, but not consistency. The other way, using the master-master scheme, is that any replica commits the write operation and then transfers its state.

3.4.10 Sharding

As mentioned in section 3.1 sharding can be used to divide reads and writes over several servers. However there are some problems with sharding though. The first one is that sharding makes some more advanced database operations very complex and inefficient. Typically these come in the form of the classical join operations seen in relational databases. Therefore, join operations are not supported in most sharded databases. The second downside is that the more nodes you add, the higher the probability of having a node failing. To handle this, sharding is usually combined with replication.

4 Research

4.1 Analysis: What types of data benefit from NoSQL use?

"Like we have this web 2.0 kind of sites where people just put in all their crap, and then by no means does fit within any relational database."

Jan Lehnardt, co-worker on CouchDB [28]

4.1.1 General types of data

The amount of data available has increased dramatically in recent years [29]. This data comes in different forms.

What form the data comes in and what the data is used for makes big differences on the different needs and requirements on the characteristics of the databases. Some types of data requires more flexible storing than others. Some types of data requires more secure types of storing than others, and some types of data requires higher processing speeds when writing to and reading from the databases.

The data that today's databases are supposed to handle can be categorized into three overall forms:

- Structured data
- Semi-structured data
- Unstructured data

When it comes specifically to what kind of data that benefits from the use of NoSQL DBMS, there are two relevant arguments/aspects; *performance* and *flexibility* [30]. Therefore we evaluate the data types from these two aspects.

4.1.1.1 Structured data

Some data comes in a *structured* form. This means that in every unique instance of data, every post has the same attributes; they adhere to a certain kind of form that they fit into. If you have an application handling all the students in a school, then every student can be defined from for example a first name, a surname, a personal ID number, a telephone number, an address and so on. Hence they all fit into the same structured form and they can easily be stored in a relational database. The reason that we chose to display the examples in a way that reminds you of RDBMS is that it is a intuitional and descriptive way to do so.

Structured Data				
First Name	**Surname**	**Personal ID Nr**	**Telephone Nr**	**Address**
Ellen	Johnson	861010-0123	123-12312312	3 John's Street
George	Pearson	870101-4321	234-23423423	14 Pear's Street

4.1.1.2 Semi-structured and unstructured data

The two following types of data – semi-structured and unstructured – are becoming more and more common. Some argue that already today, 80% of all potentially usable business information comes in unstructured form [31]. The original source for these numbers is IBM, but it is hard to find the exact documents covering the research, which is why you should not take these numbers for certain.

But by just looking at my own daily usage of data, then it is true that a very prominent part of it comes in the form of reading or writing e-mails, reports, articles, chatting, listening to audio or watching videos on youtube.

Moreover, there are other sources where multiple analysts, once again, estimate that more than 70-80% of all data within organizations comes in an unstructured form [32]. They also estimate that the total amount of available data will grow by 800% during the coming five years, and that the unstructured data is growing by 10-50 times more than structured data.

Semi-structured data

Some data comes in a *semi-structured* form. This means that every record has an overall structure that it adheres to, but the contents of some particular structural elements are not always consistent [33]. While some semi-structured data look very much like structured data, we will see that this is not always the case.

One type of semi-structured data could be that you have an application handling all the students in a school, where one student could be defined from a subset of all possible data associated with a student. The student could be defined from a first name, a surname, *a personal ID number* and an address, while another student could be defined from a first name, a last name, *a telephone number* and an address. This means that they follow some kind of common structure, but they do not always have information in the same fields. Some fields are missing out.

Semi-structured Data				
First Name	**Surname**	**Personal ID Nr**	**Telephone Nr**	**Address**
Ellen	Johnson	861010-0123		3 John's Street
George	Pearson		234-23423423	14 Pear's Street

We could also find ourselves in a situation where we are having a database that puts together students from databases from different countries. They all have a country, a first name, a surname, some kind of way of handling a personal ID number, a telephone number and an address. Since different countries can have different ways of handling for example ID numbers or addresses, we will have these elements entered on different forms into our database. This means that we will have the personal ID numbers of all the students, but they may look very different. This is also considered to be semi-structured data [29].The following example is a combination of different formatting and missing fields.

Semi-structured Data						
Country	**First Name**	**Surname**	**Personal ID**	**Birthdate**	**Telephone Nr**	**Address**
Sweden	Ellen	Johnson	861010-0123		123-12312312	3 John's Street
Iran	George	Pearson	104	870101	234-23423423	14 Pear's Street

In general, semi-structured data arises when data from sources that have differences to them are combined and/or when the data is stored in sources that do not impose a rigid structure, such as on the Internet.

Another example of this is an article [29] which discusses the OEM - Object Exchange Model - as a means of exchanging semi-structured data between object-oriented databases. They have an example covering some of the problems faced, as seen in the figure below: *"Observe that,* *for example, (i) restaurants have zero, one or more addresses; (ii) an address is sometimes a string and sometimes a complex structure; (iii) a zipcode may be a string or an integer; (iv) the zipcode occurs in the address for some and directly under restaurant for others; and (v) price information is sometimes given and sometimes missing."*

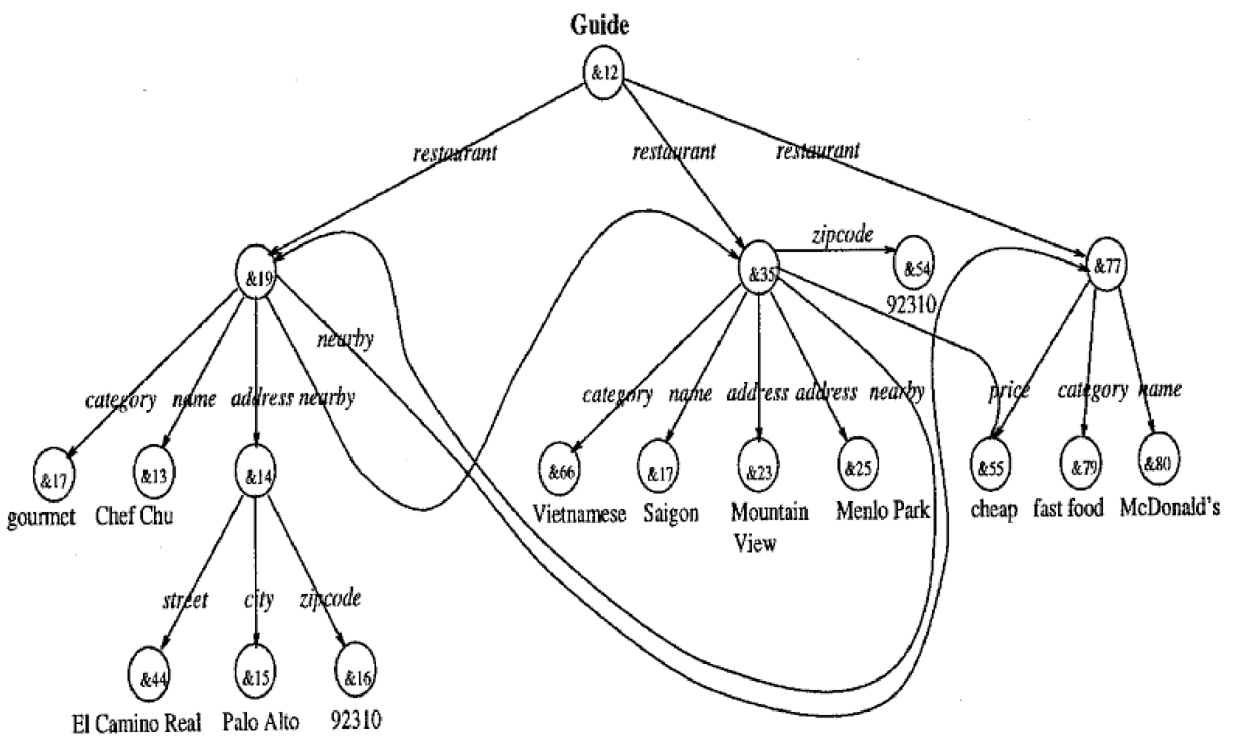


Fig. 1. An OEM graph

[29]

If comparing the first and the last example of semi-structured data, we can see that the amount of shared structure found in different posts can differ widely.

As a summary, semi-structured data can be defined as *"data that has no absolute schema fixed in advance, and whose structure may be irregular or incomplete."* [34]

Unstructured data

Some data, and some would argue more than just "some", comes in a *unstructured* form. A quick definition of unstructured data in the context of relational databases is: data that can not be stored in rows and columns [35]. This definition needs some explanation though. What can be said more elaborately is that unstructured data, unlike structured data, *"lacks the explicit semantic structure necessary for computerized interpretation"* [36]. The unstructured data contains elements which by themselves do not consist of data that a computer can find any relevant structure in, or rather, which a computer can not interpret in a sufficient way.

Unstructured data is sometimes also refered to as *raw data*. Although there is sometimes a need for distinction of unstructured data from raw data.

Raw data is data in a form that a database by itself cannot analyze and do anything useful with. The major types of raw data are texts, graphics, images, audio and video data [36]. The texts can be for example e-mails, news, letters, chats and web pages [35].

Unstructured data can be just that; raw data. But it can also contain more than just raw data. Often the raw data comes packaged with additional information about it, also known as *meta data*. Meta data can for example be information about author, type or date of creation, which makes for a combination of structured and unstructured data. In such cases, the unstructured data can often look more like semi-structured data, although it would still be categorized as unstructured data [35]. Following is an example of unstructured data that only consists of raw data.

Unstructured Data
Raw Data
"Non-interpretable" audio file data

Depending on the source of the unstructured data, it may or may not contain meta data. Even if it does contain meta data, the meta data given can be different depending on the source. Meta data in one file/document may consist of just info about the type, while the meta data in another document may contain info about both type, author and date of creation, and more.

