Near Real Time Search in Large Amount of Data

Robin Rönnberg

June 17, 2013
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Jan-Erik Moström
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

This paper consists of a project assigned by Tieto in Umeå where the project goal is to ease the process of matching similar trouble reports together. This is done by creating a support system to the errand system that Tieto uses for handling trouble reports. The support system helps developer to find similar errand by being able to free text search in errands of interest. The paper also includes a comparison of relational database and NoSQL databases regarding free-text search abilities. The comparison act as a foundation for the creation of the support system.

Contents

1 Introduction 1

2 Problem Description 3

2.1 Problem Statement . . . 3

2.1.1 Retrieving information from MHWeb . . . 5

2.1.2 Storing Errands . . . 6

2.1.3 Searching For Errands . . . 6

2.1.4 Displaying Search Results . . . 6

2.1.5 Finding Similar Errands . . . 6

2.2 Goals . . . 7

3 Comparison of NoSQL and Relational Databases for Free Text Search 9

3.1 Information Retrieval . . . 9

3.1.1 Differences Between IR and Databases . . . 11

3.1.2 IR Models . . . 13

3.1.3 Queries in IR Systems . . . 14

3.1.4 Indexing . . . 15

3.1.5 Measuring Performance of Information Retrieval System . . . 17

3.2 Free Text Searching . . . 19

3.2.1 Requirements . . . 20

3.2.2 Databases . . . 21

3.2.3 Comparison . . . 25

3.3 Database for LTE TR Tool . . . 29

4 Accomplishment 31

4.1 Preliminaries . . . 31

4.2 How the work was done . . . 32

5 Results 33

5.1 System Overview . . . 33

5.2 Database . . . 34

5.2.1 Implementation of Writer . . . 34

5.3 Web Service 38
5.3.1 Web Service Interface 39
5.3.2 JSON 41
5.3.3 Session Handling 42
5.4 Website 42
5.4.1 Structure 43
5.4.2 Design 43
5.4.3 Query Syntax 44
5.4.4 Features 47
5.5 Hosting LTE TR Tool 47

6 Conclusions 49
6.1 Goals 49
6.2 Limitations 49
6.3 Future work 49

7 Acknowledgements 53

References 55

Chapter 1

Introduction

The goal of this thesis is to try and help software developers at Tieto in Umeå to work more efficient. Almost all software developing company has some sort of errand system to keep track of new features and faults or errors in their systems. An errand system is filled with tickets. One ticket describes a new feature for some particular product or some sort of short coming of an existing product such as a bug or error of some kind. In large and complex software one bug or error can cause many different symptoms leading testers or users writing many tickets describing their version of the problem. The bug or error may cause some ripple effect, meaning that a bug in one part of a system may lead to strange behavior in another part of the system. This means that the errand system will be filled with more than one ticket, one slightly different from the next, for just one problem. The problem with this situation is that it could lead to two or more developers working and solving the same problem or that a developer tries to solve a problem that has already been solved, which is undesirable. So the first thing a developer needs to find out when starting to resolve a ticket is to find out if there are other tickets related to the same error.

At Tieto they have a huge database filled with tickets or TRs (Trouble Report) which is hard to search in because of its size and structure. The developers spend lots of time searching in this database, trying to find TRs related to one another. One of the problems is that the database contains TRs for all projects and products which imply that when searching for TRs within one project or product you search among many other projects that may not be related to the one you are interested in. Another problem is that the database is not optimized for free text search, which is often what developers want to do, this adds even more time when performing a query against the database.

What they need is an application that is able to perform a full text search against a subset of the large database. This subset should be configurable to contain a desired project and projects that are related to the desired project. This application will ease the process of finding errands that originates from the same issue. So in essences what the application will do is to help with matching TRs to each other. Preferably the application would match errands automatically to one another, so when a developer is looking at a specific errand the application could suggest other errands that seam to be related to the same problem.

One of the key aspects of this application is speed. Developers want a tool that can deliver search hits with a real time experience, can be used by multiple users at the same time without loosing performance and that is easy to access.

Chapter 2

Problem Description

In this chapter you will find a detailed problem description of all parts of the project. This includes description of the existing errand system, requirements on the new software and goals.

2.1 Problem Statement

Tieto in Umeå has an errand system today that handles all the tickets for all projects and software. The errand system is actually owned by Ericsson. The developers at Tieto in Umeå almost exclusively works on projects from Ericsson an there for use this system to a great extent. From this point forward, this system will be refered to as Tieto's errand system, since the project is primarily aimed to the developers at Tieto in Umeå. The system can be accessed by developers through a website called MHWeb where they can read the description of an errand. The website also supports searching among errand so that developers can find errands with desired properties. The errands are stored in a relation database in a clever fashion so each errand belongs to a specific project, can be assign to a specific developer, has record of what has been done and needs to be done etc. Since the software system that Tieto handles are huge and several hundred developers can work in one project there are a hierarchy of projects and subprojects. The errand system keeps track of what project an errand belongs to and which subprojects the errand is assigned to. Errands are often reassigned several times between theses subprojects before they end up in the right place. The errand system handles these changes while keeping track of all changes. So each errand carries a lot of information and has lots of relations that the database is responsible for.

The problems developers have when searching for errands in MHWeb is first to narrow the search down to the desired projects. When writing the search query they need to specify a lot of information to perform a search against the projects and subprojects they are actually interested in. Searching in every project in the whole database is not an option since it would take a lot of time to get a response and you would get hits from projects you are not really interested in. The search queries the developers create are similar to a SQL query. This means that if you overcome the first step of narrowing down where you want to search, the response time of the query is highly dependent on how you phrase it. If you are not careful and specify things in the right order the response time will be very long, up to several minutes. The second problem is that the database is not really optimized for free text search. Many developers create queries where they use SQL-term `LIKE` or `CONTAINS`.

For example you could add something like this to your query:
`WHERE 'heading' CONTAINS 'error 100'`. This way of searching is both tedious and slow. The third problem is that the database does not order the search result in any particular order. When the database finds a match it simply adds it to the response which means that when using `OR`-statement a hit that matches many of the statements does not get a higher score and will end up any where among the hits. Preferably a hit that matches more statements would get ordered higher in the response than a hit that matches fewer statements. The fourth problem is the presentation of the result. When performing a query in the previously described fashion you simply get a list of unordered errands with a HTML-link to each errand.

This way of searching and finding errands is a reality for many developers at Tieto today. What is needed is a way of searching that is simpler, faster and presented in a better way. This is why this project is being carried out. The project will have to address all the problems mentioned above that the current system has and have extra functionality for matching errands together. The best solution to these problems is to create a smaller support system that interacts with the current system. It would not be possible to go in to the current errand system and try to address all these problems without redesigning a great deal of the system which there is simply not time for in a project like this and no desire form Tieto since the current system, apart from specialized searches, works well and has a lot more requirements to consider than just to enable fast searches. Creating this support system is what the project is all about.

The whole project can be divided into five smaller parts. The first step is to retrieve errands (TRs) from the MHWeb database, where all errands are stored, and create a subset of all the errands of the current system. This subset should be configurable to consist of errands for any criteria the users of the system desires. It could be just the errands from one subproject or many different projects. This is up to the users of the system to choose. For the support system to be useful the subset would of course need to contain errands from more than just one subproject, but this should however be configurable to match the needs of the users. The user in this context could be a division at Tieto such as all the developer located in Umeå or all developers in a large project. The idea is that you should be able to set up this support system for some part of the current errand system, meaning that different groups could set up their own version of the support system, configured to match their needs. Another important part of this step is to make sure that the support system receives any changes of errands from the current errand system. The subset needs to be fairly up to date, otherwise the information in the support system will not be very useful. Fairly up to date means that changes does not need to be reflected in the subset of errands right away, but should at least take effect within one day.

The second step after you have managed a way of getting the errands you want to search in is to store them in a good way. This part is really important since it will determine want you will be able to search for, how fast you can receive search hits and how good the search hits will be. This part is vital for the whole support system, since it will set the limitations of what the rest of the system will be able to do.

The third step after storing the errands is to make them searchable. The support system will have to have some function where you can put in a search query and get back good search results. This functionality should be accessible remotely.

The fourth step after receiving the result is to present it to the user. This also means that the user needs a way of entering the search criteria and then getting some result back. This should be easy to access and presented in good way so that the user gets relevant feedback on what they searched on.

The fifth and final step of this project is to make the support system match errands. What is wanted here is for the system to take one errand as input and try and finding other errands that seems to be related to that one.

2.1.1 Retrieving information from MHWeb

MHWeb is the name of the system that developers use to find and look at errands. Fetching desirable errands from this system is necessary to obtain a subset of all the errands. This system is really huge and handles millions of errands. Within Tieto there are many different needs of looking at and processing errands, which means that the support system being built in this project is not the only one interacting with MHWeb. This system has two external API's which support systems can use to access the data in the database. This database is called MH database.

The first API communicates via SOAP (Simple Object Access Protocol) messages. This API enables reads, writes, and updates of errands. This API is intended to be used for manipulating and reading errands one at a time. It does not provide functionality for getting lots of errand with just one request. If this is to be used in this project there will be a need of first getting a list of all the errands of interest and then requesting them one by one via this API. However it actually exist a way of executing SQL-statements directly to the database which would permit a way of getting lots of errands with only one request or getting a list of errands and then fetching the errands one by one with the SOAP API. This way of directly accessing the database is not really intended for projects like this. The two external API's is to be used primarily.

The second API uses Java Message Service (JMS) to access the data in the database. JMS is a Java message oriented middleware for passing messages between two or more clients. It is a messaging standard that allows application components based on the Java Enterprise Edition to create, send, receive, and read message. This middleware allows messages to be past between a provider and subscribers. The provider in this case is MHWeb, which provides errands to be sent to the subscribers of the system. The subscribers in this case are support systems like the one being built. The basic idea is that a subscriber creates an account with the provider and makes one or more subscriptions. A subscription in this system can be described very freely. You can make a subscription on all errands from a specific project or a subscription on all errand created on a specific day or errand created by some user etc. When a subscriber makes a subscription the provider queues all the errands that matches the subscription. When the subscriber makes contact to the provider it gets the messages in the queue. If any changes would occur to the errands that a subscriber has on its subscriptions these changes are also queued up. This API does only support reads and not writes and updates of errands. MHWeb provide a JMS-client for communicating with the system. This client handles all JMS related processing such as connecting to MHWeb, buffering errands locally, exception handling and logging. But to use the client you have to write your own Java class that handles the actual storage of errands. The client provides an interface called `Writer` that is supposed to be used for writing such a class. This interface has three methods: `remove`, `write` and `close`. The `remove` method is called when an errand is removed from the subscription, the `write` method is called when an errand is added to the subscription or updated and the `close` method is called when the client closes the connection to MHWeb.

