

Institutionen för datavetenskap
Department of Computer and Information Science

Final Thesis

Distributed Storage and Processing of Image Data

by

Tobias Dahlberg

LIU-IDA/LITH-EX-A--12/051--SE

2012-10-25



Linköpings universitet

Final Thesis

Distributed Storage and Processing of Image Data

by

Tobias Dahlberg

LIU-IDA/LITH-EX-A--12/051--SE

2012-10-25

Supervisor: Fang Wei-Kleiner (IDA)
Robert Andersson (Sectra)

Examiner: Patrick Lambrix

Abstract

Systems operating in a medical environment need to maintain high standards regarding availability and performance. Large amounts of images are stored and studied to determine what is wrong with a patient. This puts hard requirements on the storage of the images. In this thesis, ways of incorporating distributed storage into a medical system are explored. Products, inspired by the success of Google, Amazon and others, are experimented with and compared to the current storage solutions. Several “non-relational databases” (NoSQL) are investigated for storing medically relevant metadata of images, while a set of distributed file systems are considered for storing the actual images. Distributed processing of the stored data is investigated by using Hadoop MapReduce to generate a useful model of the images metadata.

Table of Contents

1 Introduction.....	1
1.1 Problem Description	1
1.2 Motivation	2
1.3 Thesis Overview	3
2 Requirements	4
2.1 Main Requirements	4
2.1.1 Distributed Storage	4
2.1.2 Proof of Concept	4
2.2 Extra requirements	4
2.3 Remarks	5
3 Evaluation of Technologies	6
3.1 NoSQL Databases.....	6
3.1.1 Google’s Bigtable	6
3.1.2 Amazon Dynamo	8
3.1.3 Apache Cassandra	9
3.1.4 DataStax Enterprise	10
3.1.5 HBase	10
3.1.6 MongoDB	11
3.1.7 Skipped alternatives.....	12
3.2 Distributed File Systems	13
3.2.1 HDFS.....	13
3.2.2 GlusterFS	13
3.2.3 Skipped alternatives.....	14
3.3 Benchmark.....	14
3.4 MapReduce.....	16
3.4.1 Hadoop MapReduce	17
3.4.2 MongoDB MapReduce	17
3.5 Discussion & Conclusions	18
4 Cluster configuration.....	19

4.1 VirtualBox	19
4.2 Configuring Apache Cassandra	19
4.3 Configuring HDFS	20
4.4 Configuring HBase	21
4.5 Configuring MongoDB	21
4.6 Configuring GlusterFS	22
4.7 Configuring Hadoop MapReduce	23
4.7.1 Integration	23
5 Generating the Data Model.....	25
5.1 Purpose	25
5.2 The DICOM standard	25
5.2.1 Introduction	25
5.2.2 Modules	25
5.2.3 Attributes	25
5.2.4 Libraries.....	26
5.3 Migrating to MongoDB	26
5.4 Considerations	27
5.5 Design	27
5.6 Details	28
5.6.1 UnitMapper.....	29
5.6.2 ModelReducer.....	29
5.7 Validation.....	30
6 Summary & Discussion	31
6.1 Summary.....	31
6.2 Why NoSQL?	32
6.3 Short-term storage	32
6.4 Long-term storage	33
6.5 MapReduce.....	33
6.6 Future work	34
7 References.....	35
8 Appendix.....	37
8.1 ModelGenerator.java	37

8.2 Abbreviations..... 41

List of figures

Figure 1: Overview of the Sectra PACS structure..... 1

Figure 2: An illustration of the Bigtable column family data model 7

Figure 3: Illustration of consistent hashing. The values are rearranged minimally when node E is added or removed. 8

Figure 4: Topology of a HBase cluster 11

Figure 5: Min and median for reads and writes..... 15

Figure 6: Comparison of sorted and linked read latencies. 15

Figure 7: Comparison of sorted and linked write latencies. 16

Figure 8: Data flow in MapReduce 17

Figure 9: A simplified UML class diagram of the model generator..... 28

Figure 10: Screenshot of the DICOM model visualizer used for validating the generated model. 30

1 Introduction

This chapter is the introduction to a master thesis project (30 credit points) final report examined at the Department of Computer and Information Science (IDA) at Linköping University. The thesis is the final part of a 5 year degree leading to a Master of Computer Science and Engineering. The thesis work has been performed at Sectra Medical Systems AB¹. Sectra Medical Systems develops systems for increasing efficiency in healthcare. One of their products is a Picture Archiving and Communication System (PACS) used to store and access radiology images at hospitals around the world.

1.1 Problem Description

Figure 1 shows the main components of the Sectra PACS and how they communicate.

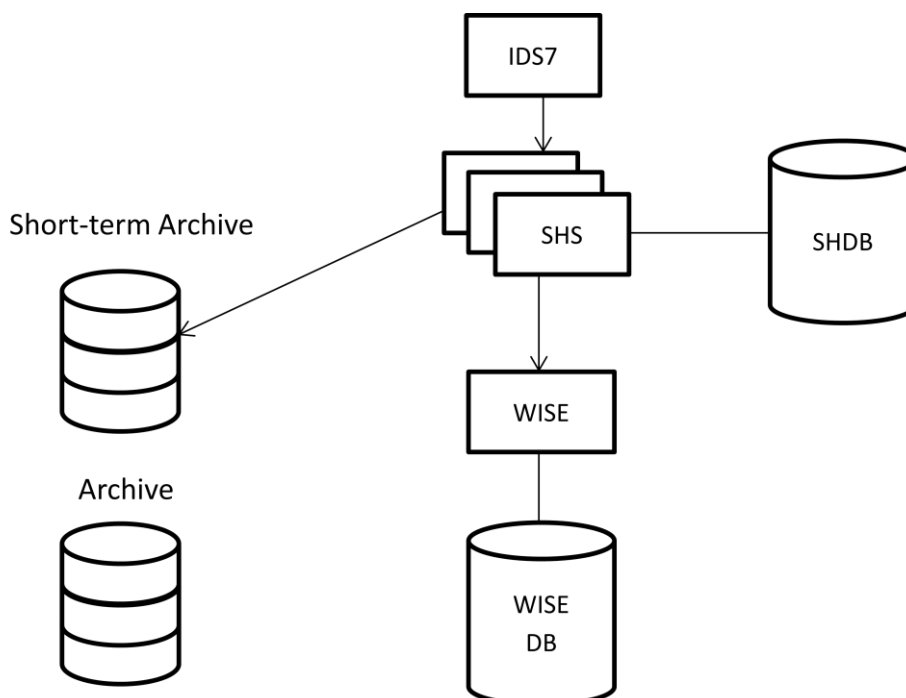


Figure 1: Overview of the Sectra PACS structure

The hub of the system is the SHS (Sectra Healthcare Server), which receives requests from IDS7 (Image Display Service) clients and fetches data from the SHDB database, WISE and the archive. The WISE Database mostly contains information about patients and the examinations they have gone through. SHDB contains a more abstract model of individual examinations, grouping images based on orientation and position. To cope with large amounts of requests from many clients simultaneously, several SHS servers can exist on a site.

The individual DICOM images are generally fetched from the short-term archive, where images are stored for about two years before moving them to the (long-term) archive. Today

¹ For more information about Sectra Medical Systems, visit www.sectra.com/medical

the archive is generally a SAN (Storage Area Network) built using specialized hardware. Without going into details about the architecture of a SAN, they provide high performance and availability due to the custom made hardware. There are some drawbacks however. The hardware is rather expensive and the structure is quite complex, even though these issues are not as significant as they have been. Adding storage capacity to a SAN could require adding a lot of extra hardware to prevent loss of performance.

The following use case(s) describes how the data in a PACS is used. The purpose is to give a better understanding of what needs to be prioritized when choosing an appropriate distributed solution.

A patient gets a CT (Computed Tomography) scan, MRI (Magnetic Resonance Imaging) scan or a Mammography screening. The images from the test are then saved in the archive, along with data such as where on the patient each image was taken etc.

Later the physician examines the result on a visualization table or on a regular computer with IDS installed. The images and their metadata are not modified, but accessed multiple times. The physician has many tests to analyze and wants to wait as little as possible to get the images.

The alternative solution explored in this project is basically to investigate the viability of changing the structure of the short-term and/or long-term archive into a more distributed one.

1.2 Motivation

The architecture and topology of distributed storage clusters differs a lot, but one common denominator of all the explored products is that relatively cheap commodity hardware can be used for the nodes in the cluster, while the software handles most of the work. This, among other things, makes it an interesting alternative to a SAN, since it is potentially easier to expand.

Many, if not all, distributed storage solutions improve the availability of data by automatic replication. Alternatives include HDFS, GlusterFS, HBase, Cassandra, MongoDB, CouchDB and many more.

Distributed storage enables parallelized processing of the stored data, e.g. by using a Map/Reduce framework like Apache Hadoop or similar. For example it can be useful for creating a data model when migrating between different versions of the Sectra PACS. Today, the process takes a rather long time and is run as a background process for several weeks in some cases. A lot of unnecessary reads and writes are made to the SHDB (Sectra Healthcare Database) where the data model is stored. To avoid this unnecessary data traffic, an intermediate storage is required, to ensure that all images belonging to the same stack are read before processing further. It will then be possible to input data more efficiently into the SHS. Some improvements to the data model generation should also be achievable. Another,

more computationally demanding, task that could be parallelized is compression of images. As a part of the investigation, a proof-of-concept that is implemented on top of the distributed storage is done in this thesis project. Mainly the migration of the DICOM headers will be considered for this.

1.3 Thesis Overview

This section describes how the report is constructed, briefly describing the content of each chapter.

Chapter 2 lists the formulated requirements of the thesis.

Chapter 3 describes the investigated products and compares them through a benchmark and discussion.

Chapter 4 gives a detailed description of how the products were configured in order to get started.

Chapter 5 introduces the DICOM standard, and goes through the design of the proof-of-concept implementation.

Chapter 6 summarizes the thesis and discusses the possible uses of a distributed storage solution.

For a list of explained abbreviations, see appendix 8.2.

2 Requirements

One of the first steps was to define the scope and requirements of the thesis. This served several purposes. Due to the limited time frame available for the thesis, the scope needed to be narrowed down and concretized. Another purpose was to be able to determine when the thesis was to be considered done.

This chapter describes the original requirements and discusses a few adjustments that were made during the thesis (by first discussing it with Sectra).

2.1 Main Requirements

2.1.1 Distributed Storage

- It shall be possible to access a random image file of size 100kB in less than 100ms from the distributed storage.
- The distributed storage shall be able to handle images of sizes between 50kB and 50MB efficiently.
- The distributed storage shall provide redundancy of data, to avoid a single point of failure.
- Some parallelization framework shall be supported by the distributed storage solution.
- A distributed storage solution that best satisfies the requirements of this case shall be chosen, after comparing several different alternatives available.

2.1.2 Proof of Concept

- A prototype that uses some kind of parallelization on top of the distributed storage shall be implemented. Either by using a framework such as Apache Hadoop, or using a custom method.
- The prototype shall implement data model generation.
- Duplicate images shall be sorted out during the migration.
- The prototype shall be able to handle single-frame DICOM files as input.
- The prototype shall be able to handle volume stacks.
- The prototype shall be able to handle time stacks.
- The prototype shall be able to handle volume stacks with a time component.
- The output shall not contain redundant information.
- The output shall represent the SHS image data model.
- The prototype shall be designed for scaling up to use in large data sets.

