Implementing an enterprise search platform using Lucene.NET

Fredrik Pettersson & Niklas Pettersson

LIU-IDA/LITH-EX-A--12/069--SE

2013-01-14
Master’s thesis

Implementing an enterprise search platform using Lucene.NET

Fredrik Pettersson & Niklas Pettersson

LIU-IDA/LITH-EX-A--12/069--SE

2013-01-14

Supervisor: Anders Fröberg
IDA, Linköpings universitet
Mats Haggren
Sectra Medical Systems AB

Examiner: Erik Berglund
IDA, Linköpings universitet
Abstract

This master’s thesis, conducted at Sectra Medical Systems AB, investigates the feasibility of integrating a search platform, built on modern search technology, into the complex architecture of existing products.

This was done through the implementation and integration of a search platform prototype, called Sectra Enterprise Search. It was built upon the search engine library Lucene.NET, written in C# for the Microsoft .NET Framework. Lucene.NET originates from the Java library Lucene, which is highly regarded and widely used for similar purposes.

During the development process a lot of requirements for the search platform were identified, including high availability, scalability and maintainability. Besides full text search for information in a variety of data sources, desirable features include autocompletion and highlighting.

Sectra Enterprise Search was successfully integrated within the architecture of existing products. The architecture of the prototype consists of multiple layers, with the search engine functionality at the very bottom and a web service handling all incoming request at the top.

To sum up, integrating a search platform based on modern search technology into the architecture of existing products infers full control of deployment, users searching in a more intuitive manner and reasonable search response times.
Acknowledgements

We would like to thank Mats Haggren, our supervisor at Sectra Medical Systems AB, as well as Anders Fröberg, our supervisor at IDA, Linköping University and the examiner Erik Berglund.

Also, to everyone at Sectra, thank you for all the help and support during the work with this thesis.

Thanks to family and friends for the support and shown interest.

Linköping, September 2012
Fredrik Pettersson & Niklas Pettersson
# Table of Contents

1. Introduction .......................................................................................................................... 1  
   1.1 Background .................................................................................................................. 1  
   1.2 Motivation ................................................................................................................... 2  
   1.3 Problem formulation ................................................................................................. 2  
   1.4 Limitations .................................................................................................................. 2  
      1.4.1 User interface ..................................................................................................... 2  
      1.4.2 Security measures ............................................................................................ 2  
      1.4.3 Deployment ....................................................................................................... 3  
   1.5 Disposition ................................................................................................................... 3  
   1.6 Related work .............................................................................................................. 3  

2. Methodology ....................................................................................................................... 4  
   2.1 Literature Study ......................................................................................................... 4  
   2.2 Agile software development ...................................................................................... 4  
      2.2.1 Identifying Requirements ............................................................................... 4  
      2.2.2 Implementation ............................................................................................... 5  
   2.3 Method criticism ....................................................................................................... 6  
   2.4 Source criticism ....................................................................................................... 6  

3. Theory .................................................................................................................................. 7  
   3.1. Search engines ......................................................................................................... 7  
      3.1.1 Full text search ................................................................................................. 7  
      3.1.2 Enterprise search ............................................................................................. 7  
   3.2 Lucene ......................................................................................................................... 7  
      3.2.1 Indexing ............................................................................................................ 9  
      3.2.2 Searching ....................................................................................................... 13  
      3.2.3 Lucene.NET ................................................................................................... 15  
   3.3 Distributed systems .................................................................................................... 15  
      3.3.1 Dependability ................................................................................................. 16  
      3.3.2 High Availability and Failover clusters ......................................................... 16  
      3.3.3 Storage Area Networks (SAN) ....................................................................... 16  
      3.3.4 Stateless services ............................................................................................ 16  
   3.4 Internet Information Services (IIS) .......................................................................... 17  
   3.5 Windows Communication Foundation (WCF) ...................................................... 17
**Table of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The Lucene architecture</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The steps of indexing using Lucene</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3</td>
<td>The steps of searching using Lucene</td>
<td>14</td>
</tr>
<tr>
<td>Figure 4</td>
<td>The Sectra PACS components</td>
<td>19</td>
</tr>
<tr>
<td>Figure 5</td>
<td>The Sectra PACS architecture for high availability</td>
<td>20</td>
</tr>
<tr>
<td>Figure 6</td>
<td>The architecture of SES</td>
<td>23</td>
</tr>
<tr>
<td>Figure 7</td>
<td>SES integration points in existing architecture</td>
<td>24</td>
</tr>
<tr>
<td>Figure 8</td>
<td>The integration layer of SES</td>
<td>29</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Disk usage of a Lucene index</td>
<td>34</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Search response times for System A</td>
<td>36</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Search response times for System B</td>
<td>36</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Search mean time with and without highlighting for System A</td>
<td>37</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Search mean time with and without highlighting for System B</td>
<td>37</td>
</tr>
</tbody>
</table>
1. Introduction

This master’s thesis work has been conducted at Sectra Medical Systems AB, located at Mjärdevi Center in Linköping, Sweden. Sectra Medical Systems AB, hereinafter referred to as Sectra, is a provider of industry-leading radiology IT, orthopaedic, osteoporosis and rheumatology solutions. Our task at Sectra has been to investigate and design a platform that is to work with every kind of Sectra client software with the means of searching for unstructured information. The information is typically obtained from Sectra databases, but could originate from other sources as well. In short, the motivation for this platform is to accelerate the retrieval of information on an intranet and at the same time making the search interface easier to use. The platform is being developed under the name Sectra Enterprise Search, hereinafter referred to as SES.

1.1 Background

Users of IT systems today expect search functions to work just as Google’s search functions. They expect being able to make a full text search on combinations of words or partial words, without having to specify which fields in the existing data structure they want to search in. However, they also expect to be able to choose a specific field to search within when they know what they are looking for (e.g. site:www.sectra.com). Users also expect to receive a response within milliseconds, regardless of how much information there is to scan.

Today, Sectra’s primary data storage is a relational database, which suits well for saving large amounts of data, but is less suited for modern search technology. All of Sectra’s customer sites are complex distributed systems, with multiple different servers and clients. The architecture includes replication, load balancing, different operating systems with different file systems, different types of relational databases and large physical distances with varying network quality.

The search functions in existing Sectra products are rather restricted, users are expected to explicitly specify which fields (e.g. name, date of birth) to search in and it is only possible to search in a predefined set of fields. The more advanced search functionality provides the user with a logical tree representation used to construct the query, which is not very intuitive and thereby may be difficult to master.
1.2 Motivation
Sectra finds high availability and scalability being very important aspects of their products and it is desirable to examine the feasibility of integrating modern search technology into the architecture of existing products. Through constructing a well-integrated search platform prototype, as a proof of concept, it will be investigated if and how this integration can be done according to such aspects.

Existing Sectra products are mainly targeted for Microsoft Windows and Sectra wants to find out if it is possible to construct a search platform based on techniques they already use, including the C# programming language and the Microsoft .NET Framework. The search engine library Lucene.NET satisfies such requirements. It originates from its Java variant Lucene which Solr, an existing enterprise search solution, is based on.

1.3 Problem formulation
The aim of this master’s thesis is to investigate the feasibility of integrating modern search technology within existing Sectra products through implementing an enterprise search platform prototype based on the Lucene.NET search engine library.

This leads to the following questions:

- What requirements, specified by Sectra and its customers, does such a search platform have to meet?
- What is an applicable architecture for such a search platform?
- What performance statistics will such a search platform implementation have?

1.4 Limitations
This section will mention a few things that were not implemented, in order to limit the scope of this master’s thesis.

1.4.1 User interface
This thesis will not include building the final Graphical User Interface (GUI). The ambition is to build the underlying search platform, but keeping the end-users in mind. This will enable other Sectra developers to easily build a GUI on top of the provided functionality in the future.

1.4.2 Security measures
Implementing security in a new product feature is by Sectra not considered a major obstacle. Solutions already implemented in the Sectra architecture can be reused, no effort on any new security solutions will be spent during the work on this product.
1.4.3 Deployment
The resulting product must not pose an increased deployment cost for Sectra during installation at customer sites. It should instead be embedded in the current deployment packages, but this will not be implemented within the boundaries of this thesis.

1.5 Disposition
An introduction to this thesis, with background information and motivations, is given in chapter 1. This chapter also contains the actual problem formulations as well as limitations of scope for this thesis.

Chapter 2 describes the work process and the methodologies used.

All facts necessary to grasp this thesis are explained in chapter 3. Everything in between search engines like Lucene and the existing Sectra architecture is covered.