2.1.2 Storing Errands

Once the errands are fetched from the MHWeb system it is not difficult to store them. The difficulty lies in storing them in a way that enables really fast free text searches. All of the information in an errand does not need to be stored. As mentioned before each errand carries lots of information and all of the information is not needed for this particular application. The end users of the system need to be consulted to find out exactly what information that needs to be stored. The size of each errand differs from a couple of kilo bytes to a couple of mega bytes. The number of errands that the new application should be able to handle is of magnitude of hundred thousand errands.

The storage does not have any special requirements besides enabling fast searches among lots of errands, separating the errands with an unique identifier, separating the data in each errand into common named fields and sorting search hits with the help of some scoring on each hit so that the better a errand matches the search query the higher the score. This opens up for usage of nontraditional relational databases, like NoSQL-databases which generally do not use SQL for data manipulation and is often highly scalable.

2.1.3 Searching For Errands

Searching for errands should be simple and powerful. The users want to be able to access the database from different environments and platforms remotely. The system needs a service that meets these requirements. This service should take in a search query as input and give back a list of sorted search hits. The query should be as simple as possible and not built like in MHWeb where you basically enter SQL-statements. The search query syntax should however support searches on the unique identifier each errand has and searches of text in specific fields (the fields mentioned in previous section).

2.1.4 Displaying Search Results

The users need a way of entering their search query and then displaying the result. This part of the solution should communicate with the service described in the previous step. The users want a simple input field where they can enter their search query, hit a search button and get search results. The search result should be easy to survey and preferably give direct feedback to the user by highlighting or emphasizing the text the user entered in the query.

Since this is a support system to the actual errand system a link to the original errand is wanted for each search hit.

2.1.5 Finding Similar Errands

This step can be seen as desirable but not mandatory. It does not have any specific requirements and is to be interpreted very freely. The essence of the whole project is to make it easier for developers to match errands together that originate from the same problem. What is wanted from the new system is a way of detecting errands that are similar and suggest this potential match to the user. This should be seen as the next step in helping the developers with matching errands but not replacing their work when it comes to matching errands together.

Designing this part of the system could get extremely complicated and you could easily make this part into a separate thesis project. Since there is a lot of work without this functionality this part has to be rather limited and treated with caution.

2.2 Goals

The goals of the system are summarized by the following list:

- Create a system that help developers to match similar errands through free text search.
- Make the system configurable to fit the needs of different developing teams.
- In an easy manner be able to specify which errands should be part of the local database.
- Create a mechanism that ensures that the local database is up-to-date with the real database.
- Free text search in up to 100 000 errands with real time feeling.
- Get useful feedback on the search result.
- Access of the system should be easy and platform independent.
- The design of the system should enable further development of the product.
- Make the system easy to set up.

These goals are non-functional requirements of the system and there is a need to have an open dialogue with Tieto to ensure that these requirements are interpreted correctly and meet the expected result.

Chapter 3

Comparison of NoSQL and Relational Databases for Free Text Search

This chapter includes a comparison between traditional relational databases and NoSQL databases with focus on free text searching. The chapter start out with describing how free text search or information retrieval (IR) differs from traditional relational databases in aspect of how the data is structured and how documents are retrieved from the database. Further it includes different IR models, ranking algorithms and how to evaluate IR systems. In the end of the chapter a comparison between a NoSQL and SQL database is conducted to provide a basis for choosing a good database in the thesis project.

3.1 Information Retrieval

Information Retrieval (IR) is the process of gathering desired information from a store of information or a database. IR aims at fetching information to answer a question or solve a problem. A user using an IR system often wants information that is related to their problem or can help them answer a question, rather than getting some exact fact. Such information is typically unstructured and needs to be stored in a different way than in a relational database management system (RDBMS), where data is stored in a structured fashion (this is describes in more detail under 3.1.1). An IR system differs from a RDBMS in the way the user gets information and the knowledge the user needs to have to obtain information. The need of knowing schemas or special query language is not present in an IR system. Searching in an IR system is typically done with a freely formed search request built up on keywords. If a person has a problem and wants to find information on that subject from an IR system the person will try to think of some keywords that relates to the problem or a general topic under which the problem could be found. Examples of IR systems are Google [12] and Bing [17]. But computerized online IR systems have been around for a very long time. Some of the first online dial-up service were MEDLINE (Medical Literature Analysis and Retrieval System Online) in 1971 [22], which served 25 persons with information from the database. But since the start of the internet IR systems has grown at a tremendous rate and is now something everyone can use to find information as long as they have internet connection.

When dealing with a system that does not have a strict query language the expected result of the system becomes harder to define. When creating an SQL query to a database, a record ether matches the query or it does not. This is not the case of an IR system. For example a person might want to get information about bananas. If a person enters the search keyword ”fruit” he might expect to get this information. The keyword does not explicitly have to be in the result. This makes it harder to create a good IR system because it is not as clear what should be in the result as in a RDBMS. Because of this, one search hit can be better than another which opens up for ranking the result. Ranking the result in a SQL database does not really make any sense when you choose to see the result in this manner. The result to a SQL query either matches or it does not. It would be strange to try and rank something that has this binary quality. A hit is 100 percent right or 100 percent wrong. Ranking the result of an IR system depends on the IR model that is used. There are a couple of different approaches when trying to rank search results which will be examined later.

Ranking search result in an IR system becomes really difficult when the expected result is so undefined. This has to do with the fact that the users of an IR system are humans and think differently when searching. Two persons searching for the same information may not use the same keywords at all when performing a search on an IR system. A good search result becomes subjective. When creating an IR system one has to have this in mind. In Iris Xie’s book *Interactive Information Retrieval in Digital Environments* [33] she talks about two approaches when designing an IR system, system-oriented and user-oriented. The system-oriented approach has been dominant in the past, but in recent years a more human and socio-technical approach has been favored. This has to do with the fact that users cannot be satisfied with the technically-oriented design. According to Iris book the user-centered approach criticized the system-centered approach for paying little attention to users and their behavior, simultaneously user-centered research does not deliver tangible design solutions. Designers taking the system-centered approach do not care about user studies and their results in their design of IR systems.

In a user-centered approach it is important to understand the users’ behavior and strategy when they are searching for information. A user might at first not even know what he/she is searching for when trying to answer a question or solve a problem. The picture of the information that is needed slowly takes shape as the user understands more and more of what he/she actually wants. The search strategy becomes an interactive process which the IR system should be aware of. There are many models describing user-oriented approaches, one of the most cite and well known is *Taylor’s Levels of Information Need* [25]. According to Taylor there are four levels of information need in the question negotiation process:

– Visceral need: The actual, but unexpressed, need for information.

– Conscious need: The conscious within-brain description of the need.

– Formalized need: The formal statement of the question.

– Compromised need: The question as presented to the information system.

At the visceral need level the user might have a vague information need but it is not clear enough for her/him to articulate the need. At the conscious level the user might have mental picture of the need but still cannot define it. At the formalized level the user might be able to express his/her need. At the compromised level the user might be able to express this need so that it could be interpreted by an IR system. This model describes how users go from having a vague idea of what they need, to how to get the information from an information system.

When taking in all this information it becomes clearer that there is a big gap between a traditional relation database and an IR system. An IR system can still have a RDBMS in the bottom of the system but it is far from being a good IR system when it stands alone. The question here becomes if using a RDBMS in the bottom of an IR system has more advantages than a system with a NoSQL database more design for this type of systems.

3.1.1 Differences Between IR and Databases

A relational database management system (RDBMS) typically stores structured data that is easy to apply to a well-defined formal model. An example of structured data that is suited for a RDBMS is names and addresses to people. Each person has a first name and surname and an address where they live. It is easy to put such data in a RDBMS and get information from it. The information on what address a person lives is a fact. Unstructured data however does not fit easily into such a well-defined formal model. Example of such data can be a tutorial describing how to conduct an experiment. The data may have some structure or steps but there is no general model of how to structure it and the data will most certainly contain lots of natural language which will differ a lot from tutorial to tutorial. Relational databases has defined schemas and formal language to manipulate and retrieve information with, where as an IR system has no fixed data model and store data in from of documents or some more loosely define schema. Each document does not need to have the same structure or follow a strict schema as in a relational database management system (RDBMS). However the documents need some separation model to be able to distinguish them from one another. Each document in an IR system can be seen as a combined quantity of data with a unique identifier to separate them. The data each document has can vary from document to document. It can store text, pictures, HTML, XML or whatever is suitable. The IR system can in such a model perform indexing, searching and other operation on each document to retrieve information.

A RDBMS uses a relational model to enable SQL queries and transactions. The queries are translated into relational algebra operations and search algorithms and results in a new relation (table). This produces an exact answer and there is no doubt of what should be in the result and what should not be. For an IR system this is much vaguer. There is no fixed language used when querying an IR system, instead IR systems often uses queries build up on keywords (terms). The result is not as defined as in a RDBMS, it is the systems best attempt at finding matching information on the submitted keywords. The result is some sort of list pointing to the documents that builds up the IR system. A database operates on attributes and relations and does limited operations on the actual data that an attributes holds. An IR system on the other hand does complex analysis on the data values themselves in each document to determine the relevance of each document to the users' requests (mentioned later under Text Processing).

In the book *Fundamentals of Database Systems* [10] the author points out the most significant differences between databases and IR systems, which can be seen in table 3.1.1.

Text Processing

As mentioned earlier an IR system usually does more operations on the actual data stored for each document than a RDBMS normally does. Some of the commonly used operations, or text processing techniques, are stopword removal, stemming, use of thesaurus and of course indexing the data. Indexing is not unique for IR systems, it is often used in RDBMS

A Comparison of Databases and IR Systems

Databases	**IR System**
• Structured data • Schema driven • Relational (or object, hierarchical, and network) model is predominant • Structured query model • Rich meta-data operations • Query returns data • Results are based on exact matching (always correct)	• Unstructured data • No fixed schema; various data models • Free-form query models • Rich data operations • Search request returns list or pointers to documents • Results are based on approximate matching and measures of effectiveness (may be imprecise and ranked)

Table 3.1: A comparison of databases and IR systems

as well to speed up retrieval of data. Indexing techniques will not be covered here, see 3.1.4 for more details on indexing.