2.2 Extra requirements

These requirements were to be implemented if there was time.

- A tool to visualize the data model should be created.
- A checksum should be used to improve the check whether images are duplicates in the proof-of-concept application.

- Evaluate communication with the WISE database with MapReduce.
- Implement a proof-of-concept for parallelized compression of images on several computers/nodes.

2.3 Remarks

Due to lack of deeper knowledge of the area beforehand, some requirements were intentionally rather fuzzy, especially for the distributed storage. This allowed for exploring many different alternatives to accomplish the task. As discovered later, the performance requirement of 100ms for retrieving 100kB was very unrestrictive, since all the tested products manage this with a significant margin.

Other requirements became more or less irrelevant as new revelations were made and priorities changed. For instance, the proof-of-concept was considered to have higher priority in the beginning than later in the process. This can be confirmed by noting that the extra requirements were all connected to improvements of the proof-of-concept application.

3 Evaluation of Technologies

The first step to finding a storage that fits the requirements well for this use case (see Chapter 2), was to keep an open mind and listing available products. This list quickly grew rather large. As interesting as it would have been, there was neither the time nor the need to try out all the products. Instead many of them had to be skipped just by reading about them.

The use of NoSQL databases is getting more and more popular and the number of available products is constantly growing. The field is still rather new and existing products are constantly adding new features. This makes it hard to compare different products, since previous comparisons quickly can get obsolete depending on what new features are being added. Therefore mostly the key features, which are not likely to change, are described and put in relation to the other products. As a starting point the sources [1], [7] and [9] were read through. The main conclusion they make is that there is no the universal choice which fits all cases. Instead you need to figure out what is most important in each specific case and choose accordingly. This chapter aims to shed some light on the relevant technologies and how the features they provide fit, or do not fit, into a PACS.

3.1 NoSQL Databases

This section starts by describing two of the biggest inspirations for several of the available NoSQL products, namely Google's Bigtable and Amazon Dynamo. Then continues by discussing some of the products and why they were skipped. Advantages and disadvantages with Apache Cassandra, HBase and MongoDB are discussed.

3.1.1 Google's Bigtable

Bigtable, described in [3], is Google's distributed database used for storing the data used in many of Google's applications, such as web indexing, Google Earth, and Google Finance. Even though it is not available for public use, its design ideas have inspired open-source clones. This section gives a brief description.

3.1.1.1 *Sorted Ordered Column-Oriented Store*

The model used by Google's Bigtable and its clones stores data in a column-oriented way, in contrast to the row-oriented format in RDBMS (Relational Database Management System). Columns without a value in a specific row are simply not stored, which makes it more efficient for sparse tables. Columns do not need to be defined when creating the table, but can be added on the fly. No limitation is set for the type of data a column can contain, as long as it can be represented as an array of bytes.

Multiple columns are grouped into a column family. These columns should be logically related in some way (although this is not a requirement). For instance a column family "name" could contain the columns "first_name", "middle_name" and "last_name". When creating a table, the column families it contains are set, but not the actual columns. This

adds lots of flexibility compared to a relational database where all columns are defined up front.

Updating a value in the database will not typically delete the old value. Instead the new value will be inserted, with a newer timestamp than the old value. A table in a column-oriented store can be viewed as a multi-dimensional map indexed by a row-key, column-key and a timestamp. Each row is uniquely identified by its row-key. Each row contains a set of columns, where the column name functions as the column-key. Furthermore every value can have multiple versions, identified by a timestamp value.

Physically data is stored by column-families. The data is sorted by row-key which makes data seeks by row-key very efficient. The sorted order does not only apply for a single node, but for the entire cluster.

The diagram illustrates a Bigtable structure. It shows a table named 'Table: MyBigTable' containing a 'Column Family: Messages'. This family is divided into two sections. The first section has a row with key '1234abcd' and columns 'Text', 'From', 'To', and 'Date', with values 'Hello', 4113, 116, and 2012-05-15. The second section has a row with key '5678ef90' and columns 'Text', 'To', 'Date', and 'Color', with values 'World', 116, 2012-05-15, and #000000. Ellipses indicate further data.

Table: MyBigTable				
Column Family: Messages				
1234abcd	Text	From	To	Date
	"Hello"	4113	116	2012-05-15
5678ef90	Text	To	Date	Color
	"World"	116	2012-05-15	#000000
...				

Figure 2: An illustration of the Bigtable column family data model

3.1.1.2 Storage Architecture

The Bigtable data model alone does not make it a distributed data store. To make data persistent the Google File System (GFS) is used, which takes care of replication. Additionally, a distributed lock service called Chubby is used for several tasks. It is used to ensure there is only one active master node at any time, discovering tablet servers and storing Bigtable schema information (column family information) to mention the most important tasks. It is important that there is only one assigned master node at any given time, to avoid issues with consistency.

The term 'tablet' is referring to a part of a table with limited size. Each table is divided into a set of tablets in order to make it simpler to distribute the table among multiple nodes. To keep track of the tablets, a cluster of tablet servers is used. A single master is taking care of assigning tablets to tablet servers, balancing tablet server load and handling schema changes.

3.1.2 Amazon Dynamo

Dynamo is a key/value store used by Amazon to power their internal services. Its principles are described in [18]. It is designed for high availability and fault-tolerance and has inspired open-source products with the concept of eventual consistency. Dynamo is built on the ideas of consistent hashing, object versioning, gossip-based membership protocol and hinted handoff. These are described briefly in this section.

3.1.2.1 Consistent Hashing

Consistent hashing is an important principle for distributed hash tables. The purpose is to partition data among the nodes in a way that changes as little as possible when a node in the cluster is removed or added. A modulo based hash would not be good for this. Instead a large hash space is created, e.g. all SHA1 (Secure Hash Algorithm) keys up to a very large value, and mapped onto a circle. The nodes are mapped evenly to the same circle. The values are then mapped to the closest node in the circle (see Figure 3). This implies minimal rearrangement when nodes are added or removed. [1]

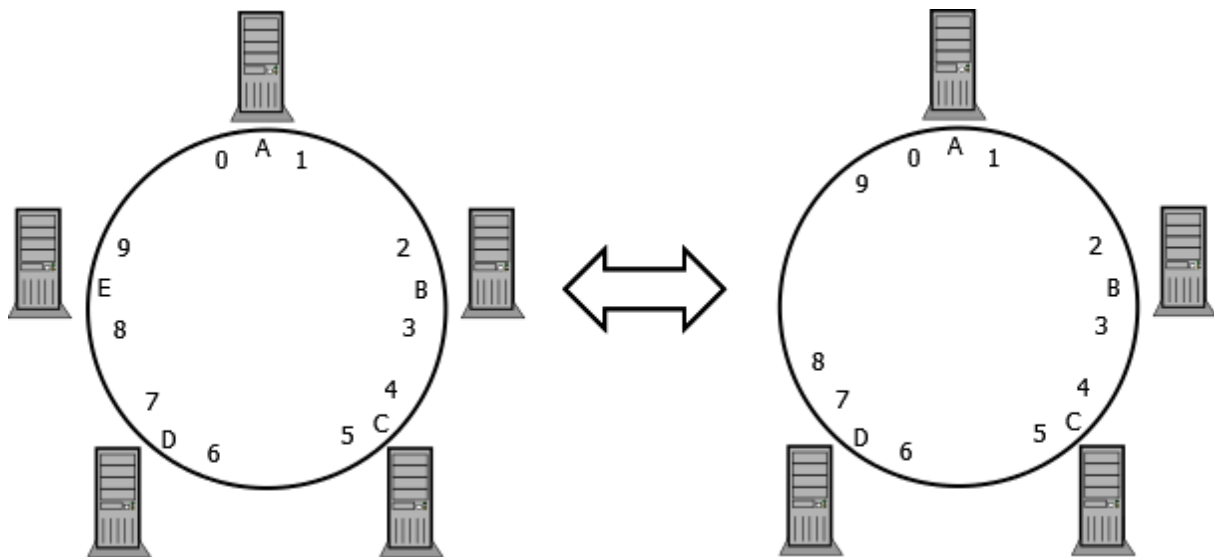


Figure 3: Illustration of consistent hashing. The values are rearranged minimally when node E is added or removed.

3.1.2.2 Object Versioning

While consistent hashing provides efficient partitioning, the purpose of object versioning is to keep data consistent. To avoid the huge overhead of ACID transactions in distributed databases, vector clocks are used together with object versioning to keep data consistent. A vector clock is basically a set of logical clock values corresponding to logical clocks on each node. A node updates the vector clock when sending and receiving values. A vector clock with all respective values equal to or greater than another vector clock's respective values is considered to "descend" from it. If neither of the two vector clocks descends from the other, there is a conflict that needs to be resolved.

3.1.2.3 *Gossip-Based Membership*

Instead of having a master node for keeping track of nodes in a cluster, Dynamo uses a Gossip-based protocol to communicate this information in a peer-to-peer manner. The communication is done periodically to detect unavailable nodes. When a node finds out that some node has been added or removed, it attempts to give a message to the other nodes in the cluster.

3.1.2.4 *Hinted Handoff*

The eventual consistency is supported by allowing writes even when some nodes in the cluster are unavailable. Writes are performed on the healthy nodes and records a hint to let the unavailable nodes know when they are available again. This improves the durability of the system. The minimum number of live nodes available when performing a write operation, W , can be set, allowing the user to make a tradeoff between availability and consistency. For maximum availability W is set to 1, meaning that only a single node needs to be live to perform the operation.

3.1.3 Apache Cassandra

Cassandra² combines some of the best from the principles of Amazon Dynamo and Google's Bigtable. There are no special master or metadata nodes (same as Dynamo) while still providing replication of data across the cluster and high performance together with the rich data model of Bigtable [5]. Consistent hashing, Gossip-based membership and hinted handoff are all implemented. Object versioning is implemented, in the sense that conflicts can be detected. The difference is that timestamps are used instead of vector clocks. This requires the nodes in the cluster to be synchronized, since comparing timestamps from different nodes would otherwise be pointless. It was first developed and used by Facebook, and later released to the community as an open-source project under an Apache license.

3.1.3.1 *Schema-optional*

The Bigtable data model uses a schema-less approach when storing the data, meaning that information about the data type of a specific column is not stored anywhere in the database. This makes it possible to add columns on-the-fly. This is the case in Cassandra as well. However a stricter schema is also supported. This enables validity checks and other benefits similar to a RDBMS. This can be referred to as schema-optional. It is even possible to set metadata for some columns letting Cassandra take care of type checking and so on, while still being able to add columns dynamically later on.

3.1.3.2 *Querying*

Cassandra requires an index to support queries on a certain field. By default, a row-key index (primary index) is provided. Creation of secondary indexes is supported to enable queries like "get Users where lastname = 'Smith' ". This kind of query is not supported for columns

² <http://cassandra.apache.org>, August 2012

without an index. As a consequence, it is only possible to query dynamic column families by row-key.

They have even introduced their own query language CQL (Cassandra Query Language) which has similar syntax as SQL but with operations such as JOIN stripped away [11]. There is still the limitation that queried columns must be indexed.

3.1.4 DataStax Enterprise

There is also a commercial product based on Cassandra. It is called DataStax Enterprise (DSE), which provides tight integration with the Hadoop MapReduce framework as well as monitoring tools and support [12]. The open-source version has built-in support for Hadoop applications to access Cassandra tables as well, but it requires more configuration.