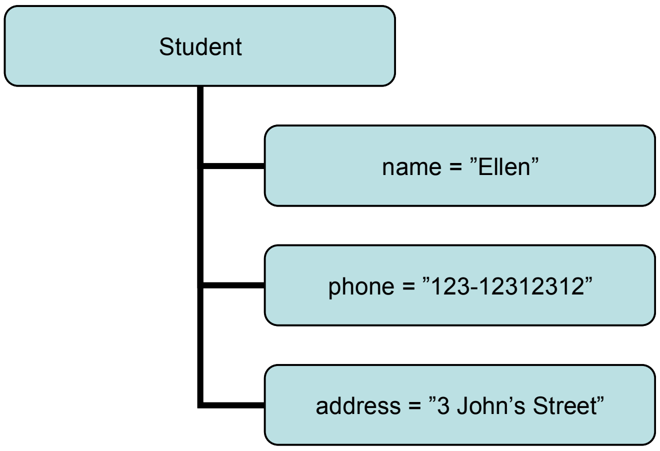
Unstructured Data	
Meta Data	**Raw Data**
Type	
mp3	"Non-interpretable" audio file data

Unstructured Data			
Meta Data			**Raw Data**
Type	**Author**	**Date of Creation YY-MM-DD**	
mp3	Author123	11-01-01	"Non-interpretable" audio file data

4.1.2 Specific types of data

4.1.2.1. Object data

Many different data-types can be seen as objects. An object in this case is of the same type as the ones you find in object-oriented programming; you save all the attributes of an object within the object itself. If you for example have a "Student" object, then it could contain information about name, phone number and address.



Many non-relational databases have a natural way of handling objects, whereas relational databases do not. The benefit of this, except for being a more natural and intuitive way of storing the data, is that you eliminate the need of JOIN-operations between the classical relational database tables. In relational databases you rely on the ability to join different tables together and in that way gather all the needed information for your operations towards the database. A object-oriented approach of storing the data means that you can easily split the different objects over partitions, enabling *sharding* for much more efficient scaling.

Sharding opens up for higher performances as the number of records in the database increases since you are not bound to have all data on one single machine or to make complex and slow solutions to split the data over partitions. This is the biggest reason why many non-relational databases scale linearly or log-linearly with the amount of data.

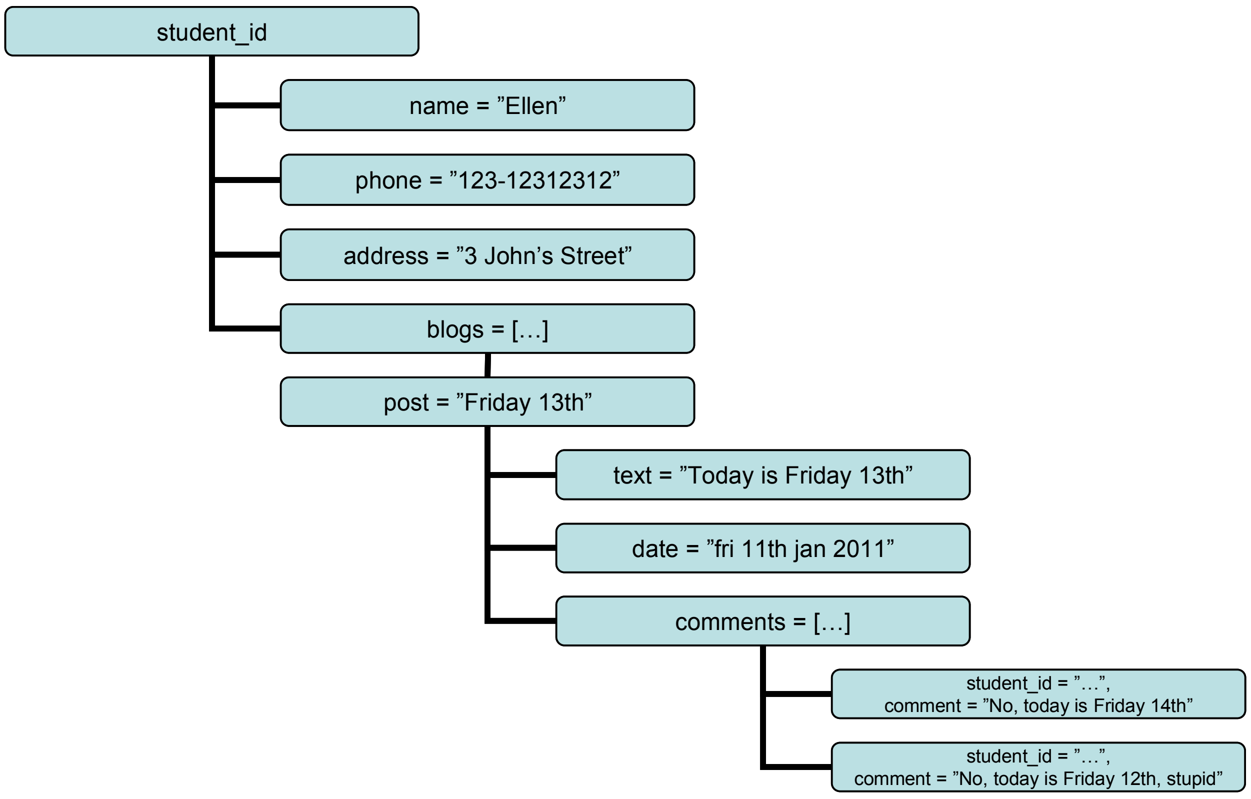
4.1.2.2. Document data

The structural philosophy of document-type data is basically the same as that of objects. You stuff all related object information into a so called "document", which in this way works just like an object.

What is special about documents though is that they never save the information of an object as a mass of data, also known as BLOB, which can be seen in key value stores. In key value stores, you cannot query the specific attributes of an object.

In document stores though, you can. This means that the term "documents" means that you have an object, and you can save more complex types of data within the different attributes of the document, and query them.

Another reason for the need for specifying that something is a document instead of just an object comes from the fact that documents often contain growing amounts of data. You could for example have a document that contains student information and blog posts.


student_id
name = ”Ellen”
phone = ”123-12312312”
address = ”3 John’s Street”
blogs = […]
post = ”Friday 13th”
text = ”Today is Friday 13th”
date = ”fri 11th jan 2011”
comments = […]
student_id = ”…”, comment = ”No, today is Friday 14th”
student_id = ”…”, comment = ”No, today is Friday 12th, stupid”

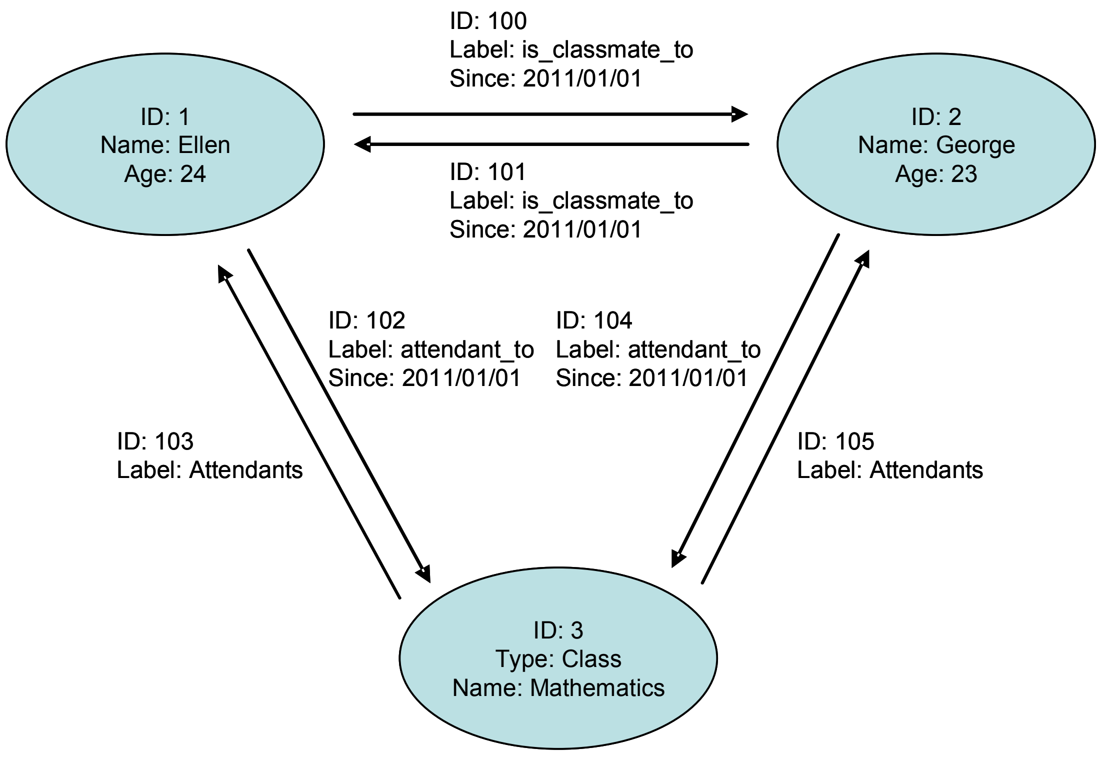
4.1.2.3. Complex data

We use complex data as a name for a type of composite data: it consists of different components, or is in other ways complicated. An example of this is the "blogs" attribute of the example above. Relational databases have no efficient ways to deal with complex types of data. The table structure they use were not built to contain complex sub-data as attributes.

Non-relational databases change this with their non-table-oriented approaches to saving data. There are different solutions to this, and one of them is the previously discussed document-type data handling.

4.1.2.4. Relation-heavy data

Some types of data rely heavily on relations to other data. An example of this is a social network where you have one person that have the relation "classmate" to another person. This opens up for new types of queries where you for example would be interested in finding a series of objects (nodes) that fulfill a certain criteria for their relations. The example below shows the principle of handling relation-heavy data-storage within a database.



The tables of relational databases cannot handle relations between posts effectively. There is no efficient solution for relations within the tables' framework, since the more relations you want to store, JOIN-operations require quickly growing amounts of data processing power [37].