Stopwords are words that are filtered out and removed because they do not increase the precision when searching. If a word exist in the text of all documents stored in an IR system, searching on that word will not help you to find specific information. Words that are expected to occur in 80 percent or more of the stored documents are typically referred to as stop words [10]. Words that do not have any meaning themselves are often obsolete and can safely be removed without affecting the search precision. Typical stopwords are *the, of, to, a, and, in, for, that, was, on, he, is, with, at, by* and *it*. There are of course situations where removing these words will have a negative impact. If a user were to search for "To be or not to be" the system may not be able to find the document the user is searching for. Which stopwords to remove needs to be considered for each IR system. The goal of stopword removal is to remove words that are obsolete and does not contribute in finding information.

Stemming is the process of reducing words to their stem, base or root form. A stem of a word is obtained by removing the suffix and prefix of an original word. A stemmer for example should identify the words "talking" and "talked" as the base form "talk". By stemming all words searching for the keyword "talking" would return documents that contain words like "talk", "talked", "talking" etc. Many search engines treat words with the same stem as synonyms as a kind of query broadening, a process called conflation [32]. Studying of stemming algorithms has been done for a very long time and is widely used in IR systems.

A thesaurus is a collection of phrases or words plus a set of relations between them. Each word has a list of synonyms and related words. In a system synonyms can then be translated into the same word. The idea is that the system should be able to group closely related words under a common word and thereby help the users to find information related to the keywords they search on without having to include synonyms in the query themselves. However the use of thesaurus have not demonstrated benefits for retrieval performance and it is difficult to construct a thesaurus automatically for large text databases [15].

Other common text processing steps are changing all characters either to upper case or lower case to get rid of case sensitive search system. Some systems remove numbers or dates to try and minimize the size of the index. In each IR system the designers of the system has to choose what data to index and what data to remove. The index should be as compact as possible to enable fast accurate searches without compromising the effectiveness of the system.

3.1.2 IR Models

Retrieval models can be seen as blueprints for creating an IR system. They provide a high abstraction level for designers, developers and researchers of IR systems which make it easier to discuss and implement retrieval systems. In this section we will look in to three different retrieval models, vector space model, boolean model and probabilistic model.

Vector Space Model

The vector space model provides a model which makes term weighting and ranking possible. In the model, queries and documents are seen as vectors in an n-dimensional space where n are the number of terms in the document collection. In the retrieval process documents are then ranked by the "distance" from the query. The distance between a query and a document can be computed in a number of ways. The model does not provided a function for this operation, but a commonly used one is the cosine of the angle between the query-vector and the document-vector. As the angel between to vectors decreases the cosine of the angle approaches one, the closer the value is one, the higher the ranking. A problem with the vector space model is that it does not define what the values of the vector components should be. The process of assigning values to the vector components is known as term weighting. This is not an easy task to do. From the book *Information Retrieval: Searching in the 21st Century* [11]:

> "*Early experiments by Salton (1971) and Yang (1973) showed that term weighting is not a trivial problem at all. They suggested so-called tf:idf weights, a combination of term frequency tf, which is the number of occurrences of a term in a document, and idf, the inverse document frequency, which is a value inversely related to the document frequency df, which is the number of documents that contain the term.*"

Boolean Model

The boolean retrieval model is the simplest of the models. It is an exact matching model that matches terms (keywords) in the search query to the documents. A document either matches the query or it does not. For example, a query with the term "water" will get all documents where the term exists within the documents text. The model does not provide any ranking since it either matches or it does not. One can choose to see the ranking value of a document as a binary value, it matches or it does not. The boolean model normally provides the standard boolean operators `AND`, `OR` and `NOT`, which can be used together with the search terms. For example, the query "food AND water" will retrieve all documents containing both terms, the query "food OR water" will retrieve all documents containing at least one of the terms and the query "food NOT water" will retrieve all documents containing the term "food" if the term "water" is absent.

The advantage of the boolean model is that it gives expert users a sense of control [11]. The model is very straight forward but does not enable any tweaking of the system to adapt the system to the content of the documents. The main disadvantage of this model is that it does not provide any ranking of retrieved documents.

Probabilistic Model

In the probabilistic model documents are ranked by the estimated probability of relevance with respect to the query and the document. The model assumes that each query has a set of relevant documents and a set of non relevant document. The task is to calculate the probability of the document being in the relevant set and compare that to the probability of the document being in the non relevant.

Let the representation of a document be denoted as D and let R be the relevance and NR be the non-relevance of that document. Calculating the probability for a document D being in the relevant set is $P(R|D)$ and calculating the probability for the document being in the non-relevant set is $P(NR|D)$. These probabilities can be calculated using Bayes' theorem [28]:

$$P(R|D) = P(D|R) \times P(R)/P(D)$$
$$P(NR|D) = P(D|NR) \times P(NR)/P(D)$$

A document is defined as relevant if $P(R|D) > P(NR|D)$ which is equivalent with $P(D|R) \times P(R)/P(D) > P(D|NR) \times P(NR)/P(D)$. The likelihood ratio $P(D|R)/P(D|NR)$ is used as a score to determine the likelihood of the document with representation D belonging to the relevant set [10].

3.1.3 Queries in IR Systems

Many IR systems supports more than just the use of keywords in the search query to make the query more powerful and expressive. Within the information retrieval domain there are a couple conventional query types, these are listed below.

Boolean Queries

Boolean queries support the use of boolean operators on the search terms (keywords). The operators are `AND`, `OR` and `NOT`. The actual syntax may differ from system to system but a commonly used syntax is to use the operator name in upper case. For example, a query could look like this: ”water AND food NOT meat”. This means all documents that has both the terms ”water” and ”food”, but does not contain the term ”meat”.

Phrase Queries

Normally when using multiple terms in a query the order of the terms is lost when performing the search. The search is done on each individual term. The support of phrase queries enables the users to preserve the order of the terms so that it becomes possible to search for a whole phrase. This is normally done by putting the terms inside quotes. When searching for a whole phrase the document returned must contain the complete phrase with the terms in the same order as in the query.

Wildcard Queries

Support of wildcard search means that the users can use a wildcard, usually denoted with ”*”, to express a sequence of unknown characters. For example, the query ”app*” would return documents containing ”apples”, ”apple”, ”application” and so on. Some systems support the use of wildcard limited to just one character, usually denoted by ”?”. The query ”da?” could return documents containing ”dad”, ”day”, ”dam” and so on.

Proximity Queries

A proximity query refers to queries with multiple terms where the distance between the terms can be specified. The distance here is simple how many words there is between two words in the document. A phrase query can be seen as a proximity query with the distance set to zero. For example, if the distance is set to one and the query contains the two terms "dogs" and "cats" the resulting document must contain the two terms with max one other term in between them. A document containing the phrase "dogs hate cats" could be returned in this case.

3.1.4 Indexing

Searching for terms in documents is the basic thing almost all IR systems support. The users want to be able to search through the text in all documents in search for information that can help them answer a question or solve a problem. A simple and straight forward way would be to sequentially match each word in all documents with the user's query. This would however not be very fast, and as the number of documents increase the search time would increase linearly to the number of documents and the length of the documents. To speed up the matching process most IR systems creates indexes and operate on an inverted index structure to match terms. The inverted index data structure is normally created when a new document is inserted into the IR system. The system scans through the text and builds up the inverted index which later can be used to retrieve information much faster. In addition to an inverted index, statistical information is also collected and stored in lookup tables. This statistical information generally includes count of terms in each document, the term position within the document and the length of the document. This statistic can then be used when terms are weighted in the ranking process. The system might rank a document higher the more times the search term occur within the document. The statistical gathering and inverted index is built up after the documents are preprocessed. Preprocessing might include stopword removal, stemming and other steps as mentioned under 3.1.1.

The way an inverted index is built up is by saving a reference to each document for each term in the document. The index contains of all terms in all documents, except for stopwords, were each term has a list of references to the documents containing the term. The index often contains a reference to where in the document the term occurs. Consider the three following documents:

Doc ID	**Text**
1	The dog is black and the dog is lonely.
2	The cat is black and white.
3	The dog and the cat are friends.

An inverted index with both reference to the document and the position in the document would look like this:

ID	**Term**	**Doc id:position**
1	black	1:4, 2:4
2	cat	2:2, 3:5
3	dog	1:2, 1:7, 3:2
4	friends	3:7
5	lonely	1:9
6	white	2:6

By using an inverted index the time of information retrieval in a system is sped up by a great deal. The system might use additional data structures, like B-tree or hashing to optimize the search process even further. A hash function can for example be used to find the search-term in the inverted index data structure, instead of searching in the inverted index sequentially.

The index technique described above is how indices are created for full text search. As mentioned before, indexing is not something only used for full text searches. Indices are widely used in RDBMS to speed up retrieval of data. The technique for indexing other data types is however not the same as for full text indices.

When people talk about an index without specifying the type of the index, they usually refer to a B-tree index [27]. The basic idea of a B-tree is that all values are stored in order. Each node in a B-tree holds copies of a number of keys, the keys act as separation values which divides its sub trees. The leaf nodes hold pointers to the actual value. The distance from a leaf node to the root node is the same. Each leaf holds a pointer to the next leaf node to speed up sequential access. Figure 3.1 shows a B-tree with the first fifteen prime numbers.

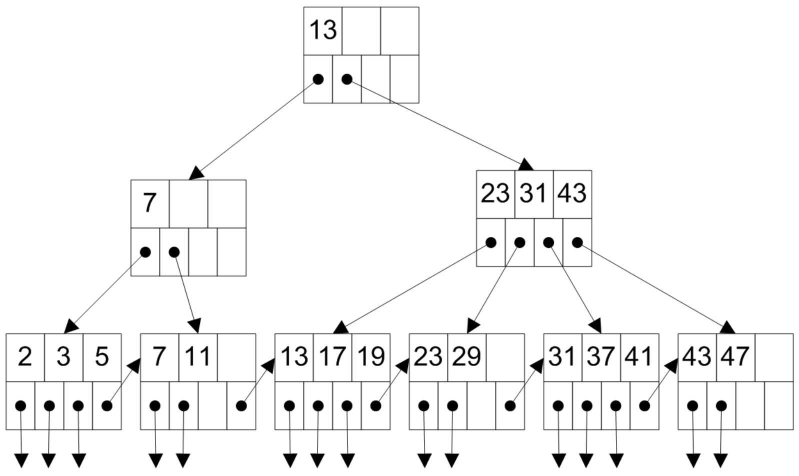


Figure 3.1: A B-tree index structure of the first fifteen prime numbers

The pointers at the leaf nodes points to the actual record (row in a table) that holds the key. Each node in figure 3.1 has the ability to store three keys, lets name these *k1, k2* and *k3* from left to right, and four pointers, lets name these *p1, p2, p3* and *p4* from left to right. Starting at the root node, when looking for a value with key x the key is compared to each key in the node and follows one of the pointers that lead closer to the right key in a leaf node. The comparison is as follows:

- Follow *p1* if $x <$ *k1*
- Follow *p2* if $x >=$ *k1* and $x <$ *k2*
- Follow *p3* if $x >=$ *k2* and $x <$ *k3*
- Follow *p4* if $x >=$ *k3*

When reaching a leaf node, p_i points to the value of k_i.