It is not designed for storing and handling large objects such as images directly. This is a common fact for all the NoSQL alternatives considered here, but Cassandra seems to have more clear limitations when it comes to this issue. For example, there is no support for streaming large objects. They must be loaded entirely into memory before being able to use them, and no automatic way of splitting the objects into chunks is provided in the open-source version. The DSE provides a HDFS compatible file system built on Cassandra called CFS, which splits files into smaller blocks [10].

3.1.5 HBase

HBase³ is more or less an open-source clone of the Bigtable design, using a column-based data model. It is not as simple to get started as with Cassandra, at least not to get a cluster up and running. This is because HBase has a significantly more complex architecture, with several different types of nodes. As opposed to Cassandra, HBase is dependent on a standalone distributed file system to store all data. This means that the DFS needs to be configured and started before HBase can be run on a cluster. It is designed to work well on top of HDFS, which it does. There was no problem using HBase once HDFS was correctly configured and started. It is not a requirement to use HDFS as the underlying file system. It worked just as good to store the data in GlusterFS as well, with minor changes to the configuration. This could be an advantage compared to Cassandra, which directly uses the local file system to make data persistent. This provides flexibility.

Unlike Cassandra, HBase requires one master node to handle metadata and keep track of the RegionServers. The RegionServers (tablet servers in Bigtable) can still function if the master goes down, but if one of them fails without the master running, there will not be any automatic failover ([13], Section 8.5. - Master). There is an open-source counterpart to Chubby called ZooKeeper. A ZooKeeper cluster keeps track of schema information and makes sure there is only one master, among other things.

³ <http://hbase.apache.org>, August 2012

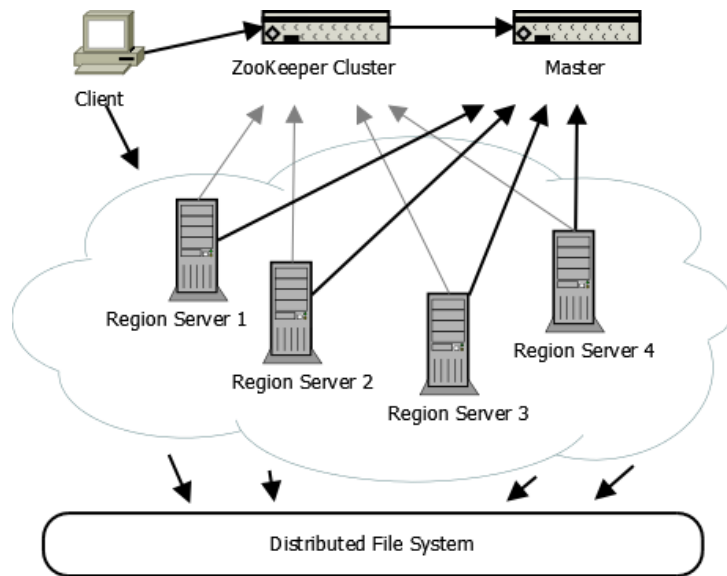


Figure 4: Topology of a HBase cluster

3.1.6 MongoDB

Being a document database, MongoDB⁴ has a bit simpler data model than Cassandra and HBase. A database contains a number of collections and a collection contains documents. Data is stored as BSON documents, without any requirements on what attributes a specific document contains. However it is a good idea to keep similar documents in the same collection, since it will be possible to perform more efficient querying. MongoDB provides this by being able to create indexes for certain attributes, making it fast and easy to do queries on them [14]. Queries on attributes without an index are also supported.

By configuring a set of nodes to form a replica set, data operations will automatically be replicated to all nodes in the replica set. If the load on the primary node in the replica set becomes a bottleneck, it is possible to perform queries directly on the secondary nodes.

MongoDB is simple to work with and supports multiple platforms and provides drivers for many programming languages including Java, C# and C++.

3.1.6.1 Horizontal Scaling

To be able to scale out as data grows bigger, sharding is provided to split data evenly across multiple servers. This is done automatically once it is setup [14]. Each shard is usually a replica set, providing fault tolerance. Additionally, there is a config server (also replicated) which contains information about the shards. Queries on a sharded collection are distributed and processed in parallel on the different shards, before combining and returning the result.

3.1.6.2 Document-based vs. Column-based

In both document stores and column-based stores the schemas are very loosely defined. A collection could be said to correspond to a column family and a document is kind of like a row in a column family. An attribute in a document would in that sense correspond to a

⁴ <http://www.mongodb.org>, August 2012

column. The richer column-based model makes it more efficient to access specific values, while MongoDB must load the entire document into memory before using it.

The document model has the advantage that more complex structures can be stored easily. A value of an attribute can in itself be a document, and BSON has native support for storing arrays of values.

3.1.7 Skipped alternatives

3.1.7.1 *Key/Value stores*

The simplest group of NoSQL databases generally intended to be very fast and used for relatively small data sets. The simple data model is also rather limiting in this case, since you may want to get only specific parts of the metadata without fetching all the data first. Many of the more sophisticated database systems use a key/value store as an underlying engine, but as standalone products, they lack many desired features for this use case. There are none or very limited querying possibilities to mention one shortcoming.

3.1.7.2 *Graph databases*

Graph databases are not interesting for this use case, since the images are not very connected to each other. It might have been a good idea to use for storing the data model for stacks, but since that was not in the scope of this thesis project, the graph databases were simply left out.

3.1.7.3 *CouchDB*

CouchDB⁵ is a document database with many similarities to MongoDB. There were several reasons why it was skipped though. The main reason was that MongoDB had better features concerning scalability and performance [8]. While MongoDB supports sharding natively, CouchDB does not, although it is possible through third-party software. Instead the scalability lies in performance, not in storage capacity. CouchDB is designed to provide efficient updates of the data, which is of little benefit in this case, where the data is not modified.

3.1.7.4 *Hypertable*

Like HBase, Hypertable⁶ is an open-source clone of Google's Bigtable. It is written in C++ in order to be more efficient, but provides interoperability with several other languages through a Thrift interface. Apart from seemingly large improvements in performance for Hypertable compared to HBase, they have very similar features. The reason why HBase was chosen was the lack of native support for Hadoop MapReduce in Hypertable. It is possible to interact through Hadoop Streaming though. If HBase proved to be the best choice for storing medical data in this case, it might have been good to look more into Hypertable.

⁵ <http://couchdb.apache.org>, August 2012

⁶ <http://www.hypertable.com>, August 2012

3.2 Distributed File Systems

3.2.1 HDFS

HDFS (Hadoop Distributed File System)⁷ implements the ideas presented for GFS, described in [2]. It is designed for handling large files well and to work well with MapReduce jobs. Also HBase is built to work well with HDFS as the underlying file system. As a downside, it is not designed for low latency and it is not a general file system.

3.2.1.1 *Architecture*

HDFS has a master-worker architecture, similar to the Bigtable/HBase structure. A set of datanodes are used for storing the data and a single namenode is used to keep track of where data is stored, serving client requests. Large files are split into blocks (usually 64MB) and distributed among the datanodes. This opens up for streaming of large files. Another upside of the metadata store is that the location information can be used to make the calculations in MapReduce jobs close to the data.

Every file, directory and block in HDFS is represented as an object in the namenode's memory, each of which occupies about 150 bytes. If 10 million files are stored, each using one block, then about 3 gigabytes of memory would be required. This makes the storage capacity a matter of how much memory is available in the namenode.

Clients do not read or write data through the namenode, since this would impose a high load. Instead clients only contact the namenode to get information about which datanode and block to use. After that the client communicates directly with the datanode where the data block resides.

The namenode can (and should) be backed up by replicating its state and making checkpoints, but if the namenode fails there is no automatic failover, which means that you must recover it manually [15]. This compromises availability demands in a critical environment like medical systems, which is definitely undesirable.

3.2.1.2 *Client access*

HDFS is not made to be POSIX (Portable Operating System Interface) compatible, with the motivation that it would cause unnecessary loss of performance. The two directly supported ways of interacting with HDFS is through a command line tool provided in Hadoop distributions, or through the Java API. There are several third party software that can be used to access HDFS through for example a web browser, but most of them are flawed and lack certain functionality, such as being able to write.

3.2.2 GlusterFS

GlusterFS⁸ is an open-source distributed file system like HDFS, but based on different design decisions. GlusterFS is a more general file system since there are optimizations for handling

⁷ <http://hadoop.apache.org/hdfs>, August 2012

⁸ <http://www.gluster.org>, August 2012

both large and small files efficiently. Large files can be striped (split) across several nodes, making it possible to read from many nodes simultaneously to get higher throughput. Very small files are handled by an underlying database system to improve performance [16]. GlusterFS is also POSIX-compatible as opposed to HDFS.

3.2.2.1 *Elastic Hash Algorithm*

The most fundamental difference with GlusterFS compared to HDFS and many other DFSs is that GlusterFS does not depend on metadata to determine the location of a specific file. Instead this is found out algorithmically. This removes many issues that metadata based systems have. Either the metadata server is a single-point-of-failure and/or a choking point, or the metadata is distributed, causing an increasing overhead as the storage scales. The Elastic Hash Algorithm in GlusterFS enables all nodes in the cluster to access any data independently and in $O(1)$ time on average.

3.2.2.2 *Client access*

Unlike HDFS you can mount a GlusterFS volume to a suitable directory on the operating system (Linux). When mounted, files can be browsed just like in any other directory. The difference being that there is a quite large overhead. It is a powerful feature, but the performance cost may or may not be too high. There are other ways to interact with a GlusterFS volume, but mounting it is the recommended way. To access a GlusterFS volume from a Windows client, a GlusterFS node must make the mounted directory accessible through a CIFS share. The Windows client can then mount it as a network drive and browse the contents of the volume. A native windows client is being considered, but not yet implemented.

3.2.3 Skipped alternatives

There was a long list of DFSs, but many were not open-source. Open-source was not a requirement, but convenient for testing. One that looked promising was Lustre, previously developed by Sun, but was abandoned when Sun was taken over by Oracle. Ceph and KosmosFS were two other alternatives with nice features. They both seemed to be rather immature products as of yet, which was the main reason why they were skipped. FhGFS (Fraunhofer-Gesellschaft File System) was another interesting DFS, but since there was no mentioning of Hadoop compatibility this was skipped as well.

3.3 Benchmark

All of the above systems, both NoSQL databases and DFS systems, were installed and configured on a three node virtual cluster. More than two nodes were needed to utilize the desired features, and running four or more virtual machines on a single computer would have been too inefficient. The three virtual machines were all running Ubuntu 11.10 (64-bit). A Java application was made to interact with all of the above systems, logging results in performance when writing and reading a varying size and number of entries. Figure 5 shows the results from a test performed with 500 files of size 100kB.