4.2 Performance test

To analyze these databases we decided to perform benchmarking test using different queries and data-sources. As a first disclaimer it must be pointed out that the results obtained in these benchmarks can in no way be used to compare the different database-solutions presented. First off we had to use different data in order to make the tests work. Neo4j for example has trouble handling huge amounts of data and thus was tested on a substantially smaller amount of data. In addition the data models while trying to remain true to the original data model idea had to be slightly modified in order to satisfy certain conditions in the databases while in many cases still remaining suboptimal for the different solutions. However these results can give the reader a general idea of the performance in the different solutions and also provide a result which can be used to link the speed of the databases to increases in database size.

These tests were performed locally on a Ubuntu-system. We chose not to test Amazon S3 and Google app engine since they are only provided as internet services and thus would provide poor conditions for establishing any distinctive results.

To begin the tests we had to establish a data model suitable for doing these tests on. We decided to construct a rather simple system imitating a social network. The basic units of our data model were as follows:

- Person
- Event
- Blog post
- Comment

Where each person wrote a random number of blog posts and had a random number of other persons as their friends. The events were hosted by a single person and had a random number of invites sent which were accepted or rejected at random. In addition each blog post and event had a random amount of comments written by a friend or invited person.
To keep this random data consistent over all the test the data was generated by a program in the beginning and saved to a file. This process was then repeated with slightly different variables to create another set of data with another set of characteristics. These different data sets were then inserted into the databases and then queried with two different queries.

Q1: Find the people who has posted comments on a certain persons blog posts.
Q2: Find all people who have attended the same party as a certain person.

To conduct the tests we created Java programs for generating, inserting and querying. These can be found in Appendix B.

The program generated the data based on parameters. Here we show the size of the data desribed in a few different terms for both the small and the big data size. The big datasize were made to be twice as large as the small data size. Any deviation is due to small random data sizes within the larger controlled data sizes as mentioned before.

	Small	Big

Persons	5000	10000
Friendships	392705	792501
Events	2500	5000
Event comments	14068	27632
Blog posts	25048	49554
Blog post comments	85830	173044

5 Results

5.1 General advice on picking the right database

As the result of our research we have found three distinct points to keep in mind when choosing a NoSQL-solution for your application development.

- The attributes of the DBMSs
- The natural data-format
- Extra application development efforts

The attributes of the data correspond to section 3.2 of our research. When designing an application with a NoSQL solution in mind the attributes of the DBMS are of great concern. Each single attribute can make or break an application depending on its individual requirement. When looking at the attributes specified in the table at section 3.2.10 it is important to not neglect speed and scalability, the two main arguments for NoSQL to begin with. These attributes however are very hard to quantify since they completely depend on the use-case and any attempt to implement these attributes in this reasearch would be a source of subjectiveness and selection.

The data format corresponds to section 4.1 of our reasearch. The natural data format of the database solution may sometimes play a critical role when choosing your NoSQL-solution. Any application development benefits greatly by a storage solution which requires no serialization of data as well as featuring fetching of data corresponding well to the use cases of the application.

Depending on the type of data you want to store, you might or might not want to look into transitioning over to a NoSQL database.

5.1.1 Does structured data benefit from NoSQL use?

Flexibility
For structured data a transition to NoSQL databases is in general not necessary. Although NoSQL databases do not have any problem handling structured data, the SQL databases work well. They are ,after all, designed to handle this kind of data.

Performance
NoSQL databases in general scale much more efficiently than SQL databases [37]. If you are going to store very big amounts of data, then you would want to consider transitioning to a NoSQL database in order to get effective scaling.

5.1.2 Does semi-structured and unstructured data benefit from NoSQL use?

Flexibility
Yes, they do. Semi-structured and unstructured data does not fit well into the tables of relational databases because of the following reasons:

- *Missing fields*. Some fields may be missing out.
- *Loose or no structure*. They do not follow a fixed structure.
- *Raw data handling*. Raw data is hard or impossible for a database to interpret and the data is even less able to be sorted into a table and still make sense. There - in general - shouldn't be any kind of "expectations" on the raw data from the database.

This shows that the high requirements on the data from the relational databases are unnecessary and creates a situation where you try to fit the data to the database, instead of fitting the database to the data.

It should be mentioned though that one way to handle these problems is to save the data as BLOBs in your RDBMS. This way you can store the data, but this is however only a storage solution. When you store the data as a BLOB, the database cannot interpret the data in the BLOB as anything else than a series of bytes.

Performance
Yes, they do benefit from NoSQL use.

- *Scaling*. We have the general fact that non-relational databases scale better than relational databases when you are dealing with increasing amounts of data [37].
- *Complexity*. The complexity increases when trying to fit non-structured data into a relational database. While the complexity of the relational databases by themselves are already high, this makes for an unintuitive, hard-to-understand and possibly slower database as you either have to increase the number of columns as you try to fit the new data into the database or when you try to fit the raw data into the database.

5.1.3 Additional comments 1: The RDBMS's requirements on the data

In order for a unique instance of data, a record, to fit into a relational database, you need to in before-hand have information about the elements of the instance/record in order to build the required table. The columns need to be defined in the beginning, and ever since that moment, all the instances of data needs to fit into that table.

This makes for a complex, difficult and slow database environment to work with if the data is not structured [37]. This way of handling the data would also seem rather unintuitive and unnecessary.

The only time that relational databases provide better flexibility is when you can not foresee what kinds of questions will be queried to the database in the future [38]. In this case, the strict structure of the relational database makes it possible to use SQL with combinations of elements/attributes that you did not even know when you designed the database.

5.1.4 Additional comments 2: The ACID aspect

RDBMS's usually follow the ACID semantics very strongly, and hence make a very good way to store data that requires security, fail-safe operations and consistent data. The typical example of data with these requirements are the ones saved in banks, where errors can not be accepted. As mentioned though, some NoSQL databases puts an emphasis on providing ACID semantics, but usually do so with a limited feature set.

5.1.5 Attribute summary table of our five investigated databases

	Amazon S3	**Google App Engine**	**CouchDB**	**MongoDB**	**Neo4j**
Atomicity	Multi-key	Single-key	Full	Document level	Full
Consistency	Strong	Eventual	Strong (master-slave only)	Eventual	Strong
Isolation	No	Yes	Yes	No	Yes
Durability	Yes	Yes	Yes	Yes	Yes
Queries	Primary keys	Yes	Primary keys + static views	Yes	Transitional queries
Map-reduce	No	Yes	Yes	Yes	Not applicable
CAP - Consistency	No	Yes	Yes	No	N/A
CAP - Availability	Yes	No	No	Yes	N/A
CAP - Partition tolerance	Yes	Yes	Yes	Yes	N/A
Replication	Yes	Yes	Yes	Yes	No**
Sharding	Manual*	Not applicable*	Manual or by plug-in	Yes	Yes

* The underlying technology uses sharding
** But they are working on it. Check out the High Availability Cluster project (HA)

Consequences for application development

As NoSQL-solutions in most cases only provide limited support for error-correction, consistency and recoverability as discussed in 3.2, this forces application developers to undertake a different approach when dealing with these issues. The databases must be examined together with the application in order to establish which types of errors are possible and whether they are crucial for the application. These problems must then be handled in the application-layer of the program. This creates an additional task for the developer and this task might be very hard to solve especially when making an application which has very tight restrictions regarding data storage properties. There are both benefits and drawbacks by using a database which forces error preventing code to be written in the application layer.

The main benefit is the increase in speed as well as scalability the absense of several features and attributes in NoSQL DBMSs allows for this. This has been the strongest argument from the NoSQL-movement for switching to NoSQL.[4][27]

5.2 Performance test results

Benchmarking

As indicated before the performance test did not include Google app Engine or Amazon S3

The performance test results are provided in Appendix A. The small and big keywords in each header provides which data set was used and Q1 and Q2 reflects whether query 1 or query 2 was used. The numbers below each header is the amount of milliseconds it took to complete each query. It is immidiately observable that MongoDB was faster than Neo4j which in turn was faster than CouchDB and all the databases fared well with increasing data size.

As for the required time to accomplish the different queries, these are the main results that were given:

Average time (ms)	**MongoDB**	**Neo4j**	**CouchDB**
Query nr 1 / BIG	10,1	95,6	295,05
Query nr 2 / BIG	44,25	103,05	1867,6
Query nr 1 / SMALL	6,25	103,85	204,65
Query nr 2 / SMALL	22,45	85,9	956,3
Query nr 1 / MULTIPLIER	1,616	0,9205584978	1,4417297826
Query nr 2 / MULTIPLIER	1,8829787234	1,1996507567	1,9529436369

The multiplier values in the result describes the percentual increase in query time resulting from a twice as large data size as mentioned in section 4.2.

For more data and graphs please see Appendix A.

6 Discussion

From the research we can conclude that each of the DBMSs certainly has its uses and problems. The usability of the databases is very much application and data model dependent. An application using very little dynamic filtering might favor CouchDB while an application having to process deep relational problems would benefit from using Neo4J. In order to make it simple we have come up with these different use cases for our three main databases.

Neo4j – relational querying
CouchDB - static filtering on dynamic data
MongoDB - dynamic filtered data

With this in mind the test data speaks very clearly. In the test queries we used dynamic filtering which is very similar to relational querying but in most aspects completely different to static filtered data. This caused CouchDB especially in test 2 to receive far worse benchmarks than the other databases. It must however be pointed out that these databases such as MongoDB which handle data similar to a standard RDBMS still drops some properties in favor of speed.

All databases had good results for their multiplier as seen in 5.2 and Appendix A. However Neo4j provided remarkable results in that regard. It increases slightly in execution time in one query and decreases in the other. The reason for this must be its graph design. The graph design allows Neo4j to ignore the size of the total data stored in the databased. The data we provided did simply increase the amount of persons and events. The amount of blogposts per person, friends per person, invites per event stayed the same. This gave Neo4j an edge since it never had to filter through all the data but just the data that did not increase in size.