Another type of index is a hash index which uses a hash function to convert a key into an address. The Address points to a location in a lookup table which points to the row that holds the wanted value. A hash index in highly dependent on the hash functions ability to produce a unique hash code. If keys result in the same hash code there will be a collision which has to be taken care of. Usually this result in a sequential comparison of the keys that points to the same hash code. Suppose the hash function produces the same hash code for all keys, this will result in a sequential search of all keys and the index does not speed up the lookup at all. There are other ways of dealing with collisions but the main problem still remains if the hash function is insufficient. Another short coming of a hash index is that the data does not need to be ordered which means that the index will not speed up sorting of values. B-trees have a big advantage over a hash index when it comes to sorting the result since the data structure in a B-tree order the keys.

3.1.5 Measuring Performance of Information Retrieval System

To be able to evaluate an IR system's performance and compare different systems with each other there is a need of a common measurement method. One of the most common methods applied on IR systems is the measurement of *recall* and *precision.* One problem with measuring performance of IR systems is the gap between humans ability to express their information need in terms of a query and the system's ability to understand that query and fetch relevant information. In IR technology one talks about the *topical relevance* and the *user relevance.* The *topical relevance* refers the system's ability to match the topic of the query to the topic of the documents. The *user relevance* refers to the system's ability to match the users' informational need with the documents. Mapping one's informational need to a query is a cognitive task. User relevance is there for much harder to measure and includes other implicit factors, such as perception, timeliness, context and the users' knowledge of the system. The users' process of having an informational need and expressing it in to a query can be described by *Taylor's Levels of Information Need* described in the beginning of 3.1.

There are many different algorithms for measuring and ranking IR systems. In this section we will only look at the most used one, *precision* and *recall.* IR is a big research area and there are lots of people working on how to measure and benchmark IR systems so that they can be compared to each other. One big event where this is done is the *Text Retrieval Conference* (TREC) [21] co-sponsored by the *National Institute of Standards and Technology* (NIST) [20] and *Intelligence Advanced Research Projects Activity* (IARPA) [14]. The purpose of TREC is to support and encourage researcher in information retrieval community by providing infrastructure for large scale text retrieval, such as test data and test problems within a wide range of domains all related to IR.

Precision and Recall

Precision and recall is based on the assumption that for a specific query the documents in the system either belongs to a relevant set, that is the document is relevant to the query, or belongs to a non-relevant set, that is the document is not relevant to the query. Recall is defined as the ratio between the number of relevant documents retrieved by the system and the total number of relevant documents. Precision is defined as the ratio between the number of relevant documents retrieved by the system and the total number of retrieved documents. For a specific query documents are either relevant or non-relevant and the documents in the system are either retrieved or not retrieved. This is illustrated in table below which classify documents in four categories, true positive (TP), false positive (FP), false negative(FN) and true negative (TN).

Relevant?

Retrieved?		True	False
	Positive	True positive (Hits)	False positive (Unexpected result)
	Negative	False negative (Miss)	True negative (Correct rejection)

– Documents in the TP class is a hit. The document is relevant and retrieved by the system.

– Documents in the FN class is a miss. The document is relevant but not retrieved by the system.

– Documents in the FP class is unexpected result. The document is not relevant but is still retrieved by the system.

– Documents in the TN class is a correct rejection. The document is not relevant and not retrieved by the system.

In terms of these classes the precision and recall can be calculated as follows:

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$

Recall is a measurement that reveals if the system misses any documents. A high recall means that most of the relevant document where retrieved, but it says nothing about how many non-relevant documents that were retrieved. Recall is a measure of completeness or quantity. Precision can be seen as exactness or quality of the retrieve system. A high precision means that most of the retrieved documents where relevant, but says nothing about how many relevant documents that was not retrieved. This means that both recall and precision is a weak measurement without each other. If the system for example always retrieves all documents, the recall will always be high since all documents are retrieved. But the precision may be very low and lots of non-relevant documents might be retrieved. Another example, if the system just return one document but that single document is almost always relevant, the system will have a very high precision, but the recall might be very low and many relevant documents might be missing in the retrieved result. This means that recall and precision are tied together, if you increase one of them it is very likely that the other one reduces. A good example of this is a brain surgeon's job of removing tumor cells in a brain. When removing tumor cells in the brain you want to be sure you remove all of the cells, you want a high recall. On the other hand you do not want to remove too much of the brain and risk brain damage of the patient, you want high precision. This trade of becomes obvious and an IR system has the same sort of trade of but with documents instead of brain cells.

The precision and recall measurement is criticized for not taking the true negatives result in consideration. By not doing this the measurement can give a skewed picture of the systems performance. Consider the case were there are two relevant documents out of 1003, and the system retrieves two documents out of which one is relevant and the other non-relevant. The system has consequently missed one document. This means that $TP = 1$, $FP = 1$, $FN = 1$ and $TN = 1000$. The recall and precision then becomes:

$$\begin{aligned} Recall &: \frac{1}{1+1} = 0.5 \\ Precision &: \frac{1}{1+1} = 0.5 \end{aligned}$$

It seems like the system is not very good, however the system correctly rejected one thousand documents and just missed one document. In other words, the system might be very good, but that is hard to see when just looking at the recall and precision. To get around this problem one can measure *true negative rate* or *accuracy*, which both use the true negative result:

$$\begin{aligned} True\ nagative\ rate &= \frac{TN}{TN + FP} \\ Accuracy &= \frac{TP + TN}{TP + TN + FP + FN} \end{aligned}$$

In the previous example the true negative rate becomes $\frac{1000}{1000+1} \approx 0.999$ and the accuracy becomes $\frac{1+1000}{1+1000+1+1} \approx 0.998$. This result gives a more complete picture of the performance of the system.

A measure that combines recall and precision in one unit is the harmonic mean called the *F-measure* [30], which is the weighted harmonic mean of precision and recall:

$$F = 2 \times \frac{precision \times recall}{precision + recall}$$

3.2 Free Text Searching

To be able to free text search or full text search data is a requirement in many systems. Finding information fast in large data sets is something that can be almost impossible without the ability to do a full text search on the data. If there is a hierarchical structure present in the data, information can be found by navigating through the structure down to a level where the number of documents becomes manageable without the use of full text search. However if you cannot structure your data in this way or the number of documents at the bottom of the hierarchic is still too great, you will probably need full text search in order to find the data you are looking for.

With this in mind, the next step is to choose what technique to use to enable full text search. The first thing to consider is what type of data you are searching in. Maybe the data is structured in such a way that you do not need all techniques applied by a specialized full text search engine such as stopword removal, stemming and other special features (mentioned under 3.1). For example, if your system is a digital phone book where you want to be able to enter a name and get a list of phone numbers, your search does not need to have specialized text processing techniques. Here a standard relational database management system (RDBMS) without explicit support for full text search might be good enough and more suitable. As mentioned before under 3.1, unstructured data is typically the sort of data where full text search becomes a powerful tool for finding information. Not all data stored by a system have to be unstructured for a full text search function to be useful, it might be so that only one column or field has this property and that can be enough to consider using full text search techniques.

In Tieto's case the data (Trouble Reports) is semi-structured and lots of text stored for each TR is written by humans in a continues text flow describing the problem. It is in these text fields valuable information exists and it is this information developers want to be able to full text search in. However structure data is also something that each TR has and should also be searchable to provide the best possible result. Example of such data can be the status of a TR, which has a small set of predefined states a TR can be in.

To make a good comparison between different databases a set of requirements has been chosen which is the minimum requirement that will be accepted in terms of full text search support. These requirements have been chosen to fit the demands Tieto has on the IR system for the project in this thesis.

3.2.1 Requirements

The requirements on the database are based on the desired functionality Tieto has for the system being developed during this master thesis. The database should first of all be able to search in up to 100 000 errands with a maximum response time of a couple of seconds. The better response time the better it is. The database should support ranking of the search result. A strict boolean retrieval model is not accepted. The result should be ranked based on relevance in descending order with the most relevant result first. There are however not any huge demands on how good the ranking should be since it is hard to evaluate the relevance of the result. To fully evaluate several databases' ranking ability would take a lot of time and there is simply not time for that in this study. It should however be present and preferably be customizable if that need would emerge. The insert time of data does not really matter since there are not that many errands per day that is updated or created (around one hundred per day). So if an insert take a couple of seconds or a couple of minutes does not really matter. One of the most important requirements is ease of use. This is important because the less time that could be spent on setting up and maintaining the database the more time can be spent on the rest of the system. The time under which the system should be developed is limited and if time can be saved during development, then that time can be used to add more quality and functionality to the system, ending up with a better product. The need of full ACID transactions is not present. In database system one talks about ACID (Atomicity, Consistency, Isolation, Durability) transactions, which is a set of properties that guarantee that database transactions are processed reliably. This can be crucial if the system does not allow dirty reads or when multiple users update values which can lead to inconsistent result when transactions are interleaved.

The requirements is summarized in the following list:

– Response time under a couple of seconds for 100 000 entries.

– Should support ranking of search result.

– Good free text search support.

– Easy to use.

– Easy to maintain.

3.2.2 Databases

When choosing database you need to consider what requirements the database have. There are a lot of different databases out there and each kind specializes to work well under some circumstances. Since a database cannot be best at every area at once it is up to the user to choose the right kind of database for his/her application.

In this comparison we divide databases into two categorize NoSQL and traditional relational databases. There are other types of databases such as hierarchical database model and network model, but those will not be discussed here.

Traditional relational databases use well defined schemas to structure the data. All data put in such a database must follow the defined schema. Data is manipulated and created by using SQL (Structured Query Language) which is a special purpose programming language for relational database management systems (RDBMS). Data is stored in tables built up of columns. Each column specifies an attribute for that table. Each row in a table has a unique identifier, called a primary key, which can be used in other table as foreign key to create relations between tables. All inserted data must follow the defined tables, and if you later on want to change a table by for example adding a column all previous data must then be change to fit the new table. This means that all data is very well structured and each row has the same attributes which simplifies data retrieval and algorithms used by the database system. A relation in RDBMS is actually a set of tuples (rows) that has the same attributes [7]. This means that a table is a relation, and if you take some attributes from one table and join them with some attributes from another table you get a new relation or table. When you query a RDBMS you always get a relation (table) back. We will not be covering all technical details of relational databases but enough to get an understanding of how they are built and what characterizes them, so it becomes possible to evaluate them with respect to full text searching. The same goes for NoSQL databases.