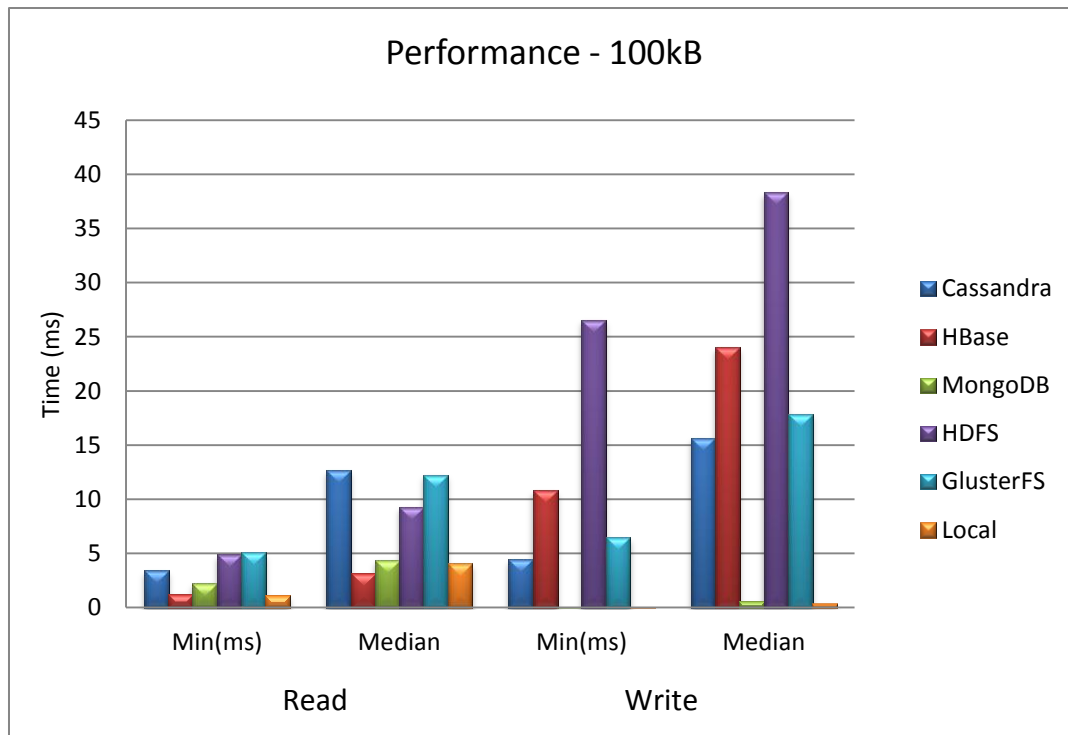


Figure 5: Min and median for reads and writes.

The reason why the median value was used to compare the overall performance was that a few values were very high. To better show the overall distribution of latency times, the values were sorted and displayed as graphs (Figure 6, Figure 7). One reason for the peak values might be the high load on the host caused by running three virtual machines simultaneously. Garbage collection or moving data from memory to disk could be causes as well.

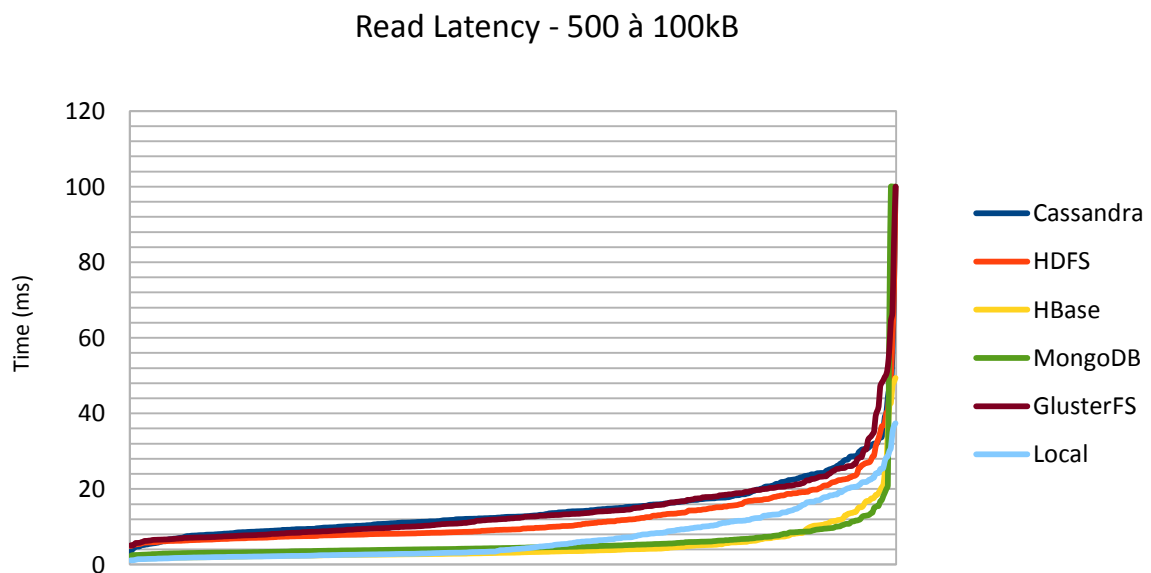


Figure 6: Comparison of sorted and linked read latencies.

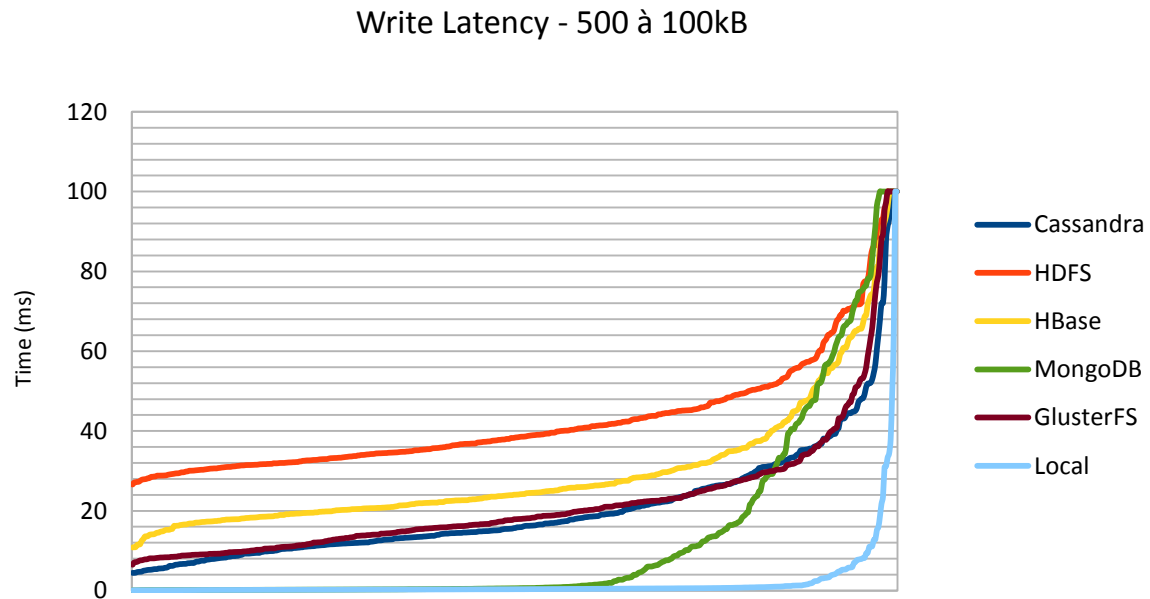


Figure 7: Comparison of sorted and linked write latencies.

For this use case the most important measure is when reading data. The original requirement (see Section 2.1) was to have a latency of less than 100ms for a 100kB file, although after consulting an expert on Sectra, a more realistic latency requirement would be around 10ms. From the figures 5 and 6 it can be concluded that the 100ms requirement is fulfilled with great margin by all the products, and the 10ms requirement in roughly 50% of the tests.

3.4 MapReduce

After installing, configuring and benchmarking the different systems, there was some experimenting with performing analytics through MapReduce frameworks.

The concepts behind Google's MapReduce implementation are explained in [4]. In short it is a framework for simplifying parallelization, fault tolerance, data distribution, load balancing, etc. when processing over a cluster. The mapping part takes each input value (such as a file or database entry) and translates it to an intermediate (key, value) form. The MapReduce framework then groups values together based on their respective keys. As default, (key, value) pairs with the same key will be grouped together. The reduce part then produces the result by processing each group independently and in parallel. The user only needs to provide the map and reduce function.

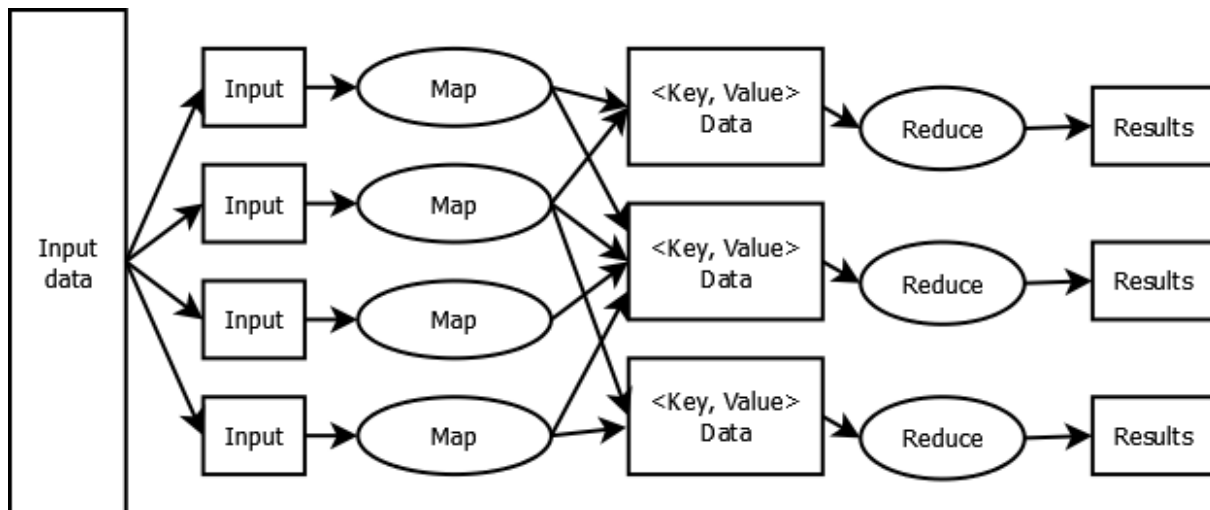


Figure 8: Data flow in MapReduce

3.4.1 Hadoop MapReduce

It was possible to integrate all the systems with Hadoop. Both Cassandra and HBase provide support for being accessed in Hadoop jobs. The only thing needed to get it working is to add the respective Java libraries to the Hadoop class path.

The Hadoop MapReduce⁹ processes use a (configurable) folder in HDFS as data storage by default. However, Hadoop enables the use of other file systems. There is a plugin to hook up GlusterFS as the underlying file system instead of HDFS. There were some issues to get it working though. After modifying the source code of the plugin, the MapReduce processes were able to start and run jobs exactly the same way as with HDFS as the file system. There was an issue when restarting the processes though, which only seems to be solved by removing the previous content of the folder used for storage. As a conclusion, it is possible to use GlusterFS as a replacement for HDFS, but the plugin needs a bit of work to be stable.

The number of other general MapReduce implementations was very small. Due to the popularity and momentum of Hadoop, third parties instead contribute to or integrate with Hadoop.

3.4.2 MongoDB MapReduce

To not only assume that Hadoop MapReduce was the framework to use, solely based on its popularity, another framework needed to be tested.