The results of this thesis are covered in chapter 4. The architecture and performance of the resulting product Sectra Enterprise Search are discussed.

Reflections and more in-depth discussion are covered in chapter 5.

Summarized results along with some conclusion and final thoughts are available in chapter 6.

A dictionary is available in the Appendix.

1.6 Related work
Some enterprise search solutions already exist on the market, such as Solr or Sphinx. However none of these two are written in C# and they are not mainly targeted for the Microsoft .NET Framework. Solr is written in Java and Sphinx is written in C++. We will explain these two solutions briefly in section 3.8.
2. Methodology

This chapter describes the work and processes behind this thesis; including methodology, structure and workflow.

2.1 Literature Study

Since the main focus of this thesis was a rather heavy software implementation, the initial time was not spent on a large literature study. We put most emphasis on implementation and read things that we needed along the way. This included reading web pages, blog posts, forum posts, articles and books needed in order to grasp technical details and progress with the implementation.

Approaching the end of the thesis, we turned to literature to dig deeper into the theory, secure what we had done in literature and to be able to provide guidelines for problems that have not yet been resolved or that we think are appropriate to study further.

2.2 Agile software development

Sutherland and Schwaber (2012) claims that information is gained by observation and not prediction and we must frequently inspect where we are in order to adapt and optimize our results. The less frequently we inspect, the bigger is the risk that we have gone the wrong way. Through inspection and adaptation, we can avoid spending unnecessary time and effort on software that will not contribute to the final product. It is also stated that small teams of developers performs best with iterative incremental work.

Pair programming, a discipline within an agile development process called Extreme Programming (XP), is an approach where developers work in pairs at a single computer. Code produced by pair programming is of much higher quality and contains as much functionality as code produced by two separate developers. Pair programming is more of a social skill then a technical, and it requires a high cooperative competence. (Wells, 1999)

2.2.1 Identifying Requirements

To be able to determine the core features of the search service to implement, an initial list of requirements was established. Some requirements were very obvious to Sectra and thereby identified already before the thesis work were started. However, many things were still unclear. To understand the challenges of implementing the search service we needed to understand the architecture of existing Sectra products and the demands of the customers of Sectra. The list of requirements was extended in cooperation with key employees at Sectra through more informal interviews, i.e. open discussions. We ensured the participation of several employees responsible for a number of different areas, e.g. user experience, server architecture, product deployment.
Since, as mentioned in the introduction, users of search services expect them to work in a certain way; we had intentions to extend the list of requirements with such expectations. We wanted to determine which kind of features a modern search service should offer and how such features should work. We turned to giants like Google for inspiration and explored the features of existing enterprise search solutions.

Together with Sectra we found multiple areas within Sectra products where a modern search service would be desirable and applicable. To limit the scope of the thesis, it was decided that we should focus on a certain product area called Teaching Files, which is a rather new feature in Sectra products. It is tightly coupled to customer demands regarding doctors and scientists having the ability to tag interesting cases of examinations and associated radiology images with keywords. Tagging such cases may be useful for educational purposes, reference examples of certain phenomena, academic research or medical statistics. Together with employees, responsible for prioritization of customer demands in this product area, a number of attributes that should be available for searches were identified see appendix III.

The list of requirements was continuously reviewed and discussed with earlier mentioned key employees throughout the development process.

**2.2.2 Implementation**

Throughout this thesis we have been working with an agile approach, meaning that we have continuously tested our software and strived to have working software as often as possible. We have worked with an ever-changing list of tasks matching identified requirements. The list was frequently changing as we learnt more about how to build the search service.

The majority of the code was pair programmed; this was beneficial in several ways. Firstly, we did not have to read up on each other’s code as much and code was continuously reviewed by the other developer. The reviewing resulted in discussions around how to write a piece of functionality, which contributed to the quality of our code.

The search service was written in the programming language C#. We worked iteratively, refining the service in iterations. The iterations was not of a specific length, although commonly one or two weeks. We worked from a backlog of tasks, picking one task at a time to work with, broke it down into smaller tasks and prioritized them. To keep track of the status of our tasks we used an interactive web application called KanbanFlow (http://kanbanflow.com) for visualizing tasks, and categorizing them after status.

To ensure maintainable code, all code and associated documentation was written according to various internal standards provided by Sectra. We also strived to have unit tests for the core software functionality, to ensure no important feature was broken when refactoring code or implementing new features.
2.3 Method criticism

To be able to make a good evaluation of the search platform it would have been good to test other search engine implementations as well. No real comparison between different search platforms is available in this thesis. The search platform prototype is as earlier mentioned rather a proof of concept, to see if it is possible using Lucene.NET.

Agile methods are preferably used in teams of 3-9 people, and it is also useful to have team members with different areas of expertise (Sutherland & Schwaber, 2012). This has not been the case for this master’s thesis, but we do however feel that this approach has been beneficial for our working process.

2.4 Source criticism

Sources on Lucene, Solr and Sphinx were often very pro the described product. It was hard to find more objective views in proper sources, other than forums or blog posts. But we tried to see through that layer of marketing.

Sources on search engines rarely include enterprise search, but rather focuses entirely on web search engines. It was hard to find sources with good general information about search engines.

Besides the API reference, we only have a single source for the main theory about Lucene, the “Lucene in Action” book. However, the authors are contributing developers to the Lucene open source project and have the deepest understanding.
3. Theory

In order to implement our solution we needed to read up on some theories, these are summed up in this chapter.

3.1. Search engines

According to Encyclopædia Britannica (2012), a search engine is a computer program used to find answers to queries in a collection of information, e.g. a database. Search engines use so-called crawlers to extract information in order to build weighted indexes. Reliable results, to a given search query, are identified through different weighting or ranking strategies.

3.1.1 Full text search

Full text search is a technique that excels at searching in large volumes of unstructured data. A search engine using full text search goes through all words stored and tries to match these words against a given query. Documents or entities matching the query will be returned. (Ali, 2011)

3.1.2 Enterprise search

Enterprise search is the practice of making both structured and unstructured data within an organization searchable. This data may originate from variety of sources, e.g. databases, documents, emails. (Mukherjee, 2004)

Enterprise search within an intranet differs a lot from searching the web, especially in the notion of what a good search result is. When searching an intranet, commonly a “right” result exists as a specific document. In contrast, when searching the web, it is the best matching set of web pages that are relevant. Enterprise search may be more difficult, since finding the right answer is often more difficult than finding the best answer. (Mukherjee, 2004)

3.2 Lucene

Lucene is a popular and widely used open-source search engine library written in Java (Apache, 2012a).

Lucene is not a ready to use software or program, it provides a simple yet powerful Application Programming Interface (API) for indexing and searching and the rest is up to the developer (Hatcher, Gospodnetić & McCandless, 2010).

Hatcher, Gospodnetić and McCandless (2010) states Lucene uses a conceptual document model. The fundamental unit of indexing and searching in Lucene is called a document, which becomes very clear when browsing the API (Apache, 2012b).
Lucene is a high performance, scalable Information Retrieval (IR) library, which includes functionality to search for documents, information within documents or metadata about documents (Hatcher, Gospodnetić & McCandless, 2010).

The architecture of Lucene is visible in the figure below, see Figure 1. The Lucene architecture.

Figure 1. The Lucene architecture

Source: Figure by authors, based on Hatcher, Gospodnetić and McCandless (2010).
3.2.1 Indexing

In order for Lucene to work with certain data, the data must first be extracted and converted into the earlier mentioned conceptual document model. According to Hatcher, Gospodnetić and McCandless (2010), the fundamental steps of indexing include:

- Extracting texts from raw data.
- Putting texts into fields of documents that Lucene understands.
- Analyzing document fields.
- Storing documents in the Lucene index.

The figure below is illustrating the steps of indexing using Lucene, see Figure 2. The steps of indexing using Lucene.

![Figure 2. The steps of indexing using Lucene](image)

Source: Figure by authors, based on Hatcher, Gospodnetić and McCandless (2010).