NoSQL databases, where NoSQL according to many stands for "Not only SQL", are databases that use a looser defined consistency model and data schema. NoSQL uses SQL-like queries to retrieve and insert data in the database. NoSQL systems generally do not provide ACID transactions, updates are eventually propagated. Eric Brewer's CAP theorem [29] states that it is impossible for a system to simultaneously guarantee all three following properties: consistency, availability and partition-tolerance. In NoSQL databases the consistency property is often sacrificed in favor for the other two. One talk instead of "eventually consistent" meaning that updates are eventually propagated to all nodes in the system [5]. The benefit of not having the same constraints on consistency is that it becomes easier to scale the database horizontally over many servers and insert data dynamically. NoSQL databases, in contrast of traditional relational databases which support ACID transactions, has BASE which is an acronym that stands for *Basically Available, Soft state, Eventually consistent*. Basically NoSQL database management systems works well when you have unstructured data in large quantities and do not need the relational model and where eventually consistency is good enough. If this is the case then NoSQL is a good choice since it offers very good performance for key-value retrieval operation. But there are of course situations where the choice is not this easy and you need to look deeper into the differences to make the right choice.

In this comparison we will look deeper into the database MySQL [19] and Apache Lucene [2] with focus on full text search capabilities. The reason for choosing MySQL is that it is one of the most well known and used free RDBMS, which means that there is lots of information about it and it is well tested. MySQL has full text search support and is also used in other applications at Tieto which makes it an excellent candidate for this comparison. Apache Lucene is a high performance, full-featured text search engine library written in Java. It is chosen because it is one of the most well known and used text search engine database solutions available. The Lucene core library is used in a lot of full text search databases like Solr [3], ElasticSearch [9], Compass [6] and Hibernate Search [13]. Lucene promises fast and reliable document storage/retrieval with lots of full text search features. Lucene fits as a good candidate for this comparison because of its reputation and features as a full text search engine.

Some may not consider Lucene as a NoSQL database, but rather just as a full text search engine. However even if Lucene is just a Java library it has all characteristics of a NoSQL database and can most certainly be used as a database. And if you were to place Lucene in one of the two categorizes NoSQL and relational databases it clearly falls under the NoSQL category. Since it is a library it requires some programming by the user to provide interfaces for inserting and retrieving data. However if you do not want to write that sort of code you could choose some of the solutions built on top of Lucene and still get all benefits of it.

Other databases that were considered was, MongoDB [18], PostgreSQL [23], Sphinx [24] and some of the previously mentioned databases built on top of Lucene. MongoDB was not used because they did not support full text search index when the candidates were selected. They did however release this feature during the thesis project, but it was unfortunately too late. PostgreSQL's full text support did not seem to differ that much from MySQL and it look like it was harder to configure. Sphinx was rejected because it is not as well known as Lucene and it did not seem to have the same ease of use as Lucene or MySQL. Because so many promising solutions are built on top of Lucene, Lucene became a natural choice in this comparison. The things that can be concluded after reading section 3.1 and this section is that if your data is structured and you need to exploit relations between attributes your better of choosing a relational database, and if your data is unstructured and you are only interested in full text search your better of choosing a specialized software built for this purpose. The question is where the breaking point exists between these candidates. When is it time to consider the other option and what do you gain/loose when you switch? When you have semi-structured data this becomes harder to answer. It is this question the comparison hopefully will clarify.

MySQL

MySQL is one of the most popular open source databases used today. It is a relational database management system and was originally developed in Sweden. It has built-in support for full text searching with functions such as stopword removal, boolean retrieval model and what they call natural language full text search, which is full text search where the result is ranked based on relevance. MySQL uses a special full text index (inverted index) for performing the full text search operations. When doing a full text search you have to use special operators made for that purpose. The syntax differs from normal SQL queries and has quite limited expression capabilities compared to regular SQL expressions. The full text index is built up on one or multiple columns from one table. You cannot build up and index using values from different tables. This also means that you cannot perform join operations and then use the indices to perform a full text search. When indexing multiple columns the index simply treats all columns as one big column, and concatenates the values into one big string. This means that you cannot know in which column a hit occurred if you have indexed multiple columns in a table. This also means that you can not specify to search in a special column or set of columns, you are simply forced to search in all of the columns present in the index [34]. You can however build several indices on one table which means you can work around this problem to some extent. Besides stopword removal, MySQL also removes short words from the index. The length is set to four characters in default mode, but this value can be changed if the user wants to include words of short length. When performing a full text search with MySQL you use the syntax `MATCH(`*`col1, col2,...`*`)` `AGAINST(`*`expr, [search_modifier]`*`)`. The `MATCH()` clause takes a list of columns to perform the search on. These have to be specified in the same order as in the index and they must of course all be present in the index. `AGAINST` takes a set of keywords separated by space and a search modifier. The search modifiers available are:

– `IN NATURAL LANGUAGE MODE`

– `IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION`

– `IN BOOLEAN MODE`

– `WITH QUERY EXPANSION`

The modifiers `IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION` and `WITH QUERY EXPANSION` are just aliases of each other. So there are basically three different modifiers. The modifier `IN NATURAL LANGUAGE MODE` is the default modifier if nothing else is specified. This modifier performs a natural language search on the keywords in the `AGAINST` clause on the columns specified in the `MATCH` clause (provided that an index exist for the specified columns). The result is ordered after relevance in descending order where the most relevant row is the one where the search string matches the searched columns best.

The `WITH QUERY EXPANSION` modifier does something that is known as blind query expansion. This function can be useful if the query is too short to find relevant result. It works by performing the query twice where the second search phrase is the first original search phrase concatenated with the words from the top rated documents from the first query. If the first query for example is "animals" the top rated documents may contain words like "cat", "dog", "turtle" and so on. In this way you broaden the query. This type of search is not suitable when the original query contains many terms since it will broaden the query too much and the query can lose its meaning.

The `IN BOOLEAN MODE` modifier simply enables a boolean retrieval model where you can specify terms by boolean operators. The operator for "AND" is "+", for "NOT" is "-" and "OR" is space " ". An expression in the `AGAINST` clause could look like this: "+yellow green -black". This simply means that the result must contain the term "yellow" and it may contain the term "green" and must not contain the term "black".

MySQL implementation of full text searching has several limitations. There is for example only one type of ranking, frequency ranking. The more times a keyword occur the better the ranking. The position of the words in the documents is not stored which means that proximity does not contribute to relevance. The size of the index is another problem in MySQL. The index works well when the index fits in memory, but if it does not fit, it can be very slow, especially when the fields are large [34]. Compared to other MySQL index types, the full text index can be very slow for insert, update, and delete operations. In the book *High Performance MySQL* [34] they list the following facts for MySQL regarding full text search:

– Modifying a piece of text with 100 words requires not 1 but up to 100 index operations.

– The field length does not usually affect other index types much, but with full text indexing, text with 3 words and text with 10,000 words will have performance profiles that differ by orders of magnitude.

– Full text search indexes are also much more prone to fragmentation, and you may find you need to use `OPTIMIZE TABLE` more frequently.

– Full text indices cannot be used for any type of sorting, other than sorting by relevance in natural-language mode. If you need to sort by something other than relevance, MySQL will use filesort.

Besides the full text functionality MySQL has the standard features of a RDBMS such as ACID transactions, join operations where you can join on attributes from one table with attributes from another and end up with a merged table.

Apache Lucene

Apache Lucene is a full featured text search engine library written in Java. Lucene in one of the most used open source search engines available. It has lots of full text search features and offers real good performance. Since it is a java library it requires that you write you own code for building up you database. Since many applications want to integrate the database in the system flow and be able to insert and retrieve data programmatically this is not such a big step.

A Lucene database is built up by documents. Each document is built up by fields, which can be compared with columns in a table which relational database is built up of. Lucene combines a *Boolean model* with a *Vector Space Model* to rank documents. Documents are first narrowed down by the boolean model based on the use of boolean logic used in the query and then ranked by the vector space model. This means that you can use boolean operators in the query and still get a result ranked by relevance. The fields in each document does not have to be the same and can vary from document to document, just like schema-less tables in other types of NoSQL databases. Lucene supports text preprocessing like stemming and stopword removal to increase the precision and quality of the search results.

Lucene supports lots of information retrieval search features. The list below summarizes some of the most significant ones.

– **Proximity searches** - You can specify maximum distance between two search terms in the query. The distance is measured in terms (words) in the documents.

– **Phrase searches** - You can search for a whole phrase in the documents text by putting your search phrase in quotation.

– **Field search** - By specifying the name of a field followed by a colon ":" and then a search term, Lucene will search for the term only in that field. If a field is named "title" and you want to search for "computers" in the documents' title you write "title:computers".

– **Boost terms** - Lucene supports boosting individual terms in a query which leads to a higher ranking of documents with that term. The boost is specified by an integer.

– **Boolean operators** - You can use boolean operators in the query. The use of boolean operators does not make the retrieval model boolean, the result will still be ranked.

– **Fuzzy search** - The fuzzy search function can be used on individual terms in the query. The fuzzy search is based on the Levenshtein Distance. The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other [31]. This means that searching for the term

”foam” could return documents containing the term ”roam” depending on what the distance is set to.

- **Field weighting** - On insertion the weight of a field can be altered to increase the relevance of that field.

- **Range searches** - You can search for a range in a specific field. For example if you have a field named ”year” and you want all documents where the ”year” field is between 2000 and 2013 you write ”year:[2000 TO 2013].

- **Wildcard searches** - Lucene supports both single and multi character wildcards in the search query.

Lucene implements the ACID transactional model, but with the restriction that only one process can make changes to the index at once. The book *Lucene in Action* [16] states the following about Lucene’s ACID transactional model:

- *Atomic* - All changes done with the index writer class are either committed to the index, or non are; there is nothing in-between.

- *Consistency* - The index will also be consistent; for example, you will never see a delete without a corresponding add in case of updating a document. You will always see all or none of the indexes if you add an index to a existing one.

- *Isolation* - While you are making changes to the index nothing will be seen by other processes until you successfully commit. When reading the index you will only see the last successful commit.