MongoDB has a MapReduce feature built in. The tests showed it had several shortcomings though. First of all, map and reduce functions are just strings describing JavaScript functions, which meant no help is to get from the IDE or the compiler, but instead you get an error during runtime if the function is faulty. Another limitation is that the output from the map function needs to have the same structure as the final output (from the reducer). This is not the case with Hadoop. And even if you get it to work the way you want, it is far too slow to

⁹ <http://hadoop.apache.org/mapreduce>, August 2012

be useful. Performing a fairly complex job on one million documents took almost ten minutes while an equivalent job with Hadoop never took more than about two minutes. Luckily, there is a plugin enabling Hadoop to access data from MongoDB. See [6] for a more in depth comparison between the MapReduce features of Hadoop, MongoDB and CouchDB.

3.5 Discussion & Conclusions

There is no clear winner that fits perfectly for storing large images, while at the same time separating metadata making it possible to analyze the metadata without reading the whole image at the same time. To accomplish this it would be necessary to use two of the systems in combination. Let a distributed file system (HDFS, GlusterFS or similar) handle the storage of the images and let one of the NoSQL systems (Cassandra, HBase or MongoDB) organize and handle the metadata. The question then becomes which two systems to choose.

At this point it is worth mentioning that DataStax Enterprise (see Section 3.1.4) might be able to provide a complete solution that can store both the files and the metadata efficiently. There was not enough time to try it out though, so there is only speculation at this point. If the performance of the file system (CFS) is good even for large images, DSE could prove to be a good choice. The tight integration with Hadoop MapReduce also fits well into the scope of this thesis.

Of the systems that have been tested, GlusterFS provides a more complete file system than HDFS. If the Hadoop integration would have worked better it would have been the obvious choice in favor of HDFS. The single-point-of-failure of HDFS is a vital disadvantage which is not to be taken lightly. The list of other mature distributed file systems is rather short.

As for the metadata storage, the size of the data is not estimated as of yet, but it might be unnecessary to use a distributed database because of too little data. However, it could be a good idea to use MongoDB to get automatic replication, and if the data size would grow it is possible to shard the data. This would not be as simple with an ordinary RDBMS.

Cassandra or HBase would also be viable choices for handling the metadata. HBase might be a slightly better fit, since it uses the existing distributed file system to store its data, and the faster reads, which is the most important in this use case. But considering the small size of the metadata they are somewhat complex for the task.

As a conclusion, using GlusterFS in a combination with MongoDB seems to be a good choice for handling large variations in data size. Hadoop integration works with a bit of tweaking. The simpler data model in MongoDB is totally sufficient for storing the metadata, without adding extra complexity that comes with a richer data model. Support for many programming languages is also a plus.

4 Cluster configuration

This chapter describes in a fairly detailed way how the systems were installed and configured to get up and running on three virtual machines. Note that this is just a description of how to get started. Only the basic configurations were made, leaving optimizing properties such as cache sizes with default values.

4.1 VirtualBox

All the tested systems are intended to work well on Linux environments, which is the dominating platform used in large clusters. For this reason the currently latest version of Ubuntu was used (Ubuntu 11.10 64-bit). The 64-bit version is not a requirement for any of the tested systems, but MongoDB for example has severe limitations when run on a 32-bit platform.

There are of course many other Linux distributions that may or may not have been a better choice, but the choice of distribution should not have a significant impact on the results from the tests, at least if compared to other general purpose Linux distributions. The purpose of creating the clusters was not to optimize the performance, but to get it working and getting a grasp on how the different products compare to each other. Besides, the performance overhead of running three virtual machines makes the performance of the operating system less significant.

2GB of memory was allocated for each virtual machine, and a virtual disk with a total size of about 15GB each. To make it possible for the virtual machines to communicate with each other through a network, a setting had to be made in VirtualBox to make the virtual machines use a bridged network card.

The first step to do in all the Ubuntu installations was to add all hostnames mapped to their respective IP-address. This was done by editing the file `/etc/hosts`. This was necessary to be able to use hostnames instead of IP-addresses when configuring the clusters, making it possible to change the IP-address afterwards.

4.2 Configuring Apache Cassandra

After downloading a distribution (1.0.8 was used in these tests) and extracting it to a suitable directory (e.g. `/usr/local/`), only a few things remained to get started. In the `conf/cassandra.yaml` file it is possible to edit which directories to use for storing data, commitlog and saved_caches. Before being able to start Cassandra all these directories had to be created and have sufficient permissions for the Cassandra daemon to use them. When this was done, Cassandra was able to start in stand-alone mode.

The only thing needed to start it in distributed mode was to edit a few more properties in `conf/cassandra.yaml`. The `'seed_provider'` property was set to the hostname of an existing node in the cluster. This means that the first node that is started should have localhost as `'seed_provider'`. The rest of the nodes could then use the already started node to get

connected to the cluster. Finally, the two properties 'listen_address' and 'rcp_address' must be changed to the actual IP-address of the node instead of 127.0.0.1. Otherwise other nodes cannot connect to the node. After a connection to one node in the cluster was established, information about all the other nodes was acquired automatically.

Starting Cassandra was only a matter of running *bin/cassandra*.

4.3 Configuring HDFS

Start by downloading the Hadoop distribution (1.0.0 was used in this case) and extracting it to a suitable directory (e.g. */usr/local/*) on all the machines in the cluster. There are three xml-files that are relevant to configuring Hadoop; *conf/core-site.xml*, *conf/hdfs-site.xml* and *conf/mapred-site.xml*. Set the property *fs.default.name* in *core-site.xml* to the URI (Uniform Resource Identifier) of the namenode. The following was added between the <configuration> tags during these tests:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://sectra-VirtualBox:9000</value>
</property>
```

In *hdfs-site.xml* the properties *dfs.replication* was set to 2, *dfs.name.dir* was set to */home/hadoop/dfs/name* and *dfs.data.dir* was set to */home/hadoop/dfs/data*. The two directories must exist and the owner should be set to the user that will run the process to be able to start HDFS. Also the logs directory need to be created (in the extracted Hadoop directory) with the correct permissions. This is where all the log files can be found.

Hadoop is written in Java and thus needs a JRE (Java Runtime Environment) to run. In Ubuntu this was done by running *sudo apt-get install openjdk-7-jre*. When installed, the path to the JRE was set as the value of *JAVA_HOME* in *conf/hadoop-env.sh*. There were other settings available in that file, but no further changes had to be made to get HDFS up and running.

Hadoop used passwordless SSH (Secure Shell) to communicate with remote Hadoop daemons (processes). When *ssh* and *sshd* were installed, the following commands were used to set this up:

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

The generated files were then added to the rest of the cluster by running the following commands on the respective nodes:

```
$ scp <user>@<host>:/home/<user>/.ssh/id_rsa* ~/.ssh/
$ scp <user>@<host>:/home/<user>/.ssh/authorized_keys ~/.ssh/
```

Finally *conf/masters* and *conf/slaves* needed to be edited. It is sufficient to only do this on one node, since the whole cluster is started from that node. The *masters* file shall begin with

the hostname of the namenode, followed by secondary master nodes (responsible for making backups and checkpoints of the namenode. The slaves file shall contain all the nodes to be used for storage, also called datanodes.

When all was correctly configured, the whole HDFS cluster was started by running *bin/start-dfs.sh* and *bin/stop-dfs.sh* stopped the cluster.

4.4 Configuring HBase

With HDFS configured and running (see section 4.3) it is relatively simple to get a functioning HBase cluster up and running. The steps are similar to the setup of HDFS. After downloading and extracting the distribution (0.9.2 was used) to a suitable directory (e.g. */usr/local/*) there were a few files that needed editing.

Starting with *conf/hbase-site.xml*, there were a few properties to add. To set the directory for storage in HDFS, the property *hbase.rootdir* was set to *hdfs://sectra-VirtualBox:9000/hbase*. If another file system was to be used, this property would be set to a corresponding URI. To setup HBase to run in a cluster, *hbase.cluster.distributed* was set to *true*. To define the nodes to run Zookeeper, *hbase.zookeeper.quorum* was set to a comma separated list of the hosts that should run Zookeeper, in this case *sectra-VirtualBox,sectra2-VirtualBox*.

For the node that would be used to start the daemons in the cluster, the file *conf/regionservers* was edited to contain the hostnames (one on each line) of the region servers. The path to the JRE was set as the value of *JAVA_HOME* in *conf/hbase-env.sh* on all nodes. Lastly, the directory *logs* had to be created with sufficient permissions on all nodes to enable the daemons to log important events.

Starting the HBase cluster was as easy as running *bin/start-hbase.sh* on a node with *conf/regionservers* correctly edited.

4.5 Configuring MongoDB

The easiest way to install MongoDB, at least on Ubuntu, was to use *apt-get install*. First the public gpg key needed to be added to make apt trust the repository. Then it was just a matter of installing the *mongodb* package.

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
$ sudo apt-get install mongodb
```

In this case all three nodes were configured to form one replica set, no sharding was involved. There was more than one way to configure the replica set. In this case, the file */etc/mongodb.conf* was edited and used when starting the MongoDB daemon. The setting for *bind_ip* was commented out, *repSet* was set to the name of the replica set to create and *rest* was set to true. It was also made sure that the directory set as *dbpath* was created and the correct permissions. Then the daemon was started by running the following command:

```
$ mongod --config /etc/mongodb.conf
```

When this was done for all nodes the mongo client was started on the node intended to be the primary node of the replica set (the others being secondary nodes) by simply running the command *mongo*. To initiate the replica set in the client was simple.

```
> rs.initiate(  
... {  
...  _id: <setname>,  
...  members: [  
...    { _id: 0, host: <host0>},  
...    { _id: 1, host: <host1>},  
...  ]  
... }  
... })
```

After this MongoDB was ready to use, where every modification of the data on the primary node was automatically replicated to the secondary nodes for redundancy.

4.6 Configuring GlusterFS

The latest distributions (3.2+) included a specific package for Ubuntu and a few other Linux distributions. This made it easy to download and install, since the downloaded package could be installed simply by opening it in Ubuntu software center (or by using the *dpkg* command). For being able to experiment with Hadoop integration, the latest beta distribution (3.3 beta 2) was installed.

After restarting the nodes, the appropriate daemons were started automatically. To connect the nodes to each other the following command was run (from the master, for each node to add):

```
$ sudo gluster peer probe <hostname>
```

To check the status of the cluster, the following command was used:

```
$ sudo gluster peer status
```

The three nodes were then connected with each other. The next step was to create a volume to store data in GlusterFS. To create a volume replicated on two nodes the following command was used:

```
$ sudo gluster volume create <volume_name> replica 2  
  <host1>:<data_path1> <host2>:<data_path2>
```

This volume could then be mounted on a client and browsed in the same way as any other directory. A volume was mounted like so:

```
$ sudo mount -t glusterfs <host>:/<volume_name> <path>
```

The *<path>* is the path where the volume could be accessed. This directory had to be created before running the mount command. After a successful mount, it was possible to interact

with the GlusterFS volume through commands like `ls`, `mkdir` etc. Changes showed up on both data paths for the respective hosts listed when creating the volume.

4.7 Configuring Hadoop MapReduce

To enable running MapReduce jobs on the cluster there were some minor configurations to make. The Hadoop distribution contains both HDFS and MapReduce. When the steps in section 4.3 were completed, configuring and starting the MapReduce daemons was an easy task. Two properties needed to be added in `conf/mapred-site.xml` (relative to where Hadoop was extracted). The following content was added (for all nodes):

```
<property>
  <name>mapred.job.tracker</name>
  <value>sectra-VirtualBox:9001</value>
</property>
<property>
  <name>mapred.system.dir</name>
  <value>/mapred</value>
</property>
```

The first property sets the access point to the Jobtracker. The second property is the directory in the underlying file system (default is HDFS) where MapReduce would store data during jobs. When configured, running the script `bin/start-mapred.sh` will start the Jobtracker and Tasktrackers on the hosts listed in `conf/slaves`.