CouchDB showed good results in a totally different regard. To achive the goal of both of the querys with CouchDB we had to statically filter the data in CouchDB, send it to our querying application and then in the application extract the vital information we needed. This meant that CouchDB did filtering for all persons in both querys and sent to the application. With that in mind CouchDB seems fast. The reason CouchDB can achieve that is because it caches its static filterings.

When looking at the knowledge we have extracted from our research we can see that all three database solutions caters to completely different scenarios. You could say that NoSQL is not about a specific issue but NoSQL is about custom tailored databases solving very specific problems.

7 Conclusions

Different types of data

NoSQL DBMS's were created to handle more loosely defined types of data, where you for example have growing amounts of data within a single object. In this report we have both looked at a few specific databases - and compared them in different aspects – and investigated how different types of data can benefit from the use of a NoSQL DBMS from the aspects of flexibility and performance.

The attribute comparison for our specific databases is presented under the 'Results' chapter.

As for the different types of data and how they can benefit from the use of a NoSQL DBMS, this is what we concluded:

Structured data

Flexibility
From a flexibility perspective, structured data in general doesn't specifically benefit from the use of a NoSQL DBMS.

Performance
From a performance perspective, structured data can benefit from the scalability strenghts of NoSQL DBMS if the amount of data stored grows big.

Semi-structured and unstructured data

Flexibility
From a flexibility perspective, semi-structured and unstructured data does benefit from the use of a NoSQL DBMS, for the following reasons: some fields may be missing out, there is no coherent structure or the data contains raw data.

Performance
From a performance perspective, semi-structured and unstructured data does benefit from the use of a NoSQL DBMS, for the following reasons: scaling is more effective and you avoid the increasing complexity of trying to fit unstructured data into a relational database, which possibly will make the database slower.

The performance test

From our test results we understood that CouchDB was not made for dynamic queries filtering on anything else than the primary key. We have also learned that to truly test the databases you have to take a far more wide range of factors into aspect. We can however see that the databases scale quite well with the increasing amount of data even while doing queries heavy in data processing. However our tests did not contain tests on sharded and replicated data and did not test inserts and selections simultaneously. This could certainly be done but is dependent on additional hard controlled parameters such as network latency and order of inserts and selections.

8 Future research

As mentioned in section 2.1 in the methods chapter, the performance test we did by far doesn't cover all the different aspects and twists of the subject. We only tested one type of data, from a few aspects, on three databases. There are lots and lots of additional tests that can be done. For future research we suggest testing:

- Other types of data
- Other databases
- Other functions of the databases, such as sharding and replication
- Other aspects of the performance, such as consistency and availability
- Other types of queries and other types of data models

Citations and sources

[1]: 2011-04-30, http://docs.amazonwebservices.com/AmazonS3/latest/gsg/

[2]: 2011-09-21, http://www.mongodb.org/

[3]: 2011-05-01, http://couchdb.apache.org/docs/intro.html

[4]: List of NoSQL databases http://nosql-database.org/

[5]: Google Trends NoSQL, http://www.google.com/trends?q=nosql

[6]: Michael Stonebreaker, 2010, SQL Databases v. NoSQL Databases, Communications of the ACM April Vol 53 NO 4

[7]: Rick Catell, 2011, Scalable SQL and NoSQL Data Stores, ACM SIGMOD Record Volume 39 Issue 4

[8]: Daniel J. Abadi, Peter A. Boncz, Stavros Harizopoulos, 2009, Column-oriented Database Systems, Proceedings of the VLDB Endowment Volume 2 Issue 2 August

[9]: Natalia Söderberg, Jan Eriksson, 2010, Utredning av NoSQL-databaser för Sogeti I Gävle, Högskolan i Gävle Akademin för teknik och miljö

[10]: Orend Kai, 2010, Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistance Layer, Technical University Munich Faculty of Informatics

[11]: MongoDB, http://www.mongodb.org/

[12]: CouchDB, http://couchdb.apache.org/

[13]: Renzo Angles, Claudio Gutierrez, 2008, ACM Computing Surveys (CSUR) Volume 40 Issue 1 February

[14]: Neo4j, http://neo4j.org/

[15]: Principles of transaction-oriented database recovery, Theo Haerder, Andreas Reuter, ACM Computing Surveys (CSUR) Volume 15 Issue 4, December 1983

[16]: Google Groups Thread: Is db.put() atomic, http://groups.google.com/group/google-appengine/browse_thread/thread/2ce6e772bb9735d5?pli=1 , 2011-04-27

[17]: MongoDB atomic operations, http://www.mongodb.org/display/DOCS/Atomic+Operations, 2011-04-27

[18]: CouchDB Consistancy, http://guide.couchdb.org/draft/consistency.html, 2011-04-27

[19]: Neo4j Transactions, http://docs.neo4j.org/chunked/snapshot/transactions.html, 2011-04-27

[20]: Google App Engine Choosing a datastore, http://code.google.com/intl/sv-SE/appengine/docs/python/datastore/hr/, 2011-04-27

[21]: MongoDB Advanced Queries, http://www.mongodb.org/display/DOCS/Advanced+Queries#AdvancedQueries-%7B%7Bgroup%28%29%7D%7D

[22]: Jeffrey Dean, Sanjay Ghemawat, 2004, MapReduce: Simplified Data Processing on Large Clusters, OSDI’04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation Volume 6

[23]: Seth Gilbert, Nancy Lynch, 2002, Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, Seth Gilbert, Nancy Lynch, ACM SIGACT News,

Volume 33 Issue 2, pg. 51-59

[24]: Visual guide to NoSQL systems, http://blog.nahurst.com/visual-guide-to-nosql-systems, 2011-05-01

[25]: The common principles behind the NoSQL alternative, http://blog.gigaspaces.com/2009/12/15/the-common-principles-behind-the-nosql-alternatives/ , 2011-09-22

[26]: Marc Seeger, 2009, Key – Value Stores: a practical overview, Hochschule der Medien Computer Science, Stuttgart Germany

[27]: SQL vs. NoSQL, Daniel Bartholomew, Linux Journal, Issue 195, 2010

[28]: Jan Lehnardt at the FLOSS Weekly podcast nr 36: CouchDB

[29]: Serge Abiteboul, info gathered 2011-04-19, *Querying semi-structured data*

[30]: Michael Stonebraker, 2010, *SQL Databases v. NoSQL Databases*

[31]: Christopher C. Shilakes and more, 2011-04-19, p15, http://ikt.hia.no/perep/eip_ind.pdf

[32]: Noel Yuhanna, 2010, *Today's Challenge in Government: What to do with Unstructured Information and Why Doing Nothing Isn't An Option*

[33]: Denise L. Draper, 2003, US Patent, *Method and apparatus for storing semi-structured data in a structured manner*

[34]: Dallan Quass and more, 1997, *Querying Semistructured Heterogeneous Information*

[35]: Robert Blumberg and more, 2003, *The problem with unstructured data*

[36]: Li Wei and more, 2010, *A tetrahedral data model for unstructured data management*

[37]: Neal Leavitt, 2010, *Will NoSQL Databases Live Up to Their Promise*

[38]: Natalja Söderberg and more, 2010, p45, *Utredning av NoSQL-databaser*

Appendix

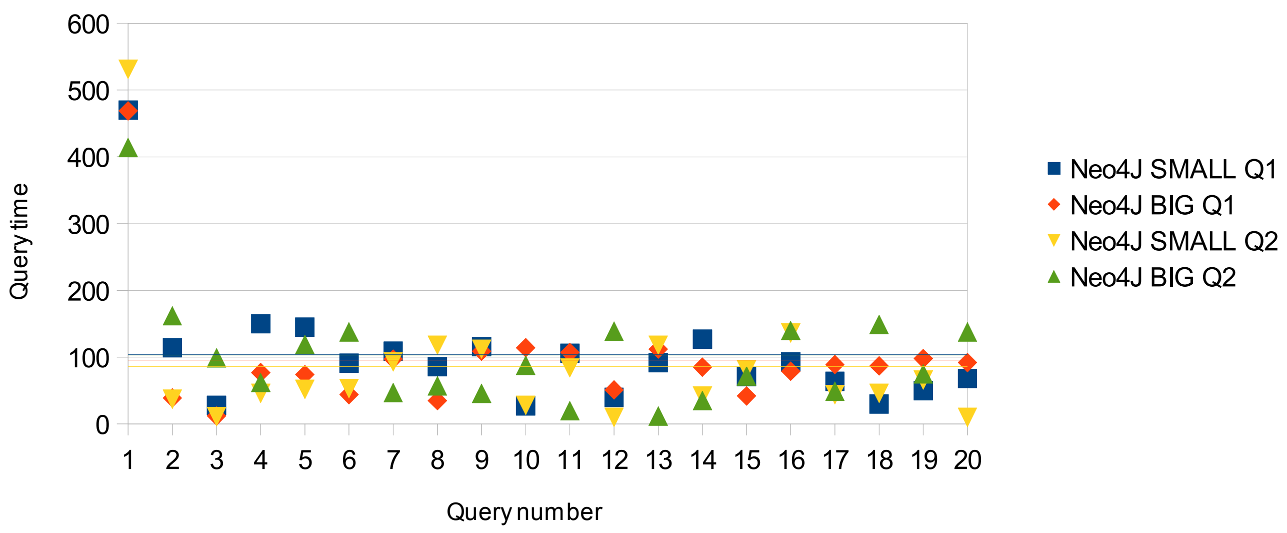
Appendix A – Results

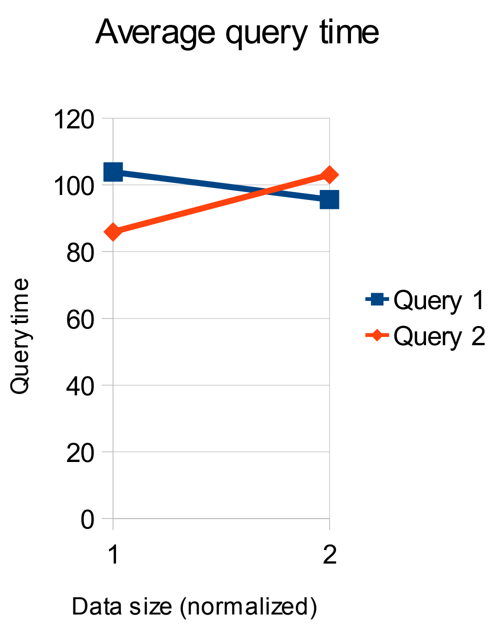
The results are divided by each DBMS and is first represented by the raw data collected. This is each querys total runtime to finish in milliseconds. Then graphs are presented visualising the increase in query time when the data size grows. In addition graphs are provided for only run 2-20 of each query.