- *Durability* - If your application hits an unhandled exception, the JVM crashes, the OS crashes, or the computer suddenly loses power, the index will remain consistent and will contain all changes included in the last successful commit. Changes done after that will be lost. Note that if your hard drive develops errors, or your RAM or CPU flips bits, that can easily corrupt your index.

The restriction that only one process can make changes to the index at once needs to be handled by the application since Lucene is just a Java library. It is up to the application to handle multiple writers and make sure that only one process writes to the index at a time. Note that there are mechanisms that will lock the index if a process is writing to the index so that you do not end up with a corrupt index when multiple processes try to access the index. There is however no restrictions on how many readers an index can have. The index does not become locked for reading when another process locks it for writing.

3.2.3 Comparison

First of all we will take a look at the general differences between NoSQL and relational databases. This will clarify the main differences between the two types of databases. After that we will take a deeper look in to two candidates, one from each side, namely MySQL and Apache Lucene.

Relational and NoSQL Databases

Choosing between a relational database and a NoSQL database all depends on the type of data you are storing and the size of the data. NoSQL databases often sacrifices consistency in order to scale better horizontally over many servers [5]. NoSQL servers offer a more loosely defined schema for storing data compared to relational databases where the schemas are very detailed. This has both advantages and disadvantages. Some of the advantages of having a more loose data schema are that it becomes faster to insert data because you do not have to verify that the data fits the schema in the same manner as in a relational database. So if your application must handle huge amount of insertions NoSQL databases often gives better performance. They are ideal for key-value storage. The disadvantages are if your data has relations that you want to exploit. Storing data in a relational model can save space and time if you have many relations, since you can avoid duplicating data. The relational model enables you to avoid duplicating data and instead let you point many attributes to one place (many to one relationship) and changing data at one place instead of many places. A relational database is better when you have structured data with natural relations between attributes. If you have semi-structured data the question becomes what type of operations you want to do on the data and how sensitive your data is when it comes consistency. Relational databases are suitable if:

- Your data has a natural structure and many relations.
- ACID transactions are important to the application.
- You want to perform join operations.
- Order the result by a specific column or field.

NoSQL databases are suitable if:

- Your data is unstructured and does not fit into a defined set of columns or fields.
- Horizontal scalability is important.
- You use your database as a key-value storage.
- You have huge amount of data with many writes.

When it comes to full text search it all depends on the specific database systems support of full text search. Using a relational database or NoSQL database that does not have full text support will never be efficient. Scanning through all data sequential and performing string operations to match terms in the database to the query is highly inefficient since you have to iterate over all terms for every search. Even an index in the database won't be enough since a full text index and a regular index does not work in the same way. A full text index (inverted index) will store all terms in a lookup table for efficient search of all the text in a column or set of columns. A regular index is built to match the whole column with a value and will there for not be efficient for searching for individual terms within a column. What a regular index (b-tree) normally does is match the beginning of a search term with the beginning of the word in a column, and can than decide which way to traverse in the tree structure to match the search word with the word in the columns. The question really does not become which type of database (relational or NoSQL) you should choose if you want to be able to do full text search. It does not matter if you choose a relational database or a NoSQL database, it really comes down to each system's full text search capabilities. When choosing database you have to look at what kind of data you have and the requirements, besides full text search, that the database has. If your data needs to be stored in a relational database you should look for such a database with the full text support you need. If there is no such database you may want to consider a hybrid solution. You could have a relational database as persistent storage for all your data and use another system to create a full text index. The same goes for NoSQL database. If you have found a database that matches your needs besides full text search capabilities, you could use another system to create a full text index.

To summarize, the full text search capabilities of a system requires that the system has special support for full text search. Choosing between relational and NoSQL databases really depends on the type of data you are storing and the requirements you have besides full text search support. You need to look at each system's full text features to see if they match you needs.

MySQL and Lucene

MySQL and Lucene were chosen because these solutions represent the most popular and documented versions of relational databases and NoSQL databases respectively. They are also the databases chosen as candidates for the project in this thesis. Other databases were considered for the project but were rejected since none of them were as well known and they did not offer any major advantages over the already chosen ones.

Table 3.2 shows the full text search features supported by the two candidates.

Feature	**MySQL**	**Lucene**
Keyword search.	✓	✓
Phrase search.	✓	✓
Use of stopwords.	✓	✓
Boolean retrieval model mode.	✓	✓
ACID transactions.	✓	✓
Blind query expansion.	✓	-
Use of wildcards	✓(Only in boolean mode)	✓
Boolean operator with result ordered by relevance.	-	✓
Search specified to individual field.	-	✓
Proximity search.	-	✓
Stemming available	-	✓
Fuzzy search	-	✓
Boost relevance factor for field	-	✓
Boost relevance factor for search term	-	✓
Schema-less tables.	-	✓

Table 3.2: Full text search features for MySQL and Lucene

It is clear from table 3.2 that Lucene has greater full text search capabilities than MySQL has. However Lucene requires more from the developer than MySQL does. Lucene is a Java library and requires knowledge of software development. For instance, it is important that only one process make changes to the index at once or you could end up with a corrupt index.

In Lucene it is up to the developer to create an environment around the index that handles multi-write situations. MySQL has this functionality built in and the user does not need to worry that the system might crash. Lucene has lots of error handling built in so there is no need to create everything from scratch, but the developer must take care of the produced error messages that might occur to prevent the system from crashing. Write collisions can also happen in MySQL, but it is a standalone system and will not crash because of this. The separation from the application and database becomes clear with MySQL which it does not with Lucene.

Both databases offers good performance and the requirement of being able to search in 100,000 errands should not be a problem in any one of them. However write operations can be expensive in MySQL. In the book *High Performance MySQL* [35] the authors actually recommends using a specialized system such as Lucene or Sphinx [24] as a complement to gain performance when doing full text search. So if performance is the biggest concern one should not choose MySQL over Lucene. Choosing MySQL over Lucene makes sense if you have structured data, want to write less code and when the full text features is sufficient for your needs. Lucene has more full text search functionality, and with more functionality comes more complexity for the developer. The extra complexity added by more sophisticated functions might be worth considering when building an IR system.

A general opinion that can be seen on various forums on the internet is that Lucene should not be used for persistent storage. One should instead have the data store somewhere else and be able to rebuild the whole index if needed. It is hard to find any valid arguments for not using Lucene as persistent storage, but one can see the benefits of having two systems working together. One persistent storage system, relational database or some NoSQL database, and one full text search system such as Lucene. This can give you the best from two worlds and you do not have to compromise if you are choosing between them. However this requires synchronization between the systems and it becomes harder to have a completely consistent system. It basically comes down to each system's requirements. There simply does not exist a general solution. Not having Lucene as persistent storage also makes sense if you doubt your programming skills and do not believe that you have the knowledge to build a stable system.

The things that can be concluded by this comparison are that Lucene is superior to MySQL when it comes to full text searching, but requires more skill from the developer. The schema-less data structure makes it easier to handle big amounts of unstructured data. If you are looking for a standalone system MySQL is the clear choice. However there are standalone system using Lucene at its core that should be considered, such as ElasticSearch [9] or Solr [3] if that is the only concern.

Reflections

Originally the comparison was meant to include a performance benchmark between the two final candidates. But as the understanding of the databases grew it became clear that this was not needed. It did not come down to just how fast a query could be executed or how well the ranking algorithm worked. They are different from each other in terms of what they are designed for, which make it easier when you figured out what your application actually needs. It became clear which one to choose without conducting this sort of benchmark, and when that became clear it seemed like creating the benchmark anyway would have very little value and it was there for not carried out. Another major contribution to the benchmark not being carried out is the fact that the book *High Performance MySQL* [35] actually suggested using Lucene for better performance. One should know that the authors of this book are performance experts and that two out of three previously worked at MySQL AB.

The comparison brought light upon an interesting solution, namely using the two candidates together in a system. Not seeing the candidates as competitors but as complements to each other. This of course requires more work, but could be an elegant solution if the search functionality does not have very hard requirements on consistency.

3.3 Database for LTE TR Tool

One of the most important requirements of this tool is performance. The tool should be able to fetch information fast with real time feeling. The solution should also lay the groundwork for further development such as automatic matching of TRs. There for the choice in this case fell upon Apache Lucene, because of its impressive set of features and leeway for custom modifications. The idea of using both Lucene and a relational database, such as MySQL, was tempting but since the actual data is already stored at MHDB, this was not chosen. If the solution should somehow loose data or have to go through some major redesign the data stored in the index could just be fetched again and the index rebuilt.

Chapter 4

Accomplishment

This chapter describes the different steps in the project. How the project was planned and how the different parts were carried out.

4.1 Preliminaries

Before building of the solution could start there were lots of things that needed to be clarify. Tieto had a document with a brief description of their problem and what they needed to be able to tackle that problem. To get a full understanding of what they needed and their expectations of the project, the current way of finding similar errands had to be understood. This meant talking to developers at Tieto and letting them explain how this was done. This also included getting knowledge of the work flow with TRs (errands) and how TRs are build up in their system. The next step was to understand how they would like to work. This was also done by talking and listening to developers needs and desires. This gave a more complete picture of the problem and what was expected of the project. Tieto had a rough idea of how to build up a system that would fulfill their needs. With the guidelines of the supervisor from Tieto the system design was created. Now a project plan could be created to divide the work into smaller more well defined tasks.

Before the building of the actual system could start there were lots of options that had to be considered and decisions to be made. Tieto did not have any special requirements on what type of database the system should use. This meant that different database solutions were studied to try and find a suitable one for this particular system. In this search of database systems both relational databases and NoSQL databases were considered. Focus was put on the databases' full text search capabilities. The programming language was also free of choice and had to be chosen. This applied for all parts of the system. Since the interfaces towards MHDB were built in Java this choice was quite easy, and the choice fell upon Java. Another decision that had to be made was choosing between the two different APIs provided by MHWeb. This was done by setting up meetings with different people from Tieto who had experience of these APIs, reading the available documentation and comparing the APIs to each other.

4.2 How the work was done

After understanding what Tieto was looking for and system design was created along with a project plan of how to achieve this in the given time, the system could start to take shape. All details were not settled but enough to get the project going. Throughout the project whenever there were different alternatives to choose between, the alternative were studied, one or two were chosen and a small prototype was created to see if the chosen alternative had the potential to be part of the solution. So lots of prototyping was done whenever there was doubt of how to solve a problem.

The system design had three distinct parts:

1. Fetching data and store it.
2. Searching in the data.
3. Displaying the data.

The parts were constructed in this order. First a conceptual prototype of all three parts was created to lay the groundwork for the system. Then all parts were refactored and broadened to include more functionality and be more fault tolerant. A close discussion with the supervisor from Tieto was maintained throughout the whole project. The development of the system took place at Tieto, which meant that the supervisor could be consulted on a daily basis. This helped the project to stay on the right course and questions could be answered very fast.