4.7.1 Integration

Hadoop is rather flexible and has many possibilities for third party products to integrate with the powerful functionality provided primarily by the Hadoop MapReduce framework. This section briefly describes how the tested products solved the integration, making it possible for Hadoop to access the stored data and write back the result. Also how GlusterFS could be used as the underlying file system instead of HDFS.

Integrating Cassandra and HBase was the most straightforward task since they are both written in Java and include integration packages in their API:s. The only configuration that had to be done was to add the respective libraries to `HADOOP_CLASSPATH` in `conf/hadoop-env.sh`:

```
export HADOOP_CLASSPATH=<cassandra_home>/lib/*
export HADOOP_CLASSPATH=`<hbase_home>/bin/hbase classpath`
```

After that it was only a matter of learning how to use the respective API:s.

Next was the integration with MongoDB, which required a little more work. First of all the plugin was downloaded as source code and not a JAR (Java Archive). When downloaded, the plugin was built using Maven. The resulting JAR file along with the MongoDB Java driver JAR was then copied into the Hadoop lib directory (could have added them to `HADOOP_CLASSPATH` instead).

Finally Hadoop was configured to use GlusterFS to store data. In `conf/core-site.xml` the property `fs.default.name` was changed to `glusterfs://<host>:<some_port>`. A bunch of properties used by the GlusterFS plugin was also added. As with MongoDB the plugin was downloaded as source code and built manually. The current implementation tried to mount the specified volume at a specified path during initialization. This caused failures. To solve it, the source code handling the automatic mounting was commented out, before recompiling the plugin. The volume was instead mounted manually on all nodes. With this modification the initialization worked fine.

5 Generating the Data Model

5.1 Purpose

The purpose of this part of the thesis was to make a more convincing case to use a distributed solution for storing images and the corresponding metadata. This proof of concept shows one of the possible applications of using distributed processing with Hadoop MapReduce on top of the distributed storage. By generating a model from the metadata, the process of migrating between different versions would become more efficient in the SHS.

5.2 The DICOM standard

In order to implement the proof-of-concept application some insight into the DICOM standard was needed. This section provides a summary that describes the relevant information needed to implement the model generation application.

5.2.1 Introduction

DICOM (Digital Imaging and Communications in Medicine) is a standard used for encoding, handling, storing, printing and transmitting information in medical imaging. The documents describing the standard contain several thousand pages [17]. Fortunately it was not necessary to read through it all since Sectra had introductory PowerPoint presentations explaining the most fundamental parts. The scope of this thesis was mainly limited to the storage of images.

5.2.2 Modules

For each type of image (e.g. CT, MR) there is a standard set of mandatory (M), conditional (C) and user optional (U) modules defined. A module is an abstract entity which is no more than a set of attributes for describing a certain aspect of the image's context. Examples of modules are patient information, image orientation and pixel data.

5.2.3 Attributes

At the lowest level, every DICOM file consists of a set of attributes. Each attribute consists of a tag, attribute name, value representation, value multiplicity, value length and the value itself. The standard describes exactly how each type of attribute shall be encoded and the purpose and content of specific attributes. The most important attributes needed for generating the data model are listed in Table 1. It is possible to insert attributes that are not defined in the standard. These attributes are generally only used by vendors and not used by others.

Tag	Name	Description
(0008,0018)	SOP Instance UID	Unique identifier for the image.
(0020,000E)	Series Instance UID	Unique identifier for the series that the image belongs to.
(0008,0060)	Modality	Short string describing the type of the image (e.g. "MR", "CT")
(0008,0033)	Content Time	One of the attributes describing when the image was created.
(0020,0032)	Image Position Patient	The position (three coordinates) of the upper left corner of the image, relative to the patient's coordinate system.
(0020,0037)	Image Orientation Patient	Row and column vectors (six coordinates total) describing the orientation of the image.

Table 1: Short description of the most important attributes used for generating the geometric model.

5.2.4 Libraries

To avoid delving in low level details when working with the DICOM files, a library was needed. A few different Java libraries were available, two of which were *dcm4che*¹⁰ and *PixelMed*¹¹. *Dcm4che* seemed to be a more complete library with many features with the disadvantage of being more complex. To get started more quickly, *PixelMed* was chosen since it had a more intuitive API with more than enough functionality to cover the needs for this application.

5.3 Migrating to MongoDB

Before being able to generate any data model there needed to be actual data. To populate MongoDB with image data, a simple Java application was written. It used the *PixelMed* API to extract information from DICOM files and the MongoDB driver to insert the same information into MongoDB. Sectra had a lot of anonymized test images (stripped of patient information) at their disposal, making it rather simple to migrate large amounts of data corresponding (mostly) to real cases.

All data except the pixel data was read from each file. Private attributes and overlay data were then removed, to save space in the database. The remaining attributes (a total size of about 3kB) were then put into a BSON document. The DICOM library supported conversion from most value types to a string. Attributes with multiple values were converted to one string where the values were delimited by a backslash character. A JSON representation of a document could look something like the following:

¹⁰ <http://www.dcm4che.org>, August 2012

¹¹ <http://www.pixelmed.com>, August 2012

```
{  
  "SOPInstanceUID": "1.3.46.670589.11.0.0.11.4.2.0.8111.5.4388",  
  "ContentTime": "112253.07",  
  "Modality": "MR",  
  "SeriesInstanceUID": "1.3.46.670589.11.0.0.11.4.2.0.8111.5.5332",  
  "ImagePositionPatient" : "-170.7\\-166.3\\-110.6",  
  "ImageOrientationPatient" : "0.999\\0.0\\0.011\\-0.0\\1.0\\0.0",  
  ...  
}
```

5.4 Considerations

The generated model is meant to better describe the physical relation among images that were taken at the same examination. If for example a part of a patient's arm is examined through a CT scan, many images are taken closely together to achieve a view of the whole volume. While the patient lies in the machine, several parts of the body may be examined, or the same body part may be examined from different angles.

The series of images is assigned a unique id called `SeriesInstanceUID` and is stored in the metadata of every DICOM image. Therefore it is easy to check if two images were taken at the same examination. The abstraction *DisplayUnit* is made to represent the whole set of images in an examination. A single image is represented by a *ViewRegion*. Knowing which *ViewRegions* that belong to a certain *DisplayUnit* is not always enough. To describe the physical relationship between *ViewRegions*, the *PhysicalRegion* abstraction is introduced. *ViewRegions* belong to the same *PhysicalRegion* if they have the same orientation in relation to the patient, determined by analyzing the geometry metadata of the DICOM images.

Except for volume stacks there are stacks that describe how a certain area changes over time. An example of this would be to examine soft tissue such as the heart by producing a set of consecutive MR images. To complicate things even further it is even possible to capture a volume that changes over time, i.e. the variation of the images is in four dimensions.

It is certain that images with different orientation do not belong to the same *PhysicalRegion*. Therefore this is the first thing to check when analyzing where to put a specific *ViewRegion*. The check if an image is a part of a temporal stack is made by simply checking for the existence of a certain DICOM-tag (`TemporalPositionIdentifier`).

5.5 Design

Using earlier simple experiments as a starting point for the model generator, adapting to more realistic data was rather straightforward. The implementation had the following design:

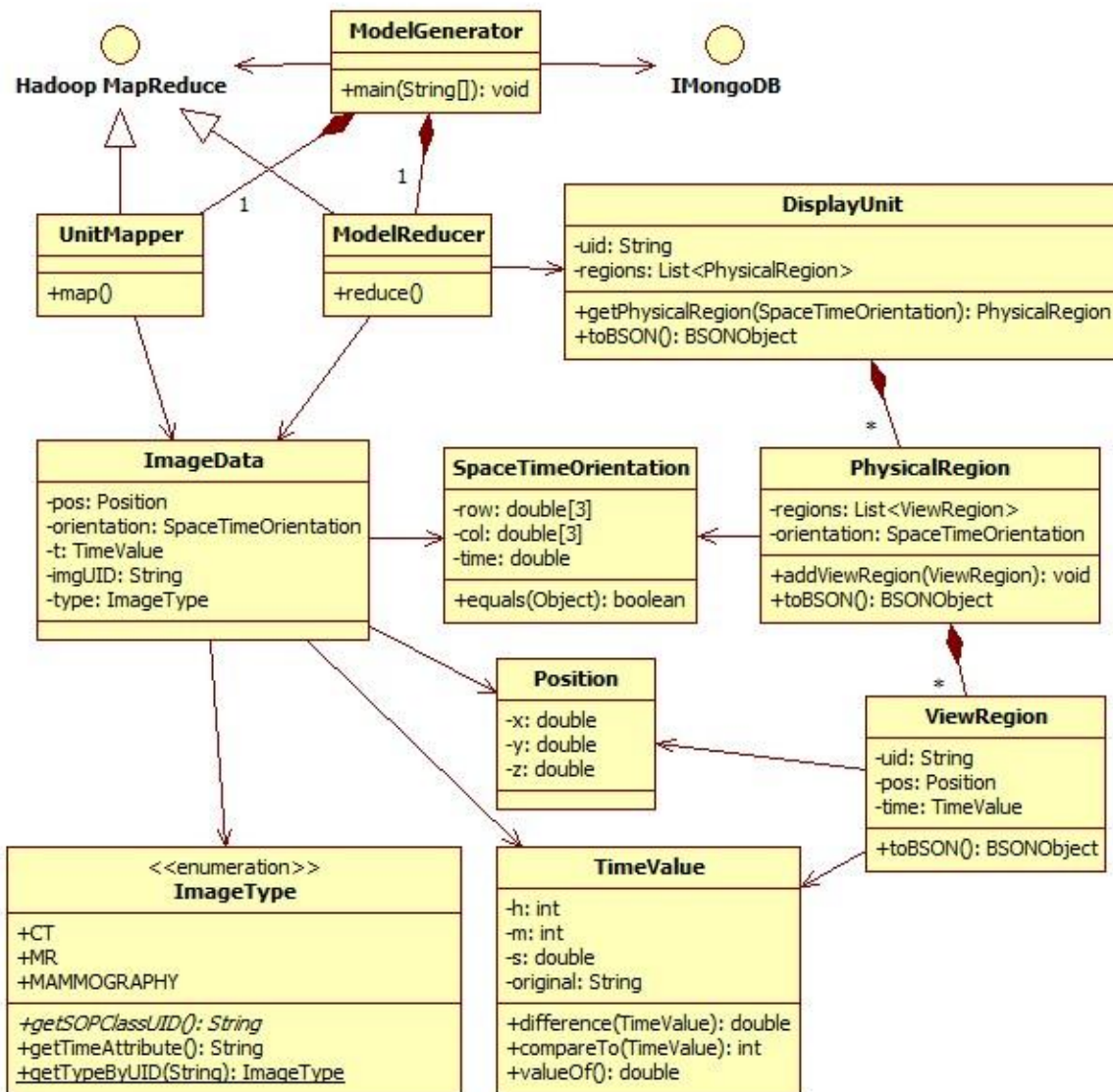


Figure 9: A simplified UML class diagram of the model generator

5.6 Details

The model generator is basically a Hadoop MapReduce job (written in Java) that extracts all relevant metadata from MongoDB, creates the model, and writes the result back to MongoDB. The chain looks like follows:

```

(key, BSON document) -> map() ->
(SeriesInstanceUID, ImageData) -> partitioning ->
(SeriesInstanceUID, Iterable<ImageData>) -> reduce() ->
(BSON document)

```

The `SeriesInstanceUID` is implicitly stored between the different steps by the framework since it is used as a key. The `uid` attribute of the `DisplayUnit` class corresponds to the `SeriesInstanceUID`. The code for the map and reduce functions as well as configuration of the Hadoop job can be viewed in appendix 8.1.