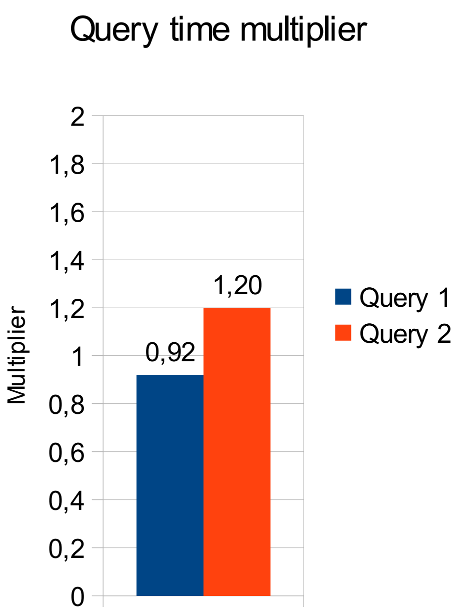
A1. Neo4J

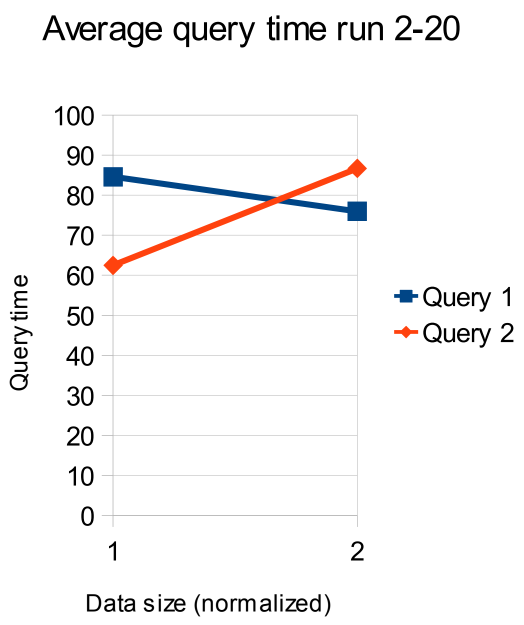
Neo4J Q1 Small	Neo4J Q1 Big	Neo4J Q2 Small	Neo4J Q2 Big
470	469	531	414
114	39	37	162
28	12	11	99
150	77	46	62
145	74	52	119
91	44	53	138
109	97	93	47
86	35	118	57
116	109	112	46
27	114	28	88
106	107	84	20
40	51	10	139
92	112	118	12
127	85	42	35
71	42	81	72
93	79	136	140
64	89	44	49
30	87	46	149
50	98	66	75
68	92	10	138

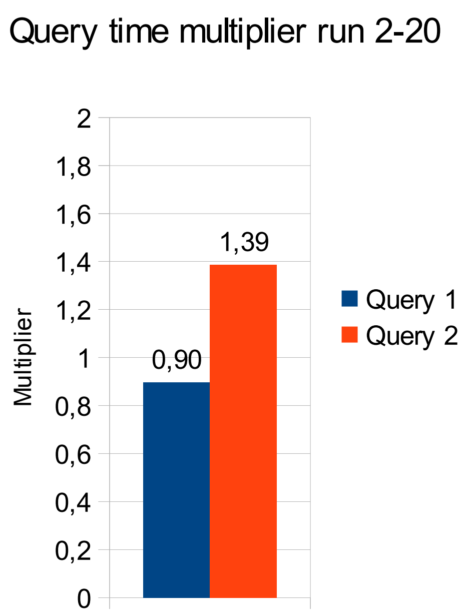
Query times




Average query time
120
100
80
60
40
20
0
Query time
1
2
Data size (normalized)
Query 1
Query 2

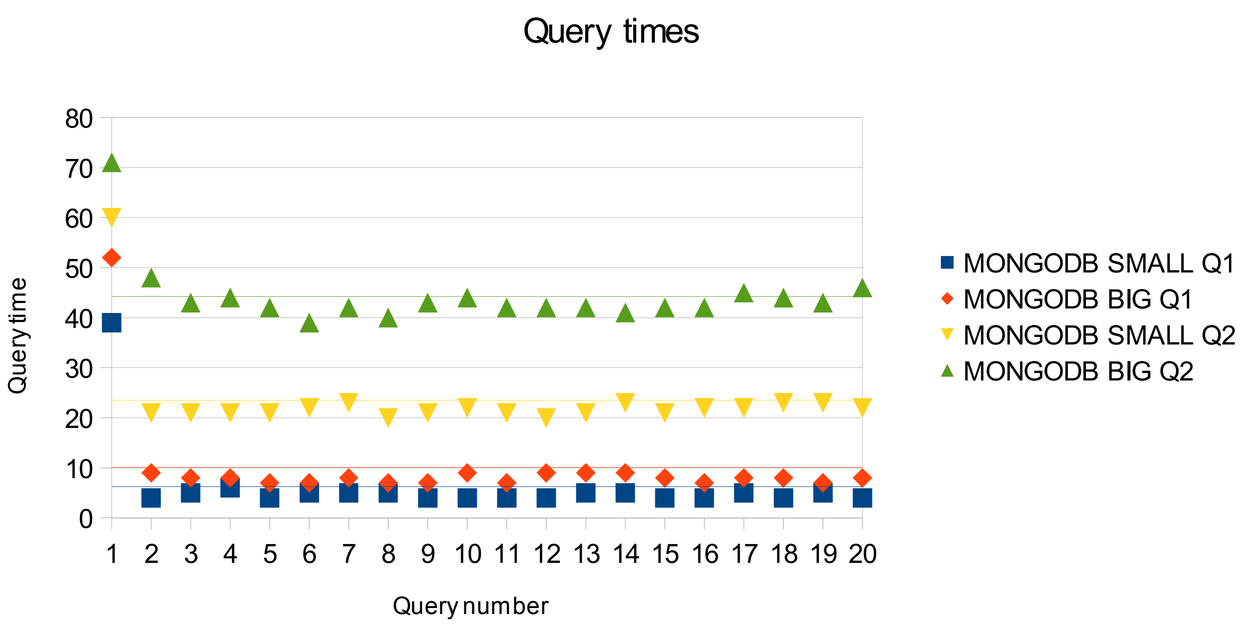

Query time multiplier
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Multiplier
0,92
1,20
Query 1
Query 2

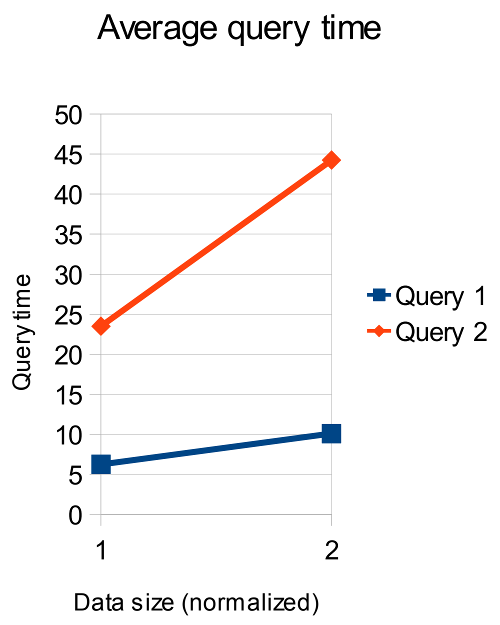

Average query time run 2-20
100
90
80
70
60
50
40
30
20
10
0
Query time
1
2
Data size (normalized)
Query 1
Query 2

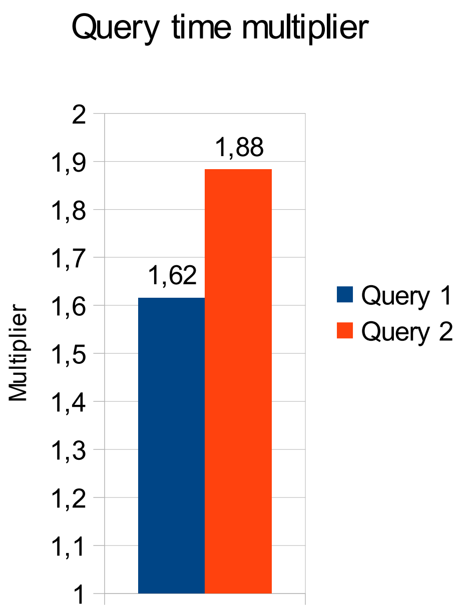

Query time multiplier run 2-20
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Multiplier
0,90
1,39
Query 1
Query 2