The documentation of the project was done parallel to the development of the system. But more time was dedicated to the development of the system in the beginning of the project, and in the end of the project more time was spent on documentation.

Chapter 5

Results

This chapter consists of a technical description of the whole system *LTE TR Tool*. The system has three main Java projects that form the complete system. The projects each have a clear responsibility namely storing, searching and displaying TRs and are described in section 5.2, 5.3 and 5.4 respectively.

5.1 System Overview

For storing TRs in a local database Apache Lucene [2] has been used. Apache Lucene is a high-performance, full-featured text search engine library written in Java. The TRs from MHWeb are collected via the JMS (Java Message Service) client and store in a Lucene index. Accessing the index is done via a web service using Apache Axis [1]. Apache Axis is an implementation of the SOAP (”Simple Object Access Protocol”) submission to W3C [26]. Finally the search result returned from the web service is displayed on a web page using JSP (Java Server pages), JavaScript and HTML. The design of the system can be seen in figure 5.1.

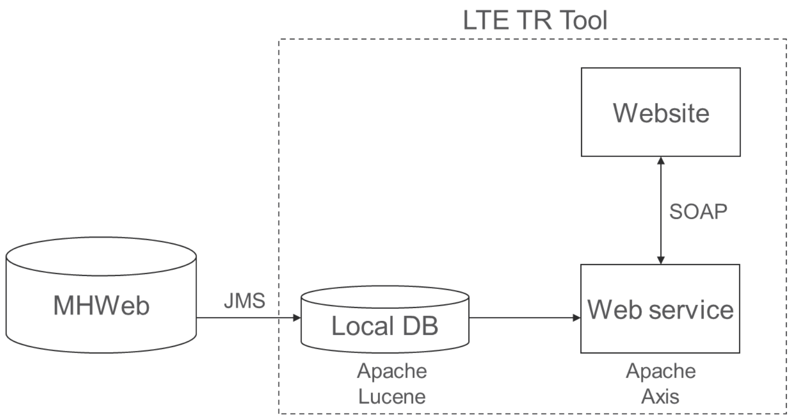


Figure 5.1: System overview

5.2 Database

Collecting TRs from the MHDB is done with the JMS (Java Message Service) API which is provided by MHWeb. The way this client works is that you register an application with MHWeb so that your application is unique. You can see this registration as an account for your application. To get TRs to your application you make a subscription on the TRs you want. Subscriptions are made at MHWeb's website. A subscription can be specified to match desirable TRs, you can for example make a subscription on all TRs assign to a specific MHO (Modification Handling Office, a virtual organization responsible for a specific task within the TR flow) or all TRs created before a specific date or all TRs created by a specific user etc. This way of specifying subscriptions is powerful and has many options so that you get the TRs you need with minimal configuration. When a subscription is made to your registered application MHWeb queues up the TRs matching your subscription. You can now configure the JMS client to access this queue and store the TRs you have subscribed. If changes occur to any of your subscribed TRs these TRs are queued up again. In this way your local database will be up to date with the MHDB as long as your JMS client is running.

The local database in the system is populated with usage of the JMS client. The client can use one or more "Writers" to store data. Writer is an abstract class for writing and removing subscribed objects into a specific data store. The client provided by MHWeb comes with two writers, one for storing the objects in a MySQL database and an XML-writer for storing the TRs in XML to disk. You can create your own writer and use it with the client if neither of the included writers fit your needs, which is the case in this project.

5.2.1 Implementation of Writer

The abstract class `Writer` has three public methods:

– `write(Header header, String data)`
  This method is called when a TR is received from the queue at MHWeb when the TR is ether new and fits the subscription or if a TR already in the subscription has changed.

– `remove(Header header)`
  This method is called when a TR has changed a no longer fits the subscription.

– `close()`
  This method is called before the JMS client stops.

The class `Header` contains information about the TR such as a unique identifier, time stamp etc. The parameter `String data` is a TR stored in a XML string. The client comes with a deserialization class for converting the XML string in to a TR object which makes it easier to extract information from the TR.

The client supports multithreading which means that when running the client there will be more than one "writer" instance, if multithreading is enabled. Since a Lucene index only can be written to by one process at a time the implementation of the `Writer` class needs to handle this feature or the index cannot be built properly.

To handle multithreading a singleton pattern has been used to create a single object which the `Writer`-objects uses to write to the index. The class design can be seen in figure 5.2. The class `LuceneWriter` extends the the abstract class `Writer` and it is this class the JMS-client will create multiple instances of when multithreading is enabled. To get access to the index LuceneWriter uses the class IndexWriterQueuer which in turn has an IndexWriter. The IndexWriterQueuer class in responsible for deserializing TRs and creating QueueItem objects which can be fed to the IndexWriter. IndexWriterQueuer gets a IndexWriter from the class LuceneIndexWriterAPI. LuceneIndexWriterAPI ensures that only one instance of IndexWriter is created and then shared between the writer objects.

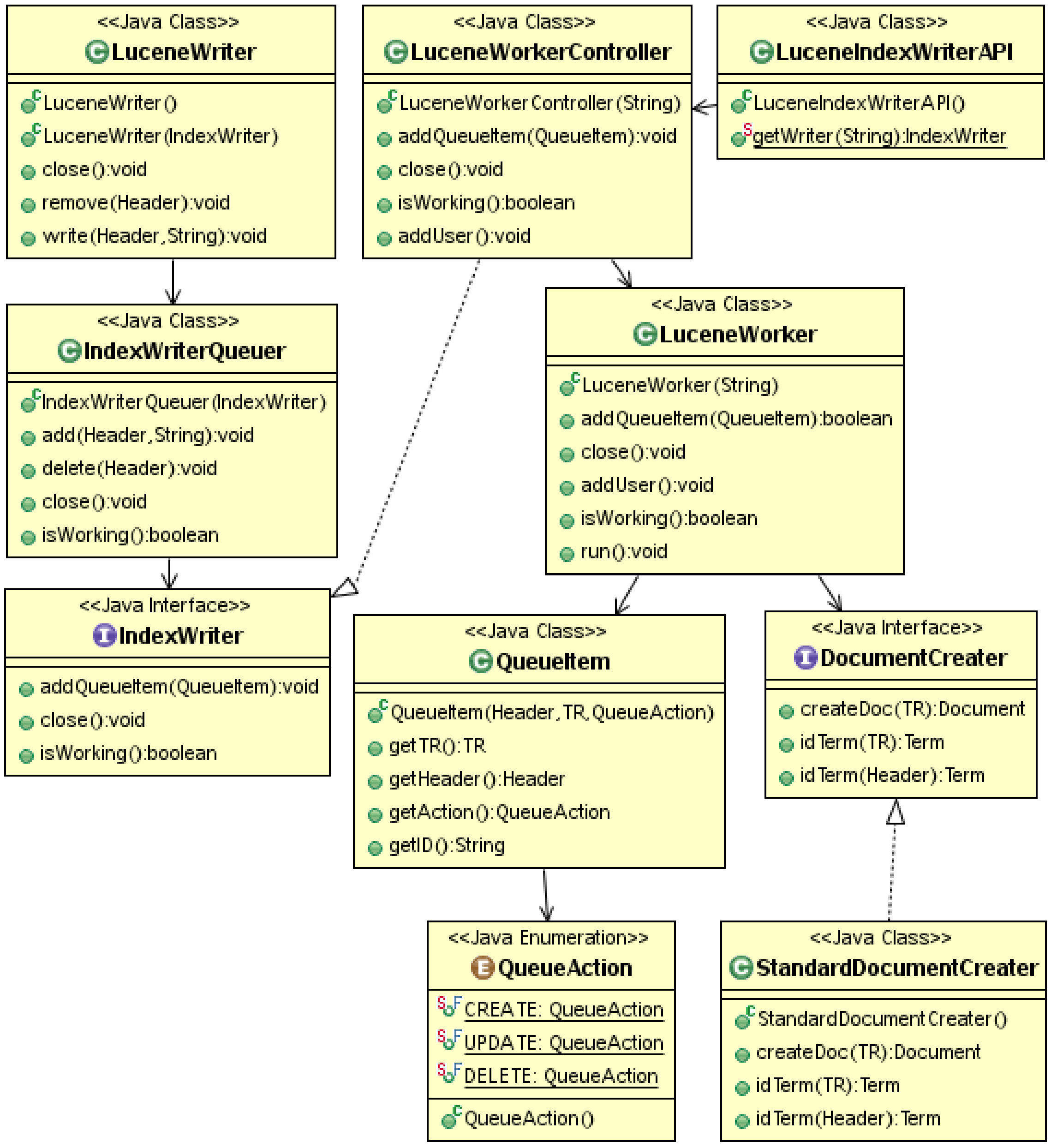


Figure 5.2: Class diagram of writer

The class `LuceneWorkerController` implements the `IndexWriter` interface and is the class the `IndexWriterQueuer` objects ultimately use. The methods in `LuceneWorkerController` are synchronized so that only one process can access them at a time. `LuceneWorkerController` in turn has one instance of the class `LuceneWorker`. `LuceneWorker` implements Java's interface `Runnalbe` which makes it possible to run the class in a separate thread. When `LuceneWorkerController` constructor is invoked a new thread is created that runs a `LuceneWorker`. `LuceneWorker` is the class that writes to the index and handles the creation of the index if it does not exist. It is in `LuceneWorker` the TR objects are converted to Lucene documents, via the interface `DocumentCreater` (this is explained in more detail later on). `LuceneWorker` internally has one buffer queue which holds `QueueItem` objects. A `QueueItem` object holds an `QueueAction` which decides if a TR is to be deleted or added. `LuceneWorker` and `LuceneWorkerController` has mechanisms for starting, pausing, killing and restarting the `LuceneWorker` (this is explained in more detail later on).

The dataflow for TRs in this implementation of `Writer` can be seen in figure 5.3.

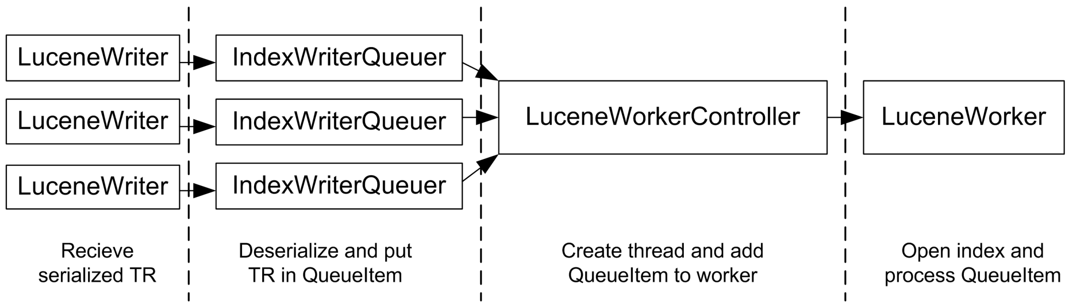


Figure 5.3: Dataflow for TRs using three threads.