3.2.1.1 Documents

A document, the fundamental unit of Lucene, is basically a set of fields. Each field has a name and a textual value (Apache, 2012b). You could optionally look at a document as a database row where the fields are the columns.
3.2.1.2 Fields

A field is a section of a document. Each field has two parts, a name and a value. Fields are optionally stored in the index, so that they may be returned with hits on the document. (Apache, 2012b)

Indexing fields

Indexing fields may be done in three different ways called “analyzed”, “not analyzed” or “no indexing”. If a field is set as “analyzed”, Lucene will index the result of the field value being passed through an analyzer (see section 3.2.1.3 Analyzing). This is the most suitable option for regular text. If a field is set as “not analyzed”, the field value will be indexed without using an analyzer. The value will be indexed as a single term which is useful for fields like a unique ID field. If “no indexing” is used, the field value will simply not be indexed. This field can thus not be searched, but one can still access its contents if it is stored (see section below on Storing fields). (Hatcher, Gospodnetić & McCandless, 2010)

You can also disable the storing of norms (see section
Boosting. No norms means that field and document boosting is disabled, with the benefit of less memory usage during searches. (Hatcher, Gospodnetić & McCandless, 2010)

**Storing fields**

In Lucene you may specify if a field should be stored or not. If you choose to store the field, Lucene will store the field value in the index. This is useful for short texts, like a document's title, which should be displayed with the results. The value is stored in its original form, i.e. no analyzer is used before it is stored. If you choose to not store the field, Lucene simply does not store the value in the index. (Hatcher, Gospodnetić & McCandless, 2010)

**3.2.1.1 Storage**

Lucene stores its index in a so called directory, which represents the location of a Lucene index. The index may be stored in a number of different ways. The most common way to store a Lucene index is simply to store files in a file system directory. Lucene could also hold all its data in memory, using a RAM directory, which is useful for an index that can be fully loaded in memory and lost upon application termination. Storing all data in fast-access memory is suitable for situations where you need very quick access to the index. The performance difference, between using a RAM directory or a file system directory, is greatly reduced when run on operating systems with in-memory file cache. (Hatcher, Gospodnetić & McCandless, 2010)
3.2.1.2 Index managing

All modifications to a Lucene index is done through a so called index writer. It is central when creating an index or when opening an existing index to add, remove or update documents. Only a single writer may be open on a Lucene index at a time. A file based write lock is obtained when a writer is opened and released only when it is closed. (Hatcher, Gospodnetić & McCandless, 2010)

Basic operations

Inserting a document into the Lucene index is rather straightforward. First, create an empty document. Add the desired fields, with names and values, to the document. Then provide the document to the Lucene functionality responsible for adding documents to the index. (Hatcher, Gospodnetić & McCandless, 2010)

To delete a document Lucene first needs to find the specific document. This is done by providing a search term to single out that document. Because of this, it is very useful to specify an ID field with a unique value for each document. (Hatcher, Gospodnetić & McCandless, 2010)

When Lucene is requested to delete a document, the document is only marked as deleted and thereby blacklisted in searches. This is a very quick operation, but the downside is that the marked document still consumes disk space. The document is not actually deleted until index segments are merged, i.e. by an explicit optimization. (Hatcher, Gospodnetić & McCandless, 2010)

To update an indexed document, Lucene basically performs a delete followed by an insert. Just as for the delete operation, a search term is needed to identify the document to update. (Hatcher, Gospodnetić & McCandless, 2010)

Index segments

Every Lucene index consists of one or more segments. Each segment is a standalone index itself containing a subset of all the documents. Segments are created when a writer make changes to the index. There is always one important file referencing all live segments, which Lucene uses to find each segment. During searches, each segment is visited and the results are combined together. (Hatcher, Gospodnetić & McCandless, 2010)

Optimizing

Since a Lucene index commonly contains many separate segments, Lucene is forced to search each segment separately and then combine the results during searches. The optimize functionality basically merges many segments down to less segments. It also reclaims disk space consumed by pending document deletions. Optimizing the index will improve search performance, since less results needs to be combined. It is a tradeoff between a larger one-time cost and faster searching. It is probably worthwhile if the index is rarely updated and there is a lot of searching between the updates. (Hatcher, Gospodnetić & McCandless, 2010)
**Boosting**

Boosting is a technique to tune the relevance of specific documents or fields. To store boosts, Lucene use so called norms. Lucene computes a default boost based on the tokens of each field, where shorter fields have a higher boost. It is also possible to specify boosts during indexing. All boosts are combined and encoded into a single byte which is the value stored as norms per field per document. When searching, norms for the searched fields are read and decoded back into a number, which is used when computing the relevance score. It is also possible to specify new boosts when searching to handle specific search preferences. Noteworthy, norms result in high memory usage at search time since all norms are loaded into memory. (Hatcher, Gospodnetić & McCandless, 2010)

### 3.2.1.3 Analyzing

Before a text fragment is indexed using the index writer, it is first passed through a so called analyzer, see Figure 2. The steps of indexing using Lucene. Analyzers are in charge of extracting only the important tokens out of a piece text. Such tokens are written to the index and the rest are not. (Hatcher, Gospodnetić & McCandless, 2010)

Lucene allows you to write your own analyzer implementation, but comes with several analyzers by default. Useful analyzers skip stop words (e.g. frequently used words without any specific meaning such as punctuations, “a” or “the”) and convert text to lowercase letters to avoid case-sensitive searches. As mentioned earlier, only the textual results returned from the used analyzer are indexed and thereby only analyzed text is matched against when searching. Analyzing is a very important part of Lucene, since analyzers contain a lot of smart logic to improve the relevance of search result. (Hatcher, Gospodnetić & McCandless, 2010)

Analyzing occurs at two occasions: during indexing, on all text fragments, and when using a query parser for searching, on the text fragments of the search query in order to find the best possible matches. (Hatcher, Gospodnetić & McCandless, 2010)

Lucene contributors have also provided a lot of different language specific analyzers. This is useful since different languages have different sets of stop words. (Hatcher, Gospodnetić & McCandless, 2010)

### 3.2.2 Searching

In order for users to perform searches using Lucene, the user-entered query must be transformed into query objects that Lucene understands. According to Hatcher, Gospodnetić and McCandless (2010), the fundamental steps of searching include;

- Parsing user-entered query expressions.
- Analyzing query expressions.
- Building proper query objects.
- Searching in the index.
- Reading the results.
The figure below illustrates the steps of searching using Lucene, see Figure 3. The steps of searching using Lucene.

![Figure 3. The steps of searching using Lucene](image)

Source: Figure by authors, based on Hatcher, Gospodnetić and McCandless (2010).

### 3.2.2.1 Reading and Searching

To read from a lucene index one needs a so called index reader. Multiple index readers can read from the same index at once, so multiple processes can search the same index concurrently. They can even read from an index while a writer is writing changes to it, but they will not obtain changes until the writer is completely done. (Hatcher, Gospodnetić & McCandless, 2010)

To search an index, Lucene provides so called index searchers, which expose several search methods. An index searcher, with access to an index reader, can perform searches on a given search query and return the results. (Hatcher, Gospodnetić & McCandless, 2010)

### 3.2.2.3 Querying

All querying in Lucene is done through specific query objects. Lucene supports a number of different query object types, where the most basic type is used to search for a term within a specific field. However, the most common approach is to use a so called query parser, which builds an appropriate query object from a user entered query expression. Query parsers are designed to handle human-entered text rather than program-generated text. The specific Lucene syntax can be found in appendix IV. Lucene provides a number of different query parsers that builds slightly different queries. An example is the multi-field query parser, which is used to build query objects that searches in multiple fields. This is very useful when it is desired to query for a term regardless of which field it might be stored in. (Hatcher, Gospodnetić & McCandless, 2010)
3.2.2.4 Search results

By default, Lucene sorts the search results in descending relevance score order, where the most relevant hits are appearing first. Lucene computes this numeric relevance score for each document, given a specific query. The actual documents are not returned at first, rather references to them. It is rarely necessary to retrieve all documents, since a user typically only is interested in the first few. So Lucene provides functionality to just pick the first relevant, desired number of documents. However, Lucene can tell you how many hits a query got in total, which a searching user may be interested in. For large indexes, retrieving a lot of documents is rather time consuming and requires a lot of memory. (Hatcher, Gospodnetić & McCandless, 2010)

3.2.3 Lucene.NET

The Lucene search engine library has been ported to several other programming languages. One of the ports is called Lucene.NET, which is targeted for the Microsoft .NET Framework users. Lucene.NET is a line by line port from Java to C#, this to make it easier to keep up with the official Java releases. However, Lucene.NET is generally some version numbers behind. One of the goals of Lucene.NET is to take advantage of special features of the .NET framework in order to maximize its performance on Windows. (Apache, 2012c)

3.3 Distributed systems

A distributed system consists of components distributed on computers interconnected in a network; this may be an internal network or external network such as the Internet. Components in a distributed system communicate and coordinate their actions by sending messages. The main motivation for distributed systems is resource sharing. Resources usually involve both hardware and software, such as printers, databases, processing power or storage systems. (Coulouris, Dollimore and Kindberg, 2005)

Concurrent program execution is an essential aspect of a distributed system meaning that different components in the system can process their work independently of each other and are able to share or ask for resources when necessary. A well-designed distributed system should never fail due to the failure of one or more components. If this is achieved it is said that the system is fault tolerant. A failure can imply a crash of a component, a component producing erroneous data or a component becoming isolated due to a network error. (Coulouris, Dollimore & Kindberg, 2005)
3.3.1 Dependability

Dependability is a concept of describing the trustworthiness of a computing system and it can be described with the use of following attributes (Avizienis, Magnus & Laprie, 2000):

- **Safety** - Absence of harm to people and environment.
- **Availability** - The readiness for correct service.
- **Integrity** - Absence of improper system alterations.
- **Reliability** - Continuity of correct service.
- **Maintainability** - Ability to undergo modifications and repairs.