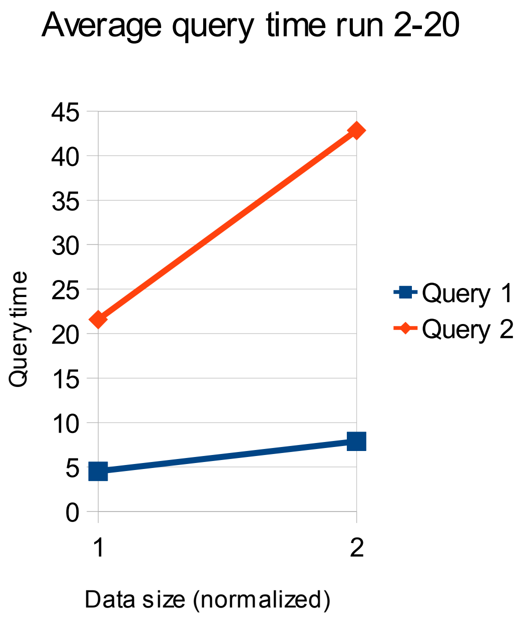
A2. MongoDB

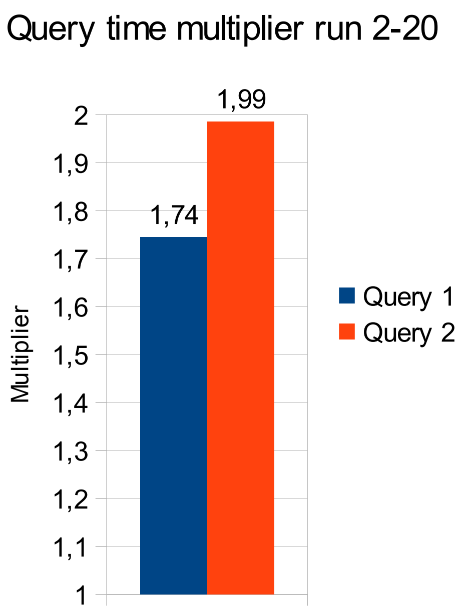
MongoDB Q1 Small	MongoDB Q1 Big	MongoDB Q1 Small	MongoDB Q2 Big
39	52	60	71
4	9	21	48
5	8	21	43
6	8	21	44
4	7	21	42
5	7	22	39
5	8	23	42
5	7	20	40
4	7	21	43
4	9	22	44
4	7	21	42
4	9	20	42
5	9	21	42
5	9	23	41
4	8	21	42
4	7	22	42
5	8	22	45
4	8	23	44
5	7	23	43
4	8	22	46




Average query time
Query time
50
45
40
35
30
25
20
15
10
5
0
1
2
Data size (normalized)
Query 1
Query 2

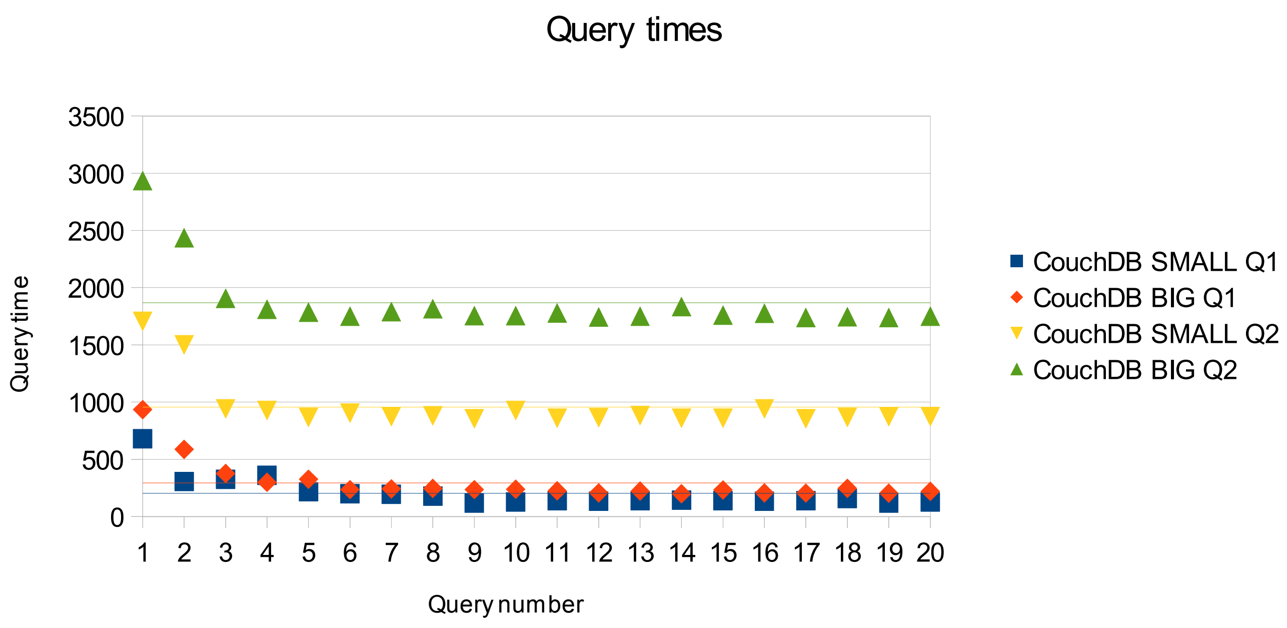

Query time multiplier
Multiplier
2
1,9
1,8
1,7
1,6
1,5
1,4
1,3
1,2
1,1
1
1,88
1,62
Query 1
Query 2

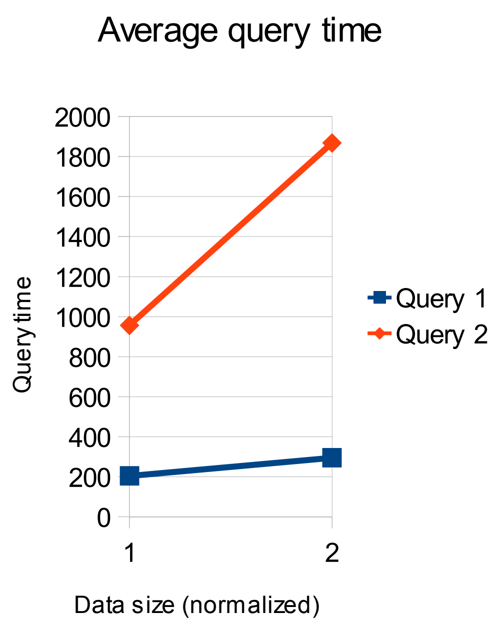

Average query time run 2-20
Query time
45
40
35
30
25
20
15
10
5
0
1
2
Data size (normalized)
Query 1
Query 2

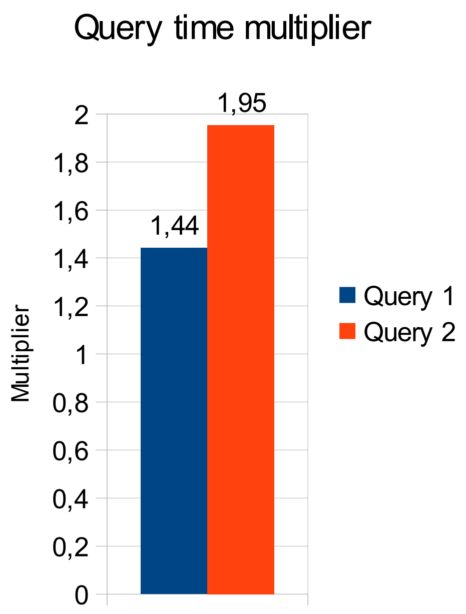

Query time multiplier run 2-20
Multiplier
2
1,9
1,8
1,7
1,6
1,5
1,4
1,3
1,2
1,1
1
1,99
1,74
Query 1
Query 2

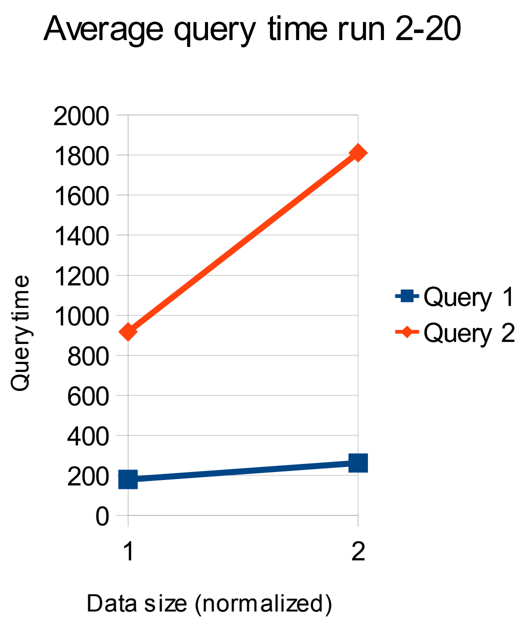
A3. CouchDB

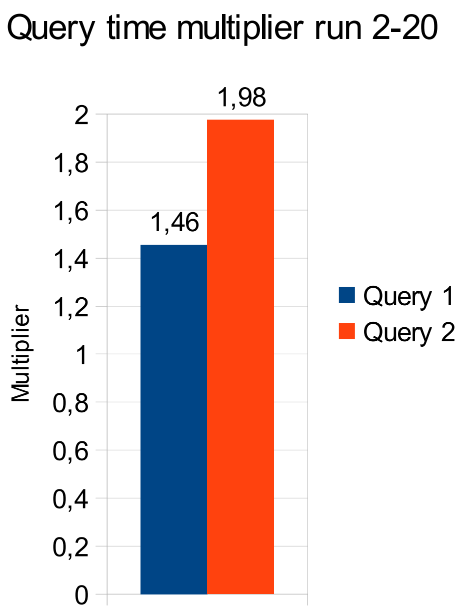
CouchDB Q1 Small	CouchDB Q1 Big	CouchDB Q2 Small	CouchDB Q2 Big
681	935	1707	2935
306	588	1502	2435
327	377	941	1906
362	300	927	1812
218	327	867	1785
199	236	904	1750
196	242	875	1789
180	246	880	1815
119	236	856	1755
127	239	927	1754
139	225	861	1779
135	208	864	1743
140	223	883	1750
145	200	861	1834
139	233	860	1759
136	207	941	1776
139	206	856	1740
158	247	866	1745
119	205	873	1740
128	221	875	1750




Average query time
Query time
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1
2
Data size (normalized)
Query 1
Query 2


Query time multiplier
Multiplier
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1,44
1,95
Query 1
Query 2


Average query time run 2-20
Query time
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1
2
Data size (normalized)
Query 1
Query 2


Query time multiplier run 2-20
Multiplier
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1,46
1,98
Query 1
Query 2