5.6.1 UnitMapper

The `UnitMapper` encapsulates the map function which is called for every document in the specified MongoDB collection. Extracting information from the BSON document is only a matter of using the API of the MongoDB Java driver. An `ImageData` object is used to encapsulate the necessary data. To be able to use `ImageData` objects as an intermediate form in the MapReduce framework, the `ImageData` class has to implement the `Writable` interface. This enables serialization and deserialization of these objects, which is used by the framework when temporarily storing or sending data between nodes.

The `ImageData` objects contain the following data:

- **Id:** The SOP Instance UID of the image, represented as a String.
- **Type:** The type of image (e.g. CT, MR). The different types are enumerated in `ImageType`.
- **Timestamp:** A value indicating when the image was created used for temporal stacks. Represented by the `TimeValue` class.
- **Temporal:** A boolean value indicating whether the image is a part of a time based stack or not.
- **Position:** The coordinates of the upper left corner of the image relative to the patient, represented by the `Position` class.
- **Orientation:** The orientation of the image, represented by the `SpaceTimeOrientation` class.

The `ImageData` object is mapped with the image's Series Instance UID used as the key.

5.6.2 ModelReducer

The partitioning groups all values with the same Series Instance UID as key, making it possible for the reducer to analyze the images in an examination and generate a model describing the physical relationship of the images. The reduce function is encapsulated by the `ModelReducer` class.

The model consists of three levels of abstraction. The `DisplayUnit` class represents an entire series of images. A `DisplayUnit` is composed of a set of `PhysicalRegions` that contain `ViewRegions` geometrically and/or temporally related to each other. Two images are defined as geometrically related if their orientations are parallel. A `ViewRegion` simply corresponds to one image. A `ViewRegion` only contains the id, position and timestamp of the corresponding image.

The model is generated by iterating through all the `ImageData` objects given to the reduce function, adding one `ViewRegion` at a time by checking if it can be put into an existing `PhysicalRegion` or if a new one needs to be created and added to the `DisplayUnit`.

When all `ViewRegions` have been added, a BSON representation of the model is generated and written to the specified output MongoDB collection.

5.7 Validation

To validate if the generated model was correct or not, another Java application was created. This was not a main requirement, but helped to demonstrate what the model generator accomplished. The visualizer starts by reverse-engineering the `DisplayUnit-PhysicalRegion-ViewRegion` model created in the reducing stage of the model generator, by fetching all generated documents from MongoDB. It was then possible to create 3D geometry for a `DisplayUnit`, using the Java3D API¹². Each `PhysicalRegion` is color coded to easily distinguish between them. The user can manually select which `PhysicalRegions` to display.

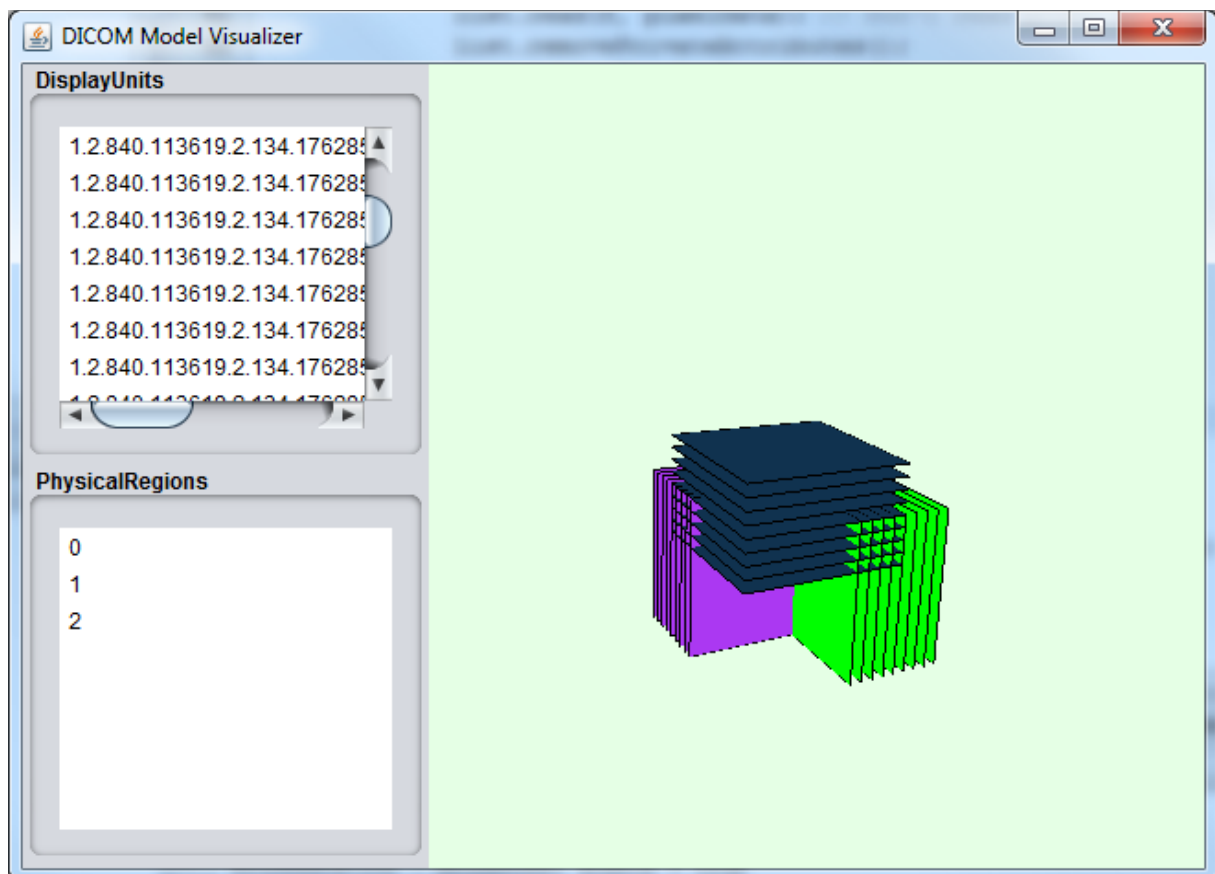


Figure 10: Screenshot of the DICOM model visualizer used for validating the generated model.

¹² <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html>, August 2012

6 Summary & Discussion

6.1 Summary

This thesis contained of two parts. In the first part ways of storing medical images in a distributed manner were investigated. The purpose with this was to explore ways to provide improved scalability and flexibility compared to the current storage structure. It boiled down to exploring two rather separate problems; storing the actual images, and storing the medically relevant metadata for each image. For the latter, several “non-relational databases” (NoSQL) were investigated, while a set of distributed file systems were considered for storing the actual images. A requirement for the distributed storage was to support some kind of batch processing, providing analytical capabilities that scales well as data sizes grow larger and gets spread among multiple nodes. Being the most popular framework for this purpose, Hadoop MapReduce was used successfully on all the investigated systems.

In the second part, an application utilizing the power of the Hadoop MapReduce framework was implemented. A stack of images are often used to describe a volume in the patient’s body or a change of a certain area over time. The task of the implemented application was to generate a model more closely related to this structure, by processing the metadata of all images in the database.

Apache Cassandra, HBase, MongoDB, HDFS and GlusterFS were the products investigated the most. After investigating the systems, there were not any clear winners. Each had their own set of strengths and weaknesses. In the simple architecture of Cassandra all nodes in the cluster have the same role, making it easy to scale by adding nodes in a live cluster. Some of the greatest strengths of HBase are the high performance concerning reads, and the use of the distributed file system to make data persistent. However, the cluster topology includes several different roles, adding much complexity compared to the amount of data to be handled. As opposed to the previous two, MongoDB is a document-based store. It is very flexible concerning both how data is stored and how to setup the cluster. For smaller databases it is enough to have a single database where data is replicated automatically among several nodes. The two file systems most investigated were HDFS and GlusterFS. Although HDFS is simple and works well with MapReduce, it has a master node as a single-point-of-failure, and is not as general as GlusterFS.

In the current structure of the Sectra PACS, there is a short-term and a long-term archive. For the short-term archive, performance is the most crucial factor to consider. Expandability of storage capacity is not a big issue since old data is moved after a certain time limit. A distributed storage would therefore not be a very good fit in this case, even if it would add flexibility through metadata storage and analytical capabilities through Hadoop MapReduce, in addition to using cheaper hardware. In the long-term archive however, minimizing the latencies is not as critical, and the amount of data is constantly growing. This makes it a good fit for replacing current SANs or object-databases etc.

6.2 Why NoSQL?

There are several reasons why a non-relational database fits well for storing DICOM data. Since the different images contain different sets of DICOM tags, storing all tags for every image would result in a large schema and a sparse table for a RDBMS. This is one of the cases NoSQL databases are meant for. There is generally no need for a strict schema, and empty values are simply not stored, which is efficient for sparse tables.

NoSQL databases are generally more suited for a distributed setting than relational databases. For large sites the data could exceed the size limit to fit on a single node, as they currently do. A failure would require manually recovering using a backup. MongoDB, Apache Cassandra and HBase all support distribution among multiple nodes, data replication, and automatic failover.

The biggest advantage with a RDBMS is the SQL language, enabling very efficient and extensive querying capabilities. The NoSQL databases generally only support rather simple queries, such as query by key. MongoDB provides good querying capabilities without the creation of indexes, making it a good replacement for a SQL database.

6.3 Short-term storage

In a production environment, the time it takes to randomly access data from the short-term archive greatly affects the overall performance of how responsive the whole system feels. There are rather tough requirements for the latency of the archive. Typically, a latency of no more than 10ms during a load of around 100 I/O operations per second is desirable. In this aspect, a SAN is probably the preferred choice since the benchmark did not show very high performance for GlusterFS. On the other hand GlusterFS nodes can be accessed in parallel from the clients, which would potentially improve performance during heavy load.

The number of images stored in the short-term archive differs depending on how big the individual site is, but it is typically in the order of tens or hundreds of millions. Both a SAN and a GlusterFS cluster are more than capable of handling this amount of data. The distributed alternative has the advantage of being cheaper. By using commodity hardware there is the additional advantage of having extra computational power that can be used for running MapReduce jobs for example. This significantly increases the flexibility of the system.

By having MongoDB running on the same machines as GlusterFS (or on an independent cluster), metadata could also be stored in a distributed way. At least in principle the MongoDB cluster could replace parts of the WISE database containing information about patients and their examinations among other things. There are several advantages with the distributed alternative. It supports scaling through automated sharding of data if there should be the need. In this case the amount of data is not that large (tens or hundreds of gigabytes) which means it could fit on a single node, but by having replicas of the database in

the cluster there is the possibility of balancing the load among the replicas to avoid bottlenecks. During heavy load WISE might be a bottleneck in the current system structure.