3.3.2 High Availability and Failover clusters

A server cluster is a closely-coupled group of interconnected computers running a server application (Coulouris, Dollimore & Kindberg, 2005). Clusters are often used to host services demanding power beyond the capability of a single machine. One area of use is highly available and scalable web services.

Similarly, a failover cluster is a group of interconnected computer nodes that work together to increase the availability of applications and services (Microsoft, 2011). If one of the cluster nodes fails, e.g. through computer or application crash, another node detects the faults and immediately begins to provide the service instead (Coulouris, Dollimore & Kindberg, 2005). Microsoft (2011) states this process is known as failover and also concludes that such processes will result in minimum disruptions of service.

3.3.3 Storage Area Networks (SAN)

A Storage Area Network (SAN) is a dedicated, high-speed network connecting servers and storage devices. It replaces the traditional server-to-storage connections as well as restrictions to the amount of data that a server can access. Using a SAN enables many servers to share a common storage utility containing many different storage devices. Hosts and applications sees and can access the storage devices attached to the SAN in the same way as it would deal with a regular, locally attached, storage device. A SAN provides physical connections (communication infrastructure) and organizes all the elements (management layer) in a way ensures that data transfer is secure, fast and robust. (IBM, 2006)

3.3.4 Stateless services

A stateless service is one who does not save any information about an ongoing connection or communication with a client. This meaning that it does not matter to the clients which process or server it communicates with, due to the fact that the server holds no state. If a process or a server fails there is nothing to recover and the client can simply re-transmit the request and get served by another process. (Coulouris, Dollimore & Kindberg, 2005)
3.4 Internet Information Services (IIS)

Internet Information Services (IIS) is a web server application created by Microsoft, which makes it easy for Microsoft Windows users to host their own web applications. IIS has a request-processing architecture which contains protocol listeners, listening for requests made to the server and worker processes, responsible for executing the actual work. When a listener receives a protocol-specific request, it forwards the request to the IIS core for processing and then returns the response to the requesting client. The IIS process managing system spawns new worker processes to handle incoming requests if an available worker process does not already exist. This behavior, and the number of worker processes to be used at a time, is configurable. (Templin, 2012)

By default, IIS supports the HTTP and HTTPS protocols, but it is possible to integrate frameworks like Windows Communication Foundation (WCF) with IIS to support services and applications that use other protocols. (Templin, 2012)

3.5 Windows Communication Foundation (WCF)

The Windows Communication Foundation (WCF) is a framework of the .NET Framework, designed for building service-oriented applications. WCF unifies different communication approaches, which simplifies development of distributed applications on Windows. (Chappell, 2010)

WCF can interact via Simple Object Access Protocol (SOAP) (Chapell, 2010). SOAP is a protocol for exchanging data using Extensible Markup Language (XML) that is commonly transmitted over HTTP (W3, 2007). Because WCF interacts via SOAP, binary protocols or a number of other ways; WCF simplifies interoperability with other platforms (Chapell, 2010).

To host a WCF service you need; a service class implementation, a host process and at least one endpoint (Chapell, 2010).

Each service class implements at least one service contract, which basically is a definition of the operations (methods) that are exposed by the service. The class can also provide a data contract which defines the type of data which can be consumed by the operations. (Chapell, 2010)

The service needs to be run within a host process, which could be an arbitrary process, but most commonly is the Internet Information Services (IIS) (Chapell, 2010).

All communication with WCF services goes through the service’s endpoints, which allow clients to access the service. Every endpoint specifies; an address, a binding and a service contract name. The address is a uniform resource locator (URL) that identifies the endpoint of a machine. The binding describes what protocol to be used when accessing the endpoint. The contract name specifies the service contract exposed through the endpoint. (Chapell, 2010)
A very common endpoint binding in WCF is the BasicHttpBinding, which by default sends SOAP over HTTP. However, to be able to make asynchronous operations, one will have to turn to another binding. A duplex binding like the WsDualHttpBinding, which also can send SOAP over HTTP, makes it possible to link together operations. This allows both sides of the communication to act as client and service. A client can call the service to start an operation, and when the operation is completed, the service can use the linked callback operation to notify the client of the result. (Chapell, 2010)

WCF provides functionality to configure how a service will deal with concurrency. Basically, a service can handle each request in separate compartments (e.g. threads, processes) or all requests in the same one. This is configured in the service class implementation. (Microsoft, 2012)

The instance context mode options in WCF controls how service instances are created and destroyed. If the mode is set to “per call”, a new instance of the service will be created upon each client operation request (method call). It will be instantly destroyed when the operation is completed. If set to “per session” instead, the same instance will be used to handle all requests from a specific client. The mode can also be set to “single”, which causes a single instance of the service to handle all operation requests from all clients. (Chapell, 2010)

Concreancy mode options in WCF controls the number of threads running in an instance context at the same time. (Microsoft, 2012)

If the mode is set to “single”, only one client request at a time is allowed to this service, it will thereby be single-threaded. If, on the other hand, the mode is set to multiple, WCF allows more than one client request at a time. Each request will then be handled concurrently on different threads, which requires the service implementation to be thread-safe. (Chapell, 2010)
3.6 The Sectra architecture

Through discussions with employees at Sectra and by reading internal documentation on the existing architecture of Sectra products, we have identified the part of the architecture that we need to understand and make use of to be able to integrate our search service.

The product in focus for this thesis was the Sectra Picture Archiving and Communication System (PACS). The three major components in Sectra PACS are:

- **Workstations**
  Thin clients for user interaction.
- **Application servers**
  Servers containing the main business logic, serving requests from workstation clients.
- **Core servers**
  Servers for accessing information databases along with file and image storages.

The components, which can be seen in Figure 4. The Sectra PACS components, are further described in the sections below.

![Figure 4. The Sectra PACS components](source)

Source: Figure by Sectra Medical Systems AB

3.6.1 Workstations

The workstations are the end user interface of Sectra PACS. The workstation is a thin client running against the server software located on the application servers. Before startup, workstations download the latest version of the client software from the application servers. The most common workstation client is called IDS7, but a number of other clients exist.
3.6.2 Application servers

The application servers contain the business logic for workstations and handles communication with databases and file storages, located on the core servers. It is at the application servers most of the information processing occurs.

Commonly, multiple application servers run alongside, but are designed to appear as a single node to clients. Using hardware load balancers taking care of eventual failover; makes the servers achieve high availability. It also makes it easier to add more application servers if needed, e.g. when the amount of clients increases. This is possible without changing the architecture of the system, which makes the application servers highly scalable.

3.6.3 Core servers

The core servers are a High Availability (HA) cluster responsible for storing examination and patient data, along with other relevant information. The most important core server component regarding patient information is the Workflow Information Service Engine (WISE). The servers achieve this high availability through a standard Windows based dual-node failover cluster architecture, every node can use an arbitrary amount of servers, but in the simplest case a node consists of one server. One node handles all the necessary processing, but all information is stored in a common Storage Area Network (SAN) and the other works as a failover node.

![Figure 5. The Sectra PACS architecture for high availability](image)

Source: Sectra Medical Systems AB.
3.7 Existing solutions

A number of enterprise search solutions already exist on the market. We will briefly discuss the two most relevant platforms; Solr and Sphinx.

3.7.1 Solr

Apache Solr is an open source enterprise search platform. It is built upon Lucene and includes features like full text search, highlighting, faceted search, database integration and various document handling. Solr is highly scalable and provides distributed search and index replication. Solr is ready to use out of the box and runs as a standalone full text search server. Solr is fully written in Java. Ports or wrappers are available in several languages, but they are still based on and needs the Java runtime environment. (Apache, 2012d)

3.7.2 Sphinx

Sphinx is a search engine platform written entirely in C++ and is like Solr ready to use out of the box (Ali, 2011). Sphinx uses an advanced full text searching syntax for parsing user queries. Sphinx is widely used, and is a free to use software. However, for enterprise use a license may need to be purchased (Sphinx, 2012).