Appendix B – Program Code

Dependencies: org.apache.http , org.json

```
import java.util.*;
import java.io.*;


public class DataGenerator
{
	static String fileName = "data1.ser";
	static char [] chars = 
{'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O
','P','Q','R','S','T','U','W','X','Y','Z'};
	static int numberOfPersons = 5000;
	static int numberOfEvents=2500;
	static int numberOfEventInvitesMin=10;
	static int numberOfEventInvitesMax=25;
	static double numberOfEventCommentsPerUserMin=0.0;
	static double numberOfEventCommentsPerUserMax=0.7;
	static int numberOfBlogPostsMin=0;
	static int numberOfBlogPostsMax=10;
	static double numberOfBlogPostCommentsPerUserMin=0.0;
	static double numberOfBlogPostCommentsPerUserMax=0.05;
	static Calendar personCalMin = new GregorianCalendar(1950,0,1);
	static Calendar personCalMax = new GregorianCalendar(2005,12,31);
	static Calendar eventCalMin = new GregorianCalendar(2000,0,1);
	static Calendar eventCalMax = new GregorianCalendar(2010,12,31);
	static Calendar blogPostCalMin = new GregorianCalendar(2000,0,1);
	static Calendar blogPostCalMax = new GregorianCalendar(2010,12,31);
	static int ownInitiatedFriendsMin=40;
	static int ownInitiatedFriendsMax=200;
	static int friendsNormalStandardDeviation=200;

	static final int UNIFORM_DISTRIBUTION=0;
	static final int NORMAL_DISTRIBUTION=1;
	static int friendshipType=UNIFORM_DISTRIBUTION;
	static int eventInvitesType=UNIFORM_DISTRIBUTION;
	static double attendanceChance=0.5;

	static Random r = new Random();
	/**
	 * @param args
	 */
	public static void main(String[] args)
	{

		Person [] persons = new Person [numberOfPersons];
		System.out.println("GENERATING PERSONS");
		for(int i=0;i<persons.length;i++)
		{
			persons[i]=new Person();
			persons[i].id=i;
			persons[i].name = generateString(3,20);
			persons[i].adress = generateString(10,30);
			persons[i].birthdate = generateCalendar(personCalMin,personCalMax);
			persons[i].country = generateString(5,15);
		}
		System.out.println("PERSONS DONE");
		System.out.println("GENERATING FRIENDSHIPS");
		for(int i=0;i<persons.length;i++)
		{
			persons[i].totalOwnInitiatedFriends = 
generateInt(ownInitiatedFriendsMin,ownInitiatedFriendsMax);
		}
		for(int i=0;i<ownInitiatedFriendsMax;i++)
		{
			//System.out.println(i);
			for(int j=0;j<persons.length;j++)
			{
```

```
                                        if(persons[j].totalOwnInitiatedFriends<i)
                                        {
                                                makeFriend(persons[j],j,persons);
                                        }
                                }
                        }
                        for(int i=0;i<persons.length;i++)
                        {
                                persons[i].finalizeFriends();
                        }
                        System.out.println("FRIENDSHIPS DONE");
                        System.out.println("GENERATING EVENTS");
                        Event [] events = new Event[numberOfEvents];
                        for(int i=0;i<events.length;i++)
                        {
                                events[i]=new Event();
                                events[i].id=i;
                                events[i].eventInfo=generateString(10,200);
                                events[i].eventDate=generateCalendar(eventCalMin,eventCalMax);
                                //Random host
                                int hostIndex=(int)(r.nextDouble()*persons.length);
                                events[i].host=persons[hostIndex];
                                //Random amount of guests
                                int guestAmount=generateInt(numberOfEventInvitesMin,numberOfEventInvitesMax);
                                for(int j=0;j<guestAmount;j++)
                                {
                                        makeInvite(events[i],hostIndex,persons);
                                }
                                events[i].finalizeInvites(attendanceChance);

                                int commentAmount=(int)
(guestAmount*generateDouble(numberOfEventCommentsPerUserMin,numberOfEventCommentsPerUserMax));
                                Comment [] comments = new Comment[commentAmount];
                                events[i].comments=comments;
                                for(int j=0;j<commentAmount;j++)
                                {
                                        comments[j]=makeEventComment(events[i]);
                                }

                        }
                        System.out.println("EVENTS DONE");
                        System.out.println("GENERATING BLOGPOSTS");
                        long totalComments =0;
                        for(int i=0;i<persons.length;i++)
                        {

                                int blogPostAmount = generateInt(numberOfBlogPostsMin,numberOfBlogPostsMax);
                                BlogPost [] blogPosts = new BlogPost[blogPostAmount];
                                persons[i].blogPosts=blogPosts;
                                if(i%1000==0)System.out.println(i+"-"+totalComments);
                                for(int j=0;j<blogPosts.length;j++)
                                {
                                        blogPosts[j]=new BlogPost();
                                        blogPosts[j].content=generateString(100,1000);
                                        blogPosts[j].publishDate=generateCalendar(blogPostCalMin,blogPostCalMax);
                                        int commentAmount=(int)
(persons[i].friends.length*generateDouble(numberOfBlogPostCommentsPerUserMin,numberOfBlogPostCommentsPerUs
erMax));
                                        Comment [] comments = new Comment[commentAmount];
                                        blogPosts[j].comments=comments;
                                        for(int k=0;k<commentAmount;k++)
                                        {
                                                comments[k]=makeBlogPostComment(blogPosts[j],persons[i].friends);
                                                totalComments++;
                                        }
                                }
                        }
                        System.out.println("BLOGPOSTS DONE");
                        System.out.println("PREPARING FOR SERIALIZATION");
                        DataHolder dh = new DataHolder();
                        dh.persons=persons;
                        dh.events=events;
                        dh.prepareForSerialization();
                        System.out.println("PREPARATION DONE");
                        System.out.println("STARTING SERIALIZATION");
```

```
		try
		{
			ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("data1.ser"));
			oos.writeObject(dh);
			oos.close();
			/*dh=null;
			persons=null;
			events=null;
			ObjectInputStream ois = new ObjectInputStream(new FileInputStream("data1.ser"));
			DataHolder dh2 = (DataHolder)ois.readObject();
			oos = new ObjectOutputStream(new FileOutputStream("data2.ser"));
			oos.writeObject(dh2);
			oos.close();*/


			/*PrintWriter pw = new PrintWriter(new FileOutputStream(new File("data1.ser")));

			dh.serialize(pw);
			pw.close();
			dh=null;
			persons=null;
			events=null;
			Scanner scan = new Scanner(new File("data1.ser"));
			DataHolder dh2 = new DataHolder();
			dh2.unserialize(scan);
			pw=new PrintWriter(new FileOutputStream(new File("data2.ser")));
			dh2.serialize(pw);
			pw.close();*/


		}
		catch(IOException io)
		{
			System.out.println("ERROR IO EXCEPTION WHEN WRITING TO FILE");
			io.printStackTrace();
		}
		catch(Exception e)
		{
			e.printStackTrace();
		}
		System.out.println("SERIALIZATION DONE");
	}

	public static String generateString(int minChar, int maxChar)
	{
		char [] string = new char [(int)(r.nextDouble()*(maxChar-minChar))+minChar];
		for(int i=0;i<string.length;i++)
		{
			string [i]=chars[(int)(r.nextDouble()*chars.length)];
		}
		return new String (string);
	}
	public static int generateInt(int min,int max)
	{
		return (int)(r.nextDouble()*(max+1-min))+min;
	}
	public static double generateDouble(double min, double max)
	{
		return r.nextDouble()*(max-min)+min;
	}
	public static Calendar generateCalendar(Calendar min, Calendar max)
	{
		Calendar cal = Calendar.getInstance();
		cal.setTimeInMillis((long)(min.getTimeInMillis()+r.nextDouble()*(max.getTimeInMillis()-
min.getTimeInMillis())));
		return cal;
	}
	public static void makeFriend(Person person,int personIndex,Person [] personStack)
	{
		int count =0;

		while(true)
		{
			count++;
			if(count>100)
```

```
				{
					System.out.println("WARNING TROUBLE FINDING FRIENDS");
				}
				if(friendshipType==UNIFORM_DISTRIBUTION)
				{
					Person friend = personStack[(int)(r.nextDouble()*personStack.length)];
					if(friend!=person && !person.areFriends(friend))
					{
						person.addFriend(friend);
						friend.addFriend(person);
						break;
					}
				}
				else if(friendshipType==NORMAL_DISTRIBUTION)
				{
					int indexModifier = (int)(r.nextGaussian()*friendsNormalStandardDeviation);
					int friendIndex = (personIndex+indexModifier)%personStack.length;
					if(friendIndex<0)friendIndex = personStack.length+friendIndex;

					Person friend = personStack[friendIndex];

					if(friend!=person && !person.areFriends(friend))
					{
						person.addFriend(friend);
						friend.addFriend(person);
						break;
					}
				}
			}
		}
		public static void makeInvite(Event event,int hostIndex,Person [] personStack)
		{
			int count =0;
			while(true)
			{
				count++;
				if(count>100)
				{
					System.out.println("WARNING TROUBLE FINDING INVITES");
				}
				if(eventInvitesType==UNIFORM_DISTRIBUTION)
				{
					Person guest = personStack[(int)(r.nextDouble()*personStack.length)];
					if(guest!=personStack[hostIndex] && !event.isInvited(guest))
					{
						event.invite(guest);
						break;
					}
				}
				else if(eventInvitesType==NORMAL_DISTRIBUTION)
				{
					int indexModifier = (int)(r.nextGaussian()*friendsNormalStandardDeviation);
					int guestIndex = (hostIndex+indexModifier)%personStack.length;
					if(guestIndex<0)guestIndex = personStack.length+guestIndex;
					Person guest = personStack[guestIndex];
					if(guest!=personStack[hostIndex] && !event.isInvited(guest))
					{
						event.invite(guest);
						break;
					}
				}
			}
		}
		public static Comment makeEventComment(Event e)
		{