As a conclusion it might be better to stick with a SAN for short-term storage since minimizing latency is of great importance. The data is moved from the short-term storage after a limited time, with the consequence that the amount of data stored is more or less constant. This again goes in favor of the SAN, since there will not be a need for continuously adding expensive hardware.

6.4 Long-term storage

If an examination is not in the short-term archive when it is requested, all images belonging to this examination is fetched from the long-term archive and put into the short-term archive again before it can be viewed by the physician who requested the examination.

The performance requirements are not as critical for the long-term storage, since images older than a certain amount of time are not accessed that often. This brings the question whether it is worth investing a lot of money in a SAN, when the performance benefits are not giving the same value back to the system performance.

The continuously expanding amount of data makes it a good fit for a distributed storage, since it is made for scaling easily. Adding a node to a live cluster is supported for GlusterFS. Because of this and the fact that cheaper hardware compared to a SAN can be used, GlusterFS (or another DFS) is well suited for this type of storage.

There is a purpose for having a distributed database for the long-term storage as well, making it possible to do queries on what images that need to be fetched among other things. Some of this data is stored in the WISE database to be able to fetch old examinations. It would be beneficial to use a distributed database for the long-term archive since the data size is constantly growing. In a sharded MongoDB collection queries are distributed among several nodes, increasing the performance which is important for certain queries that are often requested.

6.5 MapReduce

Organizations such as Google, Yahoo!, Facebook and many more are using MapReduce for many diverse use cases including distributed sort, web link graph traversal, log file statistics, document clustering and machine learning. The number of common use cases is continuously growing [1]. Integrating Hadoop MapReduce with many different products and implementing the model generating job has given lots of positive experiences. The framework provides a lot of flexibility, both considering the configuration and the usage of the API. Implementing jobs in other languages is supported through Hadoop Streaming, which further proves the possibilities.

It should perform at its best when working with large files, since one map task is run for each file which introduces an overhead. For smaller data sets, or a large set of small files, the gain would lie more in simplifying the batch processing, rather than increased performance.

6.6 Future work

A large part of this thesis was spent reading about the products and their features, leaving less time to evaluate them. The evaluated products were only tested in a virtual environment, where the benchmark may not have been very accurate. It would be interesting to use them in a more realistic cluster setting, getting more accurate data about read/write speeds.

To further compare to the current solution it would also be appropriate with a more detailed cost evaluation.

During discussions with several employees at Sectra, potential usages for NoSQL databases in other situations were considered. For example using simple key/value stores to cache data locally, potentially increasing performance of the PACS. As mentioned in Section 3.1.7.2, graph databases could also have their applications. Therefore it would be of interest to investigate these as well.

7 References

- [1] S. Tiwari. *Professional NoSQL*. Wrox; 1st edition, 2011. ISBN: 047094224X
- [2] S. Ghemawat, H. Gombioff, S-T. Leung. The Google File System. In *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, Pages 29-43, 2003. <http://research.google.com/archive/gfs.html>, doi: 10.1145/945445.945450
- [3] Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, Gruber. BigTable: A Distributed Storage System for Structured Data. In *ACM Transactions on Computer Systems (TOCS)*, Volume 26 Issue 2, June 2008, Article No. 4. <http://research.google.com/archive/bigtable.html>, doi: 10.1145/1365815.1365816
- [4] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Communications of the ACM - 50th anniversary issue: 1958 – 2008*, Volume 51 Issue 1, January 2008, Pages 107-113. <http://research.google.com/archive/mapreduce.html>, doi: 10.1145/1327452.1327492
- [5] A. Lakshman, P. Malik. Cassandra - A Decentralized Structured Storage System. In *PODC '09 Proceedings of the 28th ACM symposium on Principles of distributed computing*, Pages 5-5, 2009. <http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf>, doi: 10.1145/1582716.1582722
- [6] R. Zuidhof, J. van der Til. Comparison of MapReduce Implementations. In *Proceedings 8th Student Colloquium 2010-2011*, Pages 60-65, 2011. http://altmetis.eldoc.ub.rug.nl/FILES/root/2011/eightproceedings_sme/studcol2011.pdf
- [7] E. Jansema, J. Thijs. Comparison between NoSQL Distributed Storage. In *Proceedings 8th Student Colloquium 2010-2011*, Pages 119-123, 2011. <http://www.mendeley.com/research/comparison-between-nosql-distributed-storage-systems/>
- [8] R. Henricsson. *MongoDB vs CouchDB performance*, 2011, <http://www.bth.se/fou/cuppsats.nsf/774564d376efaa0cc1256cecc0031533e/32737dee280f07ddc12578b200454a24!OpenDocument> (2012-09-04)
- [9] I. Varley. *No Relation: The Mixed Blessings of Non-Relational Databases*, 2009, http://ianvarley.com/UT/MR/Varley_MastersReport_Full_2009-08-07.pdf (2012-09-04)
- [10] J. Luciani. *Cassandra File System Design*, 2012, <http://www.datastax.com/dev/blog/cassandra-file-system-design> (2012-09-04)

- [11] C. Robinson. *Cassandra Query Language (CQL) v2.0 reference*,
<http://crlog.info/2011/09/17/cassandra-query-language-cql-v2-0-reference/>
(2012-09-04)
- [12] DataStax Enterprise, <http://www.datastax.com/products/enterprise> (2012-09-04)
- [13] *Apache HBase Reference Guide* - <http://hbase.apache.org/book.html> (2012-09-04)
- [14] MongoDB Manual - <http://www.mongodb.org/display/DOCS/Manual> (2012-09-04)
- [15] HDFS Architecture Guide -
hadoop.apache.org/common/docs/current/hdfs_design.html (2012-09-04)
- [16] GlusterFS Features -
http://www.gluster.org/community/documentation/index.php/GlusterFS_Features
(2012-09-04)
- [17] The DICOM Standard - <http://medical.nema.org/standard.html> (2012-09-04)
- [18] Giuseppe DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin,
Sivasubramanian, Vosshall, Vogels. Dynamo: Amazon's Highly Available Key-value
Store. In *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating
systems principles*, Pages 205-220, 2007.
<http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>,
doi: 10.1145/1294261.1294281

8 Appendix

8.1 ModelGenerator.java

```
package modelgenerator;

import com.mongodb.hadoop.MongoInputFormat;
import com.mongodb.hadoop.MongoOutputFormat;
import com.mongodb.hadoop.util.MongoConfigUtil;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.bson.BSONObject;

public class ModelGenerator {

    public static class UnitMapper extends
        Mapper<Object, BSONObject, Text, ImageData> {

        /**
         * Extracts relevant data from the BSON document and maps it
         * with SeriesInstanceUID as the key.
         * This way all images that belong to the same DisplayUnit
         * will be grouped together during the reduce phase.
         *
         * @param key
         * @param value
         * @param context
         * @throws IOException
         * @throws InterruptedException
         */
        @Override
        public void map(
            Object key,
            BSONObject value,
            Context context)
            throws IOException, InterruptedException {

            Text unit = new Text(
                (String) value.get("SeriesInstanceUID"));

            ImageData data = new ImageData();

            String modality = (String) value.get("Modality");
            data.setType(ImageType.getTypeByModality(modality));

            // Check if mandatory attributes exist
            if(!data.getType().checkValidity(value)) {
                return;
            }
        }
    }
}
```

```

data.setImgUID((String) value.get("SOPInstanceUID"));
TimeValue t = data.getType().getTimeValue(value);
data.setTime(t);

// Check if the image is part of a temporal series
data.setTemporal(
    value.containsField("TemporalPositionIdentifier"));

StringTokenizer st;
// Read the orientation of the image (relative to patient)
if (value.containsField("ImageOrientationPatient")) {

    st = new StringTokenizer(
        (String) value.get("ImageOrientationPatient"));
    double[] plane = new double[6];
    for (int i = 0; i < plane.length; i++) {
        plane[i] = Double.parseDouble(st.nextToken("\\"));
    }
    data.setOrientation(new SpaceTimeOrientation(plane, t));
}

// Read the position of the image (relative to patient)
if (value.containsField("ImagePositionPatient")) {

    st = new StringTokenizer(
        (String) value.get("ImagePositionPatient"));
    double[] pos = new double[3];
    for (int i = 0; i < pos.length; i++) {
        pos[i] = Double.parseDouble(st.nextToken("\\"));
    }
    data.setPos(new Position(pos));
}

context.write(unit, data);
}
}

public static class ModelReducer extends
    Reducer<Text, ImageData, Text, BSONObject> {

/**
 * Generates the model of the specified ImageData values,
 * belonging to the DisplayUnit identified by key.
 *
 * @param key The id of the DisplayUnit
 * @param values All ImageData values for the DisplayUnit
 * @param context
 * @throws IOException
 * @throws InterruptedException
 */
@Override
public void reduce(

```

```
        Text key,
        Iterable<ImageData> values,
        Context context)
        throws IOException, InterruptedException {

    DisplayUnit du = new DisplayUnit(key.toString());
    PhysicalRegion pr;
    ViewRegion vr;
    for (ImageData val : values) {

        vr = new ViewRegion(
            val.getImgUID(),
            val.getPos(),
            val.getTime());

        pr = du.getPhysicalRegion(
            val.getOrientation(),
            vr,
            val.isTemporal());

        pr.addViewRegion(vr);
    }

    // Write the result to MongoDB, by converting to BSON
    context.write(key, du.toBSON());
}

}

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

    Configuration conf = new Configuration();
    MongoConfigUtil.setInputURI(
        conf,
        "mongodb://localhost/test.image_data");
    MongoConfigUtil.setOutputURI(
        conf,
        "mongodb://localhost/test.model");

    Job job = new Job(
        conf,
        "mongo model generation");

    job.setJarByClass(ModelReducer.class);
    job.setMapperClass(UnitMapper.class);
    job.setReducerClass(ModelReducer.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(ImageData.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(BSONObject.class);

    job.setInputFormatClass(MongoInputFormat.class);
}
```

```
        job.setOutputFormatClass(MongoOutputFormat.class);  
  
        System.exit(job.waitForCompletion(true) ? 0 : 1);  
    }  
}
```

8.2 Abbreviations

There are many abbreviations used frequently in this thesis. Here follows a list that briefly explains most of them:

PACS – Picture Archiving and Communication System

DICOM - Digital Imaging and Communications in Medicine, standard for handling, storing, printing, and transmitting information in medical imaging. Used by Sectra among others.

NoSQL – Collective term for databases that are not relational in the traditional sense. Includes key/value stores, document stores, wide column stores, object databases, graph databases and more.

DFS – Distributed File System, general term used frequently.

HDFS – Hadoop Distributed File System, open-source counterpart to GFS.

GFS – Google File System, described in [2].

CFS – Cassandra File System, a HDFS compatible file system built upon Apache Cassandra, used by DSE.

URI – Uniform Resource Identifier, a string used to identify a name or resource.

DSE – DataStax Enterprise, a commercial product based on Cassandra.

JSON – JavaScript Object Notation, a lightweight data-interchange format.

BSON – Binary JSON, a binary-encoded serialization of JSON documents.

POSIX – Portable Operating System Interface, standard for maintaining compatibility between operating systems.

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>

© Tobias Dahlberg