The engine is designed for indexing database content. It currently supports data sources like MySQL, PostgreSQL and ODBC-compliant databases. Other data sources can be indexed through a custom XML format. (Ali, 2011)

Ali (2011) also states that Sphinx has been proven to be highly scalable, just by adding computational power to each Sphinx node or extending with additional nodes.
4. Results

In this chapter we will present the actual results of this master’s thesis. We will cover identified requirements, architecture, design decisions and performance of the main result; the produced enterprise search service software called Sectra Enterprise Search (SES).

4.1 Identified requirements

Throughout the development process a lot of new requirements and feature requests emerged from different people, departments and customers of Sectra. The full list of identified requirements is available as an appendix II.

When narrowing the scope to the Teaching Files product area, it was easier to determine what kind of information that users may be interested in searching for. Together with Sectra employees and customers, a list of information to make searchable were produced, see appendix III. However, only the most interesting subset of the enlisted information was selected for inclusion in the implementation.

4.2 Architecture

In this section we will go through architecture and features of Sectra Enterprise Search (SES). SES is divided into three main layers with the following fundamental components:

**Core**

The search engine layer.

- **Search engine** containing all core functionality.
- **Index** storing all necessary information.

**Business logic**

The data specific layer.

- **Data source crawlers** moving data from sources to the data models of SES.
- **Index managers** executing indexing operations.
- **Query builders** parsing search terms and building proper search queries.

**API**

The interface layer.

- **API** exposing SES through a web service for use by other systems.

The main layers are visible in Figure 6. The architecture of SES, and are individually described further in later sections of this chapter.
4.2.1 Integration into existing architecture

The biggest influence when designing SES was the fact that it had to work well with the existing architecture of Sectra products, e.g. Sectra PACS. Parts of the system need to be interchangeable, if one part would need to be replaced in the future, for example the search engine. This should be possible without rebuilding any other parts of the system. It should also be easy to expand the system easily if a new area of interest arrives. The new area might require new information that needs to be searchable, and such information could be located in any kind source, such as PDF documents, HTML pages or databases containing relevant information. It might also require supporting a new client being able to access our system.

To fulfill demands on failover, SES consists of several processes, distributed on different machines. The load is split among a set of servers to ensure high performance and high availability. The load balancing mechanism is a hardware solution already integrated with existing Sectra products.

We are using the existing application and core servers. SES will run on the same application servers as the Sectra Healthcare Servers (SHS). SES is built to be stateless, which means that even if one of these processes or servers fail, no information will be lost, the client will...
eventually understand that the process is down, contact another one and continue working as usual. Either a request goes through, or the client will have to repeat it until it does.

To obtain high availability for the search engine index, it was stored on a Storage Area Network (SAN). SES accesses the SAN through the core servers, which are located in a high availability cluster.

The integration points are visible in the figure below, see Figure 7. SES integration points in existing architecture More on how the integration was done is available in section 4.2.5.

Figure 7. SES integration points in existing architecture

Source: Sectra Medical Systems AB with modifications by authors.
4.2.2 The core layer

The core layer in SES, seen at the bottom of Figure 6. The architecture of SES, is based on two important components; the search engine and its index. The search engine contains functionality, such as search and index managing operations, which is crucial for many other components in the architecture.

The core also contains a set of classes to reduce dependencies of the used search engine implementation. We have specified our own document and field types, to represent search engine data, along with specific result types used when returning search hits, autocomplete suggestions or general results. The core layer only communicates through this set of classes and thereby we have introduced a layer of abstraction between the layers using the core and the actual search engine implementation.

The possibility to switch the search engine implementation in the future makes the system more maintainable, which is why the search engine always is accessed through an interface. We created a specific implementation for Lucene, extending our interface. The implementation is a wrapper class around the search engine library Lucene.NET. Only the search engine wrapper talks a search engine specific language. So it would be easy to replace Lucene with another search engine, simply by creating another implementation extending the interface and wrapping the desired search engine.

The index acts as the storage of the service, being the component containing all the required data. The index is very search engine dependent, since different search engines may store things very differently. That is why the search engine specific wrapper is the only component which actually is reading from and writing to the index. The index could be stored on disk or in RAM, it is all depending on the search engine implementation. The physical location of the index could be anywhere, since it is accessed through a URL which is configurable.

Our specific Lucene.NET implementation uses a file system directory index which is stored on a Storage Area Network (SAN). We access the Lucene-specific index through a disk attached to servers in the high availability cluster (also known as the core servers). By storing the index on a SAN, accessed through the earlier mentioned servers, we can obtain fast access to our index and backup.

To deal with several processes accessing the index at once, we used functionality provided by Lucene to lock the index for writing, while performing updates to it. However, it is still possible to read from the index while it is locked.

Besides the more obvious capabilities of a search engine, like indexing and searching, the search engine interface also specifies a set of extended features including; autocompletion and highlighting. Autocompletion is a utility which basically works as a simple search function. The concept is to return a set of term suggestions from existing field values, where the suggested terms are matching a given search term. Fields that should be available for autocompletion are stored in a separate index, an index dedicated for autocompletion, alongside the regular index.
This enables certain autocomplete-specific features, like returning suggestions in alphabetic order instead of relevance, and also to improve performance. Highlighting is another helpful utility which can be used together with the regular search functionality. The utility highlights the parts of text fields which are matching a given search term. This to display where and why a search hit (document) is relevant. However, highlighting is a rather expensive feature, especially when obtaining many search hits. This is because all hits, in all relevant fields, of all relevant documents have to be highlighted. So, in our specific implementation, we made it easy to configure if this functionality should be enabled or disabled.

4.2.3 The business logic layer

On top of the core layer, we created a middle layer containing the business logic of SES, see Figure 6. The architecture of SES. This layer is responsible for data source crawling, index managing and querying. Operations which differ slightly depending on what type of data they deal with. Components within this data specific layer need to be able to; represent the data of different sources, as well as communicate with both the underlying core and the overlying interface layer.

We created our own data model types to be able to represent the relevant data found in the data sources. Data sources could basically be anything from text documents to entire databases, but in our specific implementation for Teaching Files this typically was certain columns of tables stored in a relational database. The data mapping reduces the dependencies of the underlying data, introducing a layer of abstraction between the data consumers and the actual data. This makes it easier to further extend the service with more data in the future if needed. A single data model is also very useful when representing data which originate from several different data sources.

The components, of this data specific layer, make use of the underlying core for search engine related operations, but in a way that exploits the knowledge of specific data type characteristics. The components convert data into types needed in order to communicate with the core, such as documents. Also, general results returned from the core, are converted back into the specific data models before they are returned to components of the overlying interface layer.

For each type of data that should be handled by SES, we first have to create its data model, which we in our case chose to call Teaching File data. The data model specifies the properties of the data, along with what is unique for each entity of this data type (specified through a property called ID). Also, the data model specifies which properties that should be available for searching and autocompletion. For each property it specifies how it should be stored in the index and what relevance it has in searches (the so called boost). In our case we had properties describing examination information, patient information, keywords and comments with different boosts. Autocompletion was enabled for properties like names and keywords.

To extract a certain type of data from relevant data sources into its representational data model, SES needs a data source crawler associated with that type of data. The crawler knows of relevant data sources and is able to extract all its information, or the latest changes, for
conversion to the SES related data model. Thereby, crawlers are very important when SES initially creates its index as well as when continuously synchronizing its index with data sources. In order to enable Teaching File related searching through SES, we created a Teaching File data source crawler which connects to the WISE database, a relational database accessed through the core servers. It is able to convert and merge the database entries into entities of the Teaching File data model.

To insert a certain type of data into the index, SES needs an index manager associated with that type of data. The index manager is responsible for executing the changes found by the data source crawler. Changes like insertions, updates or deletions of entities have to be reflected in the index. The index manager converts the changed entities, from data models to search engine data, and uses the underlying core to perform the needed operations on the index. The index manager contains the business logic needed to build an efficient index. We created a Teaching File index manager, which obtains changes from the Teaching File data source crawler and converts Teaching File data into documents. The documents are passed down to the Lucene.NET wrapper in order to apply the changes to the index.