			Comment c = new Comment();
			c.comment=generateString(10,150);
			long dayInMs=1000l*60l*60l*24l;
			Calendar calMin = Calendar.getInstance();
			Calendar calMax = Calendar.getInstance();
			calMin.setTimeInMillis(e.eventDate.getTimeInMillis()-dayInMs*10l);
			calMax.setTimeInMillis(e.eventDate.getTimeInMillis()+dayInMs*10l);
			c.date=generateCalendar(calMin,calMax);
			c.person=e.invites[(int)(e.invites.length*r.nextDouble())];
			return c;
```

```
	}
	public static Comment makeBlogPostComment(BlogPost bp, Person [] personStack)
	{
		Comment c = new Comment();
		c.comment=generateString(10,150);
		long dayInMs=1000l*60l*60l*24l;
		Calendar calMin = Calendar.getInstance();
		Calendar calMax = Calendar.getInstance();
		calMin.setTimeInMillis(bp.publishDate.getTimeInMillis());
		calMax.setTimeInMillis(bp.publishDate.getTimeInMillis()+dayInMs*30l);
		c.date=generateCalendar(calMin,calMax);
		c.person=personStack[(int)(personStack.length*r.nextDouble())];
		return c;
	}

}

public class CouchDBQuery {
	public static HttpClient client = new DefaultHttpClient();
	public static int portNumber = 5984;

	public static void query1(String databaseName) throws Exception
	{
		for(int h=0;h<20;h++)
		{
			long time = System.currentTimeMillis();
			String personID="p"+(500+100*h);
			HttpGet get = new
HttpGet("http://127.0.0.1:"+portNumber+"/"+databaseName+"/_design/aView/_view/attending");
			get.setHeader("Content-Type", "application/json");
			HttpResponse response = client.execute(get);
			InputStream instream = response.getEntity().getContent();
			File bufferFile = new File("bufferFile");
			OutputStream outstream = new FileOutputStream(bufferFile);
            try {
                int l;
                byte[] tmp = new byte[2048];
                while ((l = instream.read(tmp)) != -1) {
                    outstream.write(tmp, 0, l);
                }
            } finally {
                instream.close();
                outstream.close();
            }
            System.out.println("READ TO BUFFER: "+(System.currentTimeMillis()-time));

            InputStream is =new FileInputStream(bufferFile);
			String total="";
			byte [] buffer  = new byte[1024*1024];
			while (true)
			{
				//System.out.println(total.length());
				int read = is.read(buffer);
				if(read!=-1)total=total + new String(buffer,0,read);
				else break;
			}
			is.close();


			System.out.println("BUFFER READ: "+(System.currentTimeMillis()-time));
			JSONObject wrapper = new JSONObject(total);
			System.out.println("CONVERTED TO OBJECT: "+(System.currentTimeMillis()-time));
			JSONArray objects = wrapper.getJSONArray("rows");
			HashSet<String> friendsFromEvents = new HashSet<String>();
			for(int i=0;i<objects.length();i++)
			{
				JSONObject object = objects.getJSONObject(i);
				JSONArray attending = object.getJSONArray("value");
				for(int j=0;j<attending.length();j++)
				{
					if(attending.getString(j).equals(personID))
					{
						for(int k=0;k<attending.length();k++)
						{
							friendsFromEvents.add(attending.getString(k));
```

```
                                                }
                                        }
                                }
                        }
                        System.out.println("DONE: "+(System.currentTimeMillis()-time));
                }
        }
        public static void query2(String databaseName) throws Exception
        {
                for(int h=0;h<20;h++)
                {


                        long time = System.currentTimeMillis();
                        String personID="p"+(500+100*h);
                        HttpGet get = new
HttpGet("http://127.0.0.1:"+portNumber+"/"+databaseName+"/_design/aView/_view/comments");
                        get.setHeader("Content-Type", "application/json");
                        HttpResponse response = client.execute(get);
                        InputStream instream = response.getEntity().getContent();
                        File bufferFile = new File("bufferFile");
                        OutputStream outstream = new FileOutputStream(bufferFile);
            try {
                int l;
                byte[] tmp = new byte[2048];
                while ((l = instream.read(tmp)) != -1) {
                    outstream.write(tmp, 0, l);
                }
            } finally {
                instream.close();
                outstream.close();
            }
            System.out.println("READ TO BUFFER: "+(System.currentTimeMillis()-time));

            InputStream is =new FileInputStream(bufferFile);
                        String total="";
                        byte [] buffer  = new byte[1024*1024];
                        while (true)
                        {
                                //System.out.println(total.length());
                                int read = is.read(buffer);
                                if(read!=-1)total=total + new String(buffer,0,read);
                                else break;
                        }
                        is.close();


                        System.out.println("BUFFER READ: "+(System.currentTimeMillis()-time));
                        JSONObject wrapper = new JSONObject(total);
                        System.out.println("CONVERTED TO OBJECT: "+(System.currentTimeMillis()-time));
                        JSONArray objects = wrapper.getJSONArray("rows");
                        HashSet<String> commenters = new HashSet<String>();
                        for(int i=0;i<objects.length();i++)
                        {
                                JSONObject object = objects.getJSONObject(i);
                                JSONArray blogPost = object.getJSONArray("value");

                                if(blogPost.getString(0).equals(personID))
                                {
                                        for(int j=1;j<blogPost.length();j++)
                                        {
                                                commenters.add(blogPost.getString(j));
                                        }
                                }
                        }
                        System.out.println("DONE: "+(System.currentTimeMillis()-time));

                }
        }
        public static void main(String[] args) throws Exception
        {
                query2("test2");
        }

}
```

```
public class MongoDBQuery {

	public static void query1(String databaseName)throws Exception
	{
		Mongo m = new Mongo("localhost");
		DB db = m.getDB(databaseName);
		DBCollection eventsColl = db.getCollection("events");
		for(int i=0;i<20;i++)
		{
			long time = System.currentTimeMillis();
			int personID = 500+100*i;
			BasicDBObject query = new BasicDBObject();
			query.put("attending",personID);
			DBCursor  cur = eventsColl.find(query,new BasicDBObject("attending",1));

			while(cur.hasNext())
			{
				//System.out.println(cur.next());
				cur.next();
			}
			System.out.println(System.currentTimeMillis()-time);
		}
	}
	public static void query2(String databaseName)throws Exception
	{
		Mongo m = new Mongo("localhost");
		DB db = m.getDB(databaseName);
		DBCollection blogPostsColl = db.getCollection("blogPosts");
		for(int i=0;i<20;i++)
		{
			long time = System.currentTimeMillis();
			int personID = 500+100*i;
			BasicDBObject query = new BasicDBObject();
			query.put("author",personID);
			DBCursor  cur = blogPostsColl.find(query,new BasicDBObject("comments.person",1));

			BasicDBList persons = new BasicDBList();
			while(cur.hasNext())
			{
				DBObject dbo = cur.next();
				BasicDBList comments = (BasicDBList)dbo.get("comments");
				for(int j=0;j<comments.size();j++)
				{
					persons.add(((BasicDBObject)comments.get(j)).get("person"));
				}
			}
			System.out.println(System.currentTimeMillis()-time);
		}
	}
	/**
	 * @param args
	 */
	public static void main(String[] args)throws Exception
	{
		query2("test1");

	}

}


import java.io.*;

import org.apache.http.*;
import org.apache.http.client.*;
import org.apache.http.client.methods.*;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.DefaultHttpClient;


public class Neo4jQuery {

	public static HttpClient client = new DefaultHttpClient();
```

```
		public static void query1()throws Exception
		{
			for(int i=0;i<20;i++)
			{
				int personID = 500+100*i;
				long time = System.currentTimeMillis();
				String query = "{\"order\" : \"breadth_first\",\"return_filter\" :
{\"body\" : \"position.length()==2;\",\"language\" : \"javascript\"},\"uniqueness\" : \"node_global\",\"relationships\" :
[ {\"direction\" : \"in\",\"type\" : \"attending\"}, {\"direction\" : \"out\",\"type\" : \"attending\"} ],\"max_depth\" : 2}";

				HttpPost post = new
HttpPost("http://127.0.0.1:7474/db/data/node/"+personID+"/traverse/node");
				post.setHeader("Content-Type", "application/json");
				post.setEntity(new StringEntity(query));
				HttpResponse response = client.execute(post);

				response.getEntity().getContent().close();
				System.out.println((System.currentTimeMillis()-time));
				/*BufferedReader br = new BufferedReader(new
InputStreamReader(response.getEntity().getContent()));
				String line = "";
				while ((line = br.readLine()) != null)
				{
					System.out.println(line);
				}
				br.close();*/
			}
		}
		public static void query2() throws Exception
		{
			for(int i=0;i<20;i++)
			{
				int personID = 500+100*i;
				long time = System.currentTimeMillis();
				String query = "{\"order\" : \"breadth_first\",\"return_filter\" :
{\"body\" : \"position.length()==3;\",\"language\" : \"javascript\"},\"uniqueness\" : \"node_global\",\"relationships\" :
[ {\"direction\" : \"out\",\"type\" : \"blogPost\"}, {\"direction\" : \"out\",\"type\" : \"comment\"},
{\"direction\" : \"out\",\"type\" : \"commenter\"} ],\"max_depth\" : 3}";

				HttpPost post = new
HttpPost("http://127.0.0.1:7474/db/data/node/"+personID+"/traverse/node");
				post.setHeader("Content-Type", "application/json");
				post.setEntity(new StringEntity(query));
				HttpResponse response = client.execute(post);

				response.getEntity().getContent().close();
				System.out.println((System.currentTimeMillis()-time));
				/*BufferedReader br = new BufferedReader(new
InputStreamReader(response.getEntity().getContent()));
				String line = "";
				while ((line = br.readLine()) != null)
				{
					System.out.println(line);
				}
				br.close();*/
			}
		}
		public static void main(String[] args) throws Exception
		{
			query1();

		}

}
```