To enable searches on a certain type of data, SES needs a query builder associated with that type of data. The query builder contains functionality for building queries, just as the name suggests. When a search term is passed from the overlying interface layer, the query builder constructs a suitable query based on the search term and any known data specific characteristics. The search term is parsed according to a predefined syntax that can be found in appendix IV. The query is then executed using the underlying core and the results (search hits) are converted from documents to entities of the data model before they are returned. In the Teaching File case, we created a Teaching File query builder which converts search terms into queries suitable for searches related to Teaching File data. Searches are done using the Lucene.NET wrapper of the core layer and all document results are converted to Teaching File data.

4.2.4 The interface layer

The interface layer, seen at the top of Figure 6. The architecture of SES, is the main entry point to our service. It is a set of operations and tools available for use in other software. The interface layer is responsible for client communication. Listening for requests and forwarding them to appropriate components of underlying layers.

To fulfill the demand of the system parts being interchangeable and easy to expand we chose to expose our service as an Application Programming Interface (API). It could be distributed in a number of ways including a software library, a web service etc.
The interface layer exposes the following set of operations:

- **Insert or update data**
  Inserts or updates the specified data (or data batch) to the index and returns the result of the operation.

- **Delete data**
  Removes the specified data (or data batch) from to the index and returns the result of the operation.

- **Optimize index**
  Asynchronous operation which optimizes the index to increase performance. Triggers client callback when ready.

- **Create initial index**
  Create the initial index asynchronously. Triggers client callback when completed.

- **Synchronize index**
  Checks for changes (inserts, updates or deletes) in the data sources that should be synchronized with our index and if so, applies such changes.

- **Search for term**
  Searches for the specified term in the index and returns the hits.

- **Autocomplete term**
  Searches for the specified term in the autocomplete index and returns a list of suggestions.

- **Keep alive connection**
  Dummy call to keep connection sessions alive.

We chose to distribute the API through a web service. We did this by implementing a Windows Communication Foundation (WCF) service contract exposing the earlier mentioned set of operations and the data models they consume. The web service is hosted using Internet Information Services (IIS). We used a WsDualHttpBinding, sending SOAP messages over HTTP. We used the concurrency mode “single” and the instance context mode “per session”, which means that every request will spawn a new thread within the IIS worker process. The threads are independent of each other to ensure that the service implementation is thread-safe. In this way, the threads do not share any variables or resources with each other than the index.

We are hosting our service on a cluster of servers, the earlier mentioned application servers, with hardware load-balancing built in, see Figure 7. SES integration points in existing architecture.
4.2.5 The integration layer

The integration of SES into existing architecture was briefly described in Figure 7. SES integration points in existing architecture. However, some additional components were required, that are not really visible in any of the architecture figures above. This forms an extra layer of “glue” which contains all the components needed at the different integration points of SES, including:

- **Data sources modifications** to track data changes and to handle cooperation between multiple SES instances.
- **SES clients** communicating with the SES API, located in SHS.

A more descriptive image of the integration layer and its components is available in Figure 8. The integration layer of SES.

![Figure 8. The integration layer of SES](Image)

Source: Authors.

Windows Communication Foundation (WCF) provides a feature, which given a WCF service URL will generate a C# and .NET client class with the same methods and data types as specified in the WCF service. This made it possible to easily generate clients communicating with the interface layer of the service. To keep track of the SES client instances, a so called client pool was used. The basic concept is that when you need a client in order to perform an operation, you just acquire one from the pool and release it back to the pool when you are done.
A pool is available in every application server, e.g. the Sectra Healthcare Servers (SHS). The client pool is visible in the top left of Figure 8. The integration layer of SES.

To allow workstation clients of the application servers to make use of SES, additional functions in the existing application server interface were created. The functions are simply wrapping the functionality of underlying SES clients, which now were easily acquired from a pool. Thereby, all thin workstation clients communicating with the SHS are also able to perform SES searches. To sum up, when a workstation client search request reaches the SHS, the request is forwarded to an acquired SES client who communicates the request to the actual SES service.

A startup task was created for the SHS, which acquires a SES client and requests SES to perform initial indexing. SES will negotiate with other co-existing SES services to determine which one to be responsible for the initial indexing. This negotiation is done through a lock mechanism, preferably implemented through a database table and underlying database transaction functionality. If a specific SES service turns out to be responsible for the initial indexing, it will start to extract the necessary information from the relevant data sources into the index. For this to work, SES needs to know where the data sources are located and which information that is relevant. This is, as mentioned earlier in section 4.2.3, specified through a specific data type and its specific data source crawler. Noteworthy, crawlers may need some additional functionality depending on the type of data it is associated with. Such functionality could include data mappings to extract the data from the associated database tables or methods to extract text from PDF documents.

Another recurring task was created in SHS in order to keep the searchable information up to date. The task triggers a SES client synchronize request at a predefined interval. When SES receives such a request, it fetches information changes in the data sources and updates the index according to those changes. To keep track of changes, a separate database table in the data source must be populated. This is preferably done through database triggers. The tracking triggers detect changes in the actual database tables and write information about the changes to the tracking tables.

For Teaching Files, the negotiation and tracking functionality was implemented within the main data source, which was the WISE database. The SHS performs a synchronization request to SES every second. This makes SES and its Teaching File data source crawler fetch changes from a Teaching Files tracking database table within WISE. This table is populated by some Teaching File tracking triggers, upon changes within Teaching Files related database tables. In order to be able to communicate with the database of WISE, we used a library called NHibernate with an extension called Fluent NHibernate.

Because of the fact that it can be time consuming, depending of the size of the index, to build the initial index at the first time setup that task is done asynchronously in its own thread. We want our service to be able to answer to incoming calls before the indexing is done, otherwise the clients will believe that the service is down.
4.3 Deployment

Deployment is expensive and keeping downtime as short as possible is crucial for Sectra customers. SES should not be deployed through a stand-alone install package, since it infers an additional manual install step during deployment. Although SES is a stand-alone process after deployment, it could be included in install packages for other Sectra products. As mentioned earlier, building this package is not in the scope of this thesis. However, we felt it was important to keep deployment in mind when designing the system. A number of steps required for deployment were identified:

Manual steps:

- **Installing the SES binaries**
  Move SES application files to the application server. Should be implemented as a step of the current SHS installer.

- **Setup IIS to serve SES**
  IIS has to be configured to serve the SES application. Manual task for setting up SHS similarly already exists, just extending this task to set up SES as well.

- **Data source configuration**
  Create triggers and tables for tracking and storing changes of relevant data sources, e.g. the WISE database. Should be done through SQL scripts run by the existing WISE installer.

- **Configure SES**
  Although SES comes with a default configuration, it may require some on-site tuning.

Automatic steps:

- **Initial indexing**
  Create initial index asynchronously. Started upon SHS startup if needed, e.g. the index does not exist.

- **Synchronize index**
  Synchronization is needed on a regular basis to ensure index is synced with relevant data sources. Controlled by SES clients in SHS.
4.4 Design decisions

Most design choices were made in cooperation with Sectra architects or supervisors to make sure the search platform would match Sectra’s interests as good as possible.

It was decided that workstation clients always should go through the application servers (e.g. SHS) when accessing SES. This enables usage of existing security mechanisms for authentication and authorization. Going through the application servers also felt natural since it is the only other server a workstation client currently needs know about. We did not want to introduce a lot of changes or dependabilities in such clients. Encapsulating the features inside SHS was a decision taken in cooperation with Sectra employees.

We chose to implement the enterprise search API as a web service. In this way, it is easy to access and extend for other programmers at Sectra. Also, by encapsulating the underlying functionality of the business logic and core layers, the clients using the service does not need to know anything about how the search engine or its internal parts work. This makes it easy support a number of different clients, since it is possible for every client to build a customized user interface on top of our service. Currently as mentioned before all workstation clients access the web service through the application servers. However, with a web service one could in the future connect clients directly to the service, without using the application servers if desirable.

Being a separate process, or web service, also enables the SES to run on a completely separate machine, which could be dedicated to only SES.

We have made it possible to configure a number of things to increase the flexibility of the service. This is done through editing a configuration file; this file is being used by the service. Some examples of configurable options are: Highlighting (on/off), Store path of indexes, fields enabled for autocompleting, Logging (on/off), and number of hits to return. Some options are configurable due to the fact that they might be performance expensive, for example highlighting and returning a lot of hits.

4.5 Performance

Here we will talk about the tests conducted and the performance of Sectra Enterprise Search (SES).

4.5.1 Test setup

In order to evaluate the performance of SES, a number of tests in different settings were conducted. It is interesting to see how the search service behaves during different types of searches and when the index size increases.
4.5.1.1 Systems
We used two different test systems, called A and B, to evaluate the performance of the search service. One system had very high performance, while the other was a more realistic setup matching actual customer sites. We used single core node running the WISE server and 2 application server nodes, each running a SHS server and a SES service.

System A
Dedicated high performance system.
- **CPU** - Intel Core i7 2600 @ 3.40 GHz x4
- **RAM** - 8.00 GB Dual-Channel DDR3 @ 665 MHz
- **Disk** - INTEL SSDSA2BW300G3H (SSD)

System B
Realistic system setting, virtual machine running on a dedicated server machine given the following performance;
- **CPU** - Intel Xeon @ 2.50 GHz x2
- **RAM** - 6.00 GB DDR2 @ 667 MHz
- **Disk** - SATA Raid 15000 rpm

4.5.1.2 Tests
We wrote some additional code, within the workstation client IDS7. When the code is executed, it performs a number of test searches sequentially, against a SHS server communicating with SES, and measures the response times to calculate the overall mean search time.

To see if the system was sensitive to certain types of search terms, and to be able to compare with the existing relational database searches, we performed three different types of searches:

- **Exact** - Searching for a specific term within a specific field. Tend to be faster since the search scope is limited.
- **Simple** - More simple searches, searching for a specific term in any field.
- **Join** - Searching for a set of terms in any field. Would force expensive joins when searching in corresponding relational database storage.

Search terms were randomized from a predefined set of terms and combined according to the type of search to perform.

Generally we wanted to perform the most of our tests on the largest index available. However, in order to investigate how the search service would behave when the index is getting larger, we wanted to perform our tests on a number of different index sizes. That is, populating indexes with different numbers of documents.

The data generating the stored documents were real records from a database of an actual Sectra customer, although the patient data were anonymized. Each document contained 25 stored fields, where 19 fields were searchable.
We used 4 different indexes, populated with different numbers of documents:

- **Index 1** - 5.6 million documents
- **Index 2** - 4.2 million documents
- **Index 3** - 2.8 million documents
- **Index 4** - 1.4 million documents

### 4.5.2 Memory usage

Regardless of system, the process handling the searches allocated between 0.4 and 1 GB of memory, when under heavy search load. The memory usage never reaches the cap, indicating that the amount of memory available is sufficient, in both systems. However, the rate at which data is read into the memory affects the overall performance.

### 4.5.3 Disk usage

The disk usage seems to be very close to linear, as we can see in Figure 9. Disk usage of a Lucene index. A smaller index, with about 1.4 million documents, uses approximately 550 MB. The size is of course depending on the content of each document and increasing the number of stored fields for each document, would of course increase the index size. What we have shown however, is that the size would be linear to the number of documents and thereby easy to predict.

![Figure 9. Disk usage of a Lucene index](image)

Source: Authors.
4.5.4 Processor usage

We found it important to check how much search operations will stress the Central Processing Unit (CPU) of the system, since a lot of CPU load may affect other co-existing Sectra software.

System A

When System A is idle, the CPU usage is stable and below 10%. Although, during the search tests the CPU usage went up to be rather stable within the 25-35 % range. Eventually, with some peaks above 40%, but generally a very stable, coherent CPU usage.

System B

When System B is idle, the CPU usage was often below 10% usage. However, with some shorter peaks at just above 30 %, occurring every 15 seconds or so. This is probably due to some actions performed by co-existing software, e.g. the SHS. When exposing System B to the heavier search load, the CPU usage went up and down a lot, a behavior we did not notice for System A. It may be due to the CPU having to wait for something else, like reading from disk or accessing memory. It is however hard to determine. The CPU usage during load, peaked at 60-70 % usage, but occasionally dropped down to only 15-20 % usage.

4.5.5 Response time

As mentioned earlier, we used three different kinds of searches during the tests. We performed 500 searches of each type of query, each index size and for each system.

The mean response time for each kind of search is shown in Figure 10 (System A) and Figure 11 (System B).
Figure 10. Search response times for System A

Search mean time (System A)

Source: Authors.

Figure 11. Search response times for System B

Search mean time (System B)

Source: Authors.
4.5.6 Highlighting performance

While developing SES we noticed that highlighting seemed to be expensive. In order to measure how expensive highlighting was, we ran all the tests again but this time we enabled highlighting. For the tests, an index with 5.6 million documents was used.

As we can see in figures below (Figure 12 and 13), highlighting is much more expensive for System B. The mean cost of highlighting in System A is about 35 ms and 280 ms for System B.

**Figure 12. Search mean time with and without highlighting for System A**

![Search mean time (System A)](image)

Source: Authors.

**Figure 13. Search mean time with and without highlighting for System B**

![Search mean time (System B)](image)

Source: Authors.
4.6 Frameworks & Tools

Besides the obvious usage of earlier described frameworks like Lucene.NET, IIS and WCF we also used a lot of additional frameworks, tools and utilities.

- **Visual Studio 2010** - Integrated Development Environment (IDE) used in order to write C# code for the .NET framework (http://www.microsoft.com/visualstudio).
- **Mercurial (hg)** - Distributed source control management tool used for code version control (http://mercurial.selenic.com).
- **Kanbanflow** - Web application for the daily agile workflow management, keeping track of tasks (http://www.kanbanflow.com).
- **NHibernate** - Object-Relational Mapping (ORM) solution for the Microsoft .NET Framework. It provides a framework for mapping an object-oriented domain model to a traditional relational database (http://nhforge.org).
- **FluentNHibernate** - Fluent, XML-less, compile safe, automated, convention-based mappings for NHibernate (http://www.fluentnhibernate.org).
- **Log4Net** - A class library for log output to a variety of output targets. A port of log4j for the Microsoft .NET platform (http://logging.apache.org/log4net).
5. Discussion

In this chapter, we will discuss subjects concerning the result along with some other things encountered during the development of the search platform. We will also talk briefly about how to continue the development of Sectra Enterprise Search.

5.1 Performance

The initial studying of Lucene, made us think we would get better performance, e.g. shorter search response times, than we actually ended up with. Articles discussing the performance of various Lucene implementations provides response times that are a lot lower than ours, but none seems to mention the amount of data or fields in the documents, only the amount of documents. The documents in our index contain a lot of data distributed over 19 different fields, which could be one of the reasons to why we are experiencing longer response times. Some possible reasons for System B giving a much higher response time is that the access time for the drive storing the index is much longer than for the fast SSD disk belonging to system A. System B is also a virtual machine with virtual CPU’s meaning that the incoming work has to be scheduled to the physical underlying hardware.

When running our service on System A, both autocompletion and searching feels really responsive. This was not the case for System B, the response time varied a lot depending on the complexity of the query. However, when comparing the achieved search response times with existing search functionality and discussing the results with Sectra employees; the overall performance of SES seems acceptable.

Through the performance tests we conclude that SES would gain a lot of performance by running on a dedicated machine, with high performance hardware. Another great benefit from running SES on its own server is that it would not affect other Sectra services or software by consuming a lot of resources and CPU time.
5.2 Future work

As stated earlier, SES should be considered a proof of concept. SES works and was successfully integrated into the existing Sectra products. However there is still room for refinement and more features.

While it is difficult to foresee what kind of features that might be important to end users, we have assembled a list with features which we find interesting and valuable;

- **Making more attributes searchable**
  In cooperation with end users, figure out which attributes that are desirable to search for.

- **Faceted search**
  Present hit results divided into different categories, this enables users to use the same search function independently of what kind of information he or she is looking for, e.g. a user that searches for examinations or patients, the result is then a list of the patients and a list of examinations matching the search query.

- **Synonyms**
  Synonyms can be used to represent a set of words with the same meaning, in order to eliminate problems that might occur if users use different words for the same thing, e.g. head and skull.

- **Spell checking**
  Functionality for suggesting other search terms based on the recent search term. This would help users to correct spelling mistakes or to find words with similar spelling.

- **Results pagination**
  A lot of performance can be gained through not returning all the hits for a query. There should be some sort of results pagination functionality which only fetches a limited amount of hits per page. As it is today the service returns X number of hits, where X can be chosen in the configuration, but there is no functionality for loading the next number of hits. However, a user is very seldom interested in viewing all hits, it is often only the first few hits that are interesting.

- **Full multi-language support**
  SES has basic support for any language as it is, but only fully supports English. It is possible to use letters from other languages, but the search terms and the indexed fields are being analyzed as if they were written in English. This may result in unwanted behavior, since analyzers tend to remove stop words which likely differ between languages.

- **Extend with more data**
  SES currently only handles data related to the Teaching File product area. However, SES is already designed to support multiple data types. It would be rather easy to extend SES to enable searching in other areas, like patient data or report texts.
5.2.1 User interface

The largest and most important step needed, in order to make SES a shippable product, is to build the GUI. The visualization of the search features and the presentation of search results is probably going to determine how much the end-users will appreciate SES. We think a lot can be gained through developing the GUI iteratively in collaboration with the end users.

5.2.2 Testing

In order to determine how SES will perform in a real system environment, more thorough testing should be done by a tester that is more familiar with Sectra products, who knows exactly what to observe. It would also be great to perform tests on SES while the other co-existing Sectra software is under heavy load. This would give a better understanding of how SES and other Sectra software affect each other. The difference in response time between system A and B, as we mentioned earlier was quite big. By finding and eliminating bottlenecks, a lot of performance might be gained.

5.2.3 Choosing the right framework

If Sectra decides to make a real product out of this it is important that they choose the framework that would suit their needs best. By choosing Solr or Sphinx, Sectra will save a lot of programming hours, and might win some performance. However Sectra consider programming to be cheaper than configuration, in the sense that programming is more of a one-time cost and configuration implies more support and a higher deployment cost.

By choosing Lucene.NET Sectra will gain full control of the deployment step and since the search platform may be fully integrated into existing Sectra products and be deployed together with it, the deployment cost will be a lot less than for deploying a stand-alone system.

Lucene, Sphinx and Solr, pretty much contains the same features, Solr is built upon Lucene, and Sphinx is very similar to Solr but it is written in C++ instead. It is hard to say if any of these search engines are better than the other, we consider them to be equally good in the sense of features.

One of the most challenging tasks during this master thesis was to decide and implement a solution for to populate the index. When deploying the system, an initial indexing is required in order to fill the index with the information that needs to be searchable, and also synchronize this information in run-time. This involves finding the interesting data, converting it and sending it to the search engine, and also listens for changes. By using Lucene we had to implement this ourselves, but it gave us the power to easily change or extend the functionality when needed and wanted.
6. Conclusions

It is very possible to implement an enterprise search platform prototype based on the Lucene.NET search engine library; it is also possible to successfully integrate it within existing Sectra products. Implementing the search platform themselves, instead of using existing solutions like Solr or Sphinx, seems feasible since it allows Sectra to have full control of integration and deployment. However, Sectra Enterprise Search (SES) is only a proof of concept, and it would probably require a lot of additional development to make it into a shippable product.

A number of requirements were identified during the development process, see appendix II, specified by Sectra and its customers. It was hard to find out exactly what information that should be made searchable, since it may vary from a Sectra product area to another. However, for the Teaching Files product area, the most important information to search was identified, see appendix III.

An applicable architecture for a well-integrated search platform is visible in Figure 6. The architecture of SES. The integration points are visible in Figure 7. SES integration points in existing architecture and Figure 8. The integration layer of SES.

The performance statistics are reasonable. Disk usage and search response times are very close linear to the number of documents within the index. Although, some features like highlighting may be very time consuming.

However, if some additional time were given for performance tuning and refinements of the Lucene configuration, it is most certainly possible to further reduce the response times for searches.

Even if it is possible to achieve rather short search response times, the real gain is the flexibility of the search platform. It is easy to search large data sets through combining constraints on a number of different attributes in a convenient way. Searching with SES, using the syntax described in appendix IV, feels more modern and intuitive than the old SQL-like style.
Bibliography


I. Dictionary

This appendix briefly explains the most commonly used words and abbreviations.

- **API** - Application Programming Interface.
- **SHS** - Sectra Healthcare Server, the Sectra application server software serving end-users at workstations.
- **SES** - Sectra Enterprise Search, an enterprise search service software product which is the main result of this thesis.
- **WISE** - Workflow Information Service Engine, the Sectra server software and associated relational SQL database, responsible for handling patient data.
- **WCF** - Windows Communication Foundation, framework for service applications in .NET.
- **IIS** - Internet Information Services.
- **SAN** - Storage Area Network.
- **Lucene** - Open source search engine library written in Java.
- **Lucene.NET** - C# port of the Java version for the Microsoft .NET Framework.
- **Teaching Files** - Sectra product area enabling customers to tag interesting cases with keywords for reference, education or research.
II. Identified requirements

The search platform should be...
- ... highly available.
- ... highly scalable.
- ... maintainable (easy to extend and upgrade).
- ... able to search in different sets of data.
- ... well integrated with current architecture.
- ... deployed within the existing architecture.

The search platform should include functionality for...
- ... autocompletion.
- ... highlighting hits in result set.
- ... full text search.
- ... building an initial index automatically.
- ... synchronizing searchable data automatically.
- ... configurable boosting for different fields.

The produced code should be...
- ... well documented.
- ... written in C# for the Microsoft .NET Framework.
- ... written according to Sectra guidelines.
III. Attributes of Teaching Files

This appendix lists the attributes of the Teaching Files product area, produced together with Sectra employees and customers. Only a subset of the attributes was selected for inclusion in the proof of concept implementation (marked in bold).

- Examination
  - **Identifier (ID)**
  - Accession number
  - Date and time
  - Body part
  - **Modality** - What kind of modality was used, e.g. CT, MR.
  - **Station name** - The identifier of the specific modality used.
  - **Institution** - The medical clinic where the examination were performed.
  - **Performing physician**
  - Reading physician
  - **Description** - Additional information about the examination.

- Request
  - **Referring physician** - The physician requesting the examination.
  - Address
  - Question
  - Anamnesis
  - Unit - The medical clinic requesting the examination.

- Patient
  - **Medical record number** - Patient identifier like SSN or Swedish PNR.
  - **Name**
  - **Date of birth**
  - Age at time of examination
  - **Sex**

- Teaching File
  - **Keyword** - The keyword the examination was tagged with.
  - **Comments** - Additional comments about the keyword tagging.
  - **Author** - The user who tagged.
  - Creator - The user creating the keyword.
  - **Date** - The date when tagged.

- Report
  - **Text**
  - Date
  - Signature
  - Doctor
IV. Lucene query parser syntax

This appendix describes the syntax available for parsing by the Lucene query parser, based on Apache (2012e).

Fields
It is possible to search in a specific field. If users know which field they want to search in, they may write the field name and the search term separated by a colon, e.g. Title:Sectra.

Wildcard
Wildcard is useful when the user is uncertain of spelling or the ending of a word, e.g. Sectr*, Se*ra. The asterisk symbol (*) will be exchanged for 0 or more characters. It is also possible to specify that you are just uncertain about one character, e.g. Se?tra.

Fuzzy
By adding a tilde symbol (~) at the end of a search term tells the query parser that you are unsure of the spelling of this word, e.g. Sektra~. It is also possible to tell queryparser of how wide this search should be by specifying a number after the tilde symbol. Specifying 1 means perfect spelling, which will only match exact terms, and the closer to 0 the wider the search.

Proximity
It is often desirable to say that you want the words to be near each other in the documents you are looking for. This is done with the proximity operator, e.g. “Sectra Company”~7. This means that the words “Sectra” and “Company” should be present in the document and they should be within 7 words from each other.

Range
Range queries can be used to find documents that are between a lower and upper bound, e.g. date:[20120910 TO 20120918]. This will find all documents which has a Date between the two values. This works for both numerical and alphabetical ranges. Range queries are inclusive if specified with [ ] and exclusive if specified with { }, e.g. title:{Company TO Sectra}. This will match all titles alphabetically between “Company” and “Sectra”, including “Company” but not “Sectra”.

Boosting
Boosting makes it possible to give weights to some search terms, weights can make a search term either more important or less important, e.g. Sectra^4 Company. Will search for documents containing Sectra and/or company, but the weight makes the search term “Sectra” more important than “Company”. The default value is 1.
**Boolean**

Boolean expressions are probably the most useful feature of queryparser. Available operators are: AND, OR, NOT. A search for “Sectra NOT Company” will find documents containing “Sectra” and exclude those containing “Company”, while “Sectra AND Company” will find documents containing both “Company” and “Sectra”. Of course “Sectra OR Company” will find documents containing “Company”, “Sectra” or both of them.

**Grouping**

The query parser supports grouping, which means that you can combine boolean operators to perform advanced queries including subqueries. Searching for “((Sectra OR Company) AND Product) NOT Software”, will return documents containing “Product” and “Sectra” and/or “Company” but not “Software”.
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – från publiceringsdatum under förutsättning att inga extraordinära 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 icke-kommersiell 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 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/

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon 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/.

© Fredrik Pettersson & Niklas Pettersson