LuceneWorkerController and LuceneWorker

The class `LuceneWorkerController` is responsible for creating a thread running a `LuceneWorker`, waking the thread if it paused, restarting thread if it has stopped running and adding `QueueItem` objects to the queue in the thread. The thread running the `LuceneWorker` is created when the constructor of `LuceneWorkerController` is invoked. When `LuceneWorkerController` method `addQueueItem(QueueItem)` is called it tries to add the `QueueItem` to the `LuceneWorker`. If the thread running the `LuceneWorker` is paused it will be unpaused by triggering an thread interrupt and if it is dead it will be restarted. A `QueueItem` can only be added if the thread is alive, which means that if a restart is needed the calling thread needs to wait until the `LuceneWorker` thread is restarted and then try to add the `QueueItem` again. The method `addQueueItem` in `LuceneWorker` will return false if the thread is dead. Waiting for a restart is done with Java's `wait` method which is available for `java.lang.Object` (in other words all objects). When the `wait()` method is invoked the calling thread will wait until another thread invokes the method `notify()` (which is also available for all objects). By using this mechanism busy waiting is avoided since the thread calling `wait()` will sleep while waiting. To be able to use this mechanism the thread calling `wait()` and the thread calling `notify()` needs to synchronize over a shared object. `LuceneWorkerController` synchronizes over the `LuceneWorker` object and `LuceneWorker` synchronizes over itself. The full algorithm in the `addQueueItem(QueueItem)` in `LuceneWorkerController` can be seen under listing 5.1.

Listing 5.1: Algorithm for `addQueueItem` method in `LuceneWorkerController`

```
1. Try to add QueueItem to LuceneWorker
2. If adding QueueItem to LuceneWoker is successful
       2.1 Interrupt LuceneWorker
3. Else
       3.1 Create new thread
       3.2 Start thread
       3.3 Wait until thread has started.
       3.4 Try to add QueueItem to LuceneWorker
       3.5 If adding QueueItem ot LuceneWorker is successful
              3.5.1 Interrupt LuceneWorker
       3.6 Else
              3.6.1 Throw Exception
```

The class `LuceneWorker` is responsible for writing changes to the Lucene index. `LuceneWorker` internally has a queue holding `QueueItem`. `LuceneWorker` implements Java's interface `java.lang.Runnable` so that it can be run in a separate thread. A class implementing this interface must define a method of no arguments called `run`. When a thread is started the `run` method will be invoked by the thread.

Listing 5.2: Algorithm in `run` method in `LuceneWorker`

```
1. Get write lock
2. Set variable run to true
3. Notify all threads waiting on current thread
4. Release write lock
5. While run is true
       5.1 process queued QueueItems
              5.1.2 Open index if it is closed
              5.1.3 Get QueueItem from queue
              5.1.4 Write changes to index
       5.2 If queue is empty
              5.2.1 pause thread
              5.2.2 If thread was not interupted
                     5.2.2.1 Get write lock
                     5.2.2.3 If queue is empty
                            5.2.2.3.1 close index
                            5.2.2.3.2 Set variable run to false
                     5.2.2.4 Release write lock
```

DocumentCreater

The class `StandardDocumentCreater` implements the interface `DocumentCreater`. `DocumentCreater` has three methods:

- `createDoc(TR):Document`
  Creates a Lucene document from a TR object.
- `idTerm(TR):Term`
  Returns a Lucene `Term` object with the ID of the TR. The `Term` object is used to remove documents from the index.

– `idTerm(Header):Term`
  Has the same function as the previous method but takes a Header object instead of a TR object as argument.

In the method `createDoc` the data from the TR object is extracted and put in a new Lucene document. The TR class is simply a data storage class whose methods return strings, such as observation text, id, status, registration date etc. Some methods returns other objects in case the nature of the data has entries, for example progress information has multiple entries where each entry carries date, time stamp and some progress information. These objects only stores strings and are not nested yet another level.

A Lucene document is built up by fields. Each field has a name and a value. When building a Lucene document from a TR object one could simply directly call the TR objects methods in the code and create and name Lucene fields to store in the document. When you later want to extract the data from the Lucene document you need to write similar code to fetch each field. If you where to change the fields' names you would have to change the code that extracts the information as well. To avoid this double work an enum type has been created that can be used both when storing and when fetching data.

Each element in the enum type is named after the Lucene document field name and stores the name of the method in a TR object that returns the data to be put in the field, a description of the field, if the method returns objects instead of strings and the name of the wanted methods in the possible returned objects.

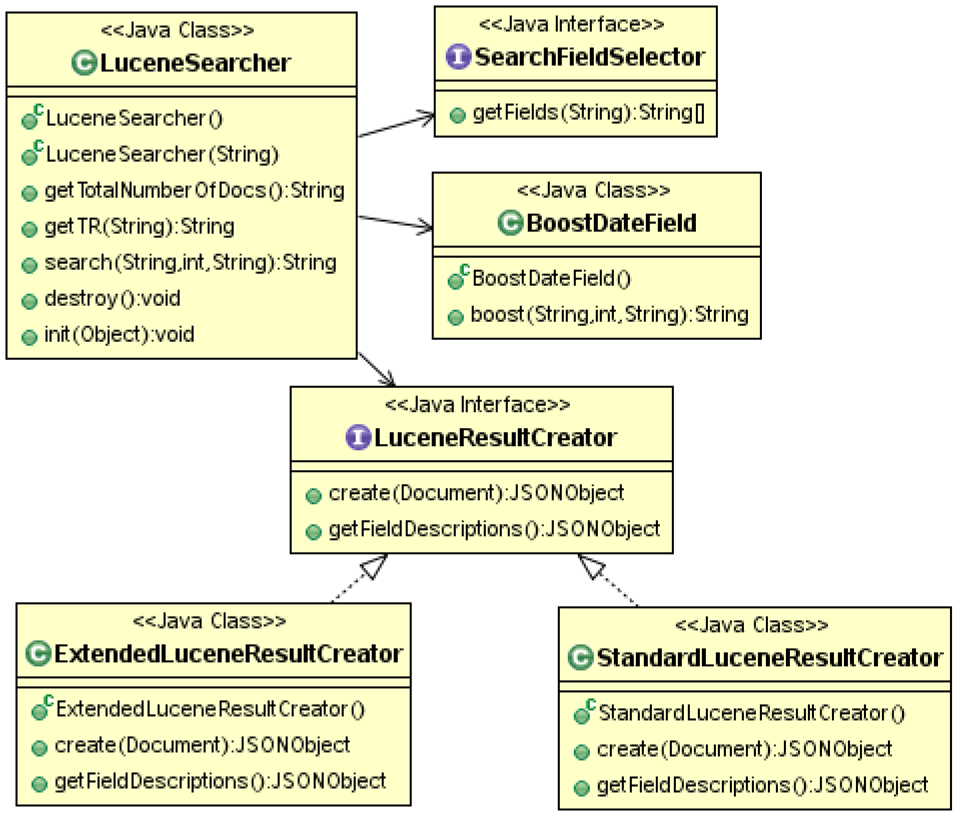
The `createDoc` method uses reflection with the help of this enum type to extract the data from the TR objects, see listing 5.3.

Listing 5.3: `createDoc` algorithm

```
1. Create a new Lucene document
2. For each enum element
   2.1 Get method name
   2.2 If method returns objects
      2.2.1 Get names of object's methods
      2.2.2 For each object
         2.2.2.1 Invoke each method and store values in the Lucene document
   2.3 If method returns string
      2.3.1 Invoke method and store value in the Lucene document
3. Return document
```

5.3 Web Service

The Lucene index is exposed via an Apache Axis [1] web service. This web service can only be used to read in the index, not write. Axis is an implementation of the SOAP ("Simple Object Access protocol") submission to W3C [26]. It uses a XML based protocol to define a messaging framework. The framework has been designed to be fully independent of any programming model and other implementation specific semantics. Axis encapsulates this technique and enables deployment of a web service interface from a Java class. The class that builds up the web service methods is `LuceneSearcher`, which can be seen in figure 5.4.



<<Java Class>>
LuceneSearcher
LuceneSearcher()
LuceneSearcher(String)
getTotalNumberOfDocs():String
getTR(String):String
search(String,int,String):String
destroy():void
init(Object):void
<<Java Interface>>
SearchFieldSelector
getFields(String):String[]
<<Java Class>>
BoostDateField
BoostDateField()
boost(String,int,String):String
<<Java Interface>>
LuceneResultCreator
create(Document):JSONObject
getFieldDescriptions():JSONObject
<<Java Class>>
ExtendedLuceneResultCreator
ExtendedLuceneResultCreator()
create(Document):JSONObject
getFieldDescriptions():JSONObject
<<Java Class>>
StandardLuceneResultCreator
StandardLuceneResultCreator()
create(Document):JSONObject
getFieldDescriptions():JSONObject

Figure 5.4: Class diagram for web service

5.3.1 Web Service Interface

Three methods are exposed in the web service:

- `search(String, int, String)`
- `getTR(String)`
- `getTotalNumberOfDocs()`

It is these three methods that can be used in the web service. All methods return data in a JSON formatted string. JSON is described under 5.3.2.

The method `search(String, int, String)` is used for searching in the index. The first argument is the search query. The query is a normal string that contains the keywords, phrases or any other operators included in the Lucene query syntax. The syntax will be explained more under 5.4 where the website that uses the web service is described. The second argument is the maximum number of hits returned by the method. By keeping the maximum number of hits down the response time will be faster. The third argument is *search properties*. The search properties are JSON encoded attributes that can be used to tell the web service that you want a special kind of search. The attributes can be encoded via a special class that is used both for encoding and decoding the properties. There are currently three search properties available. Two binary properties, namely *fuzzy search mode* and *escape special characters*. If *fuzzy search mode* is set to true then Lucene's fuzzy search function will be used on all terms in the query. The fuzzy search function provided by Lucene is mentioned under section 3.2.2, it basically means that Lucene will get hits on terms that does not match the original term exactly but is very close. For example, searching for the term "goat" could return hits on the term "boat". If the property *escape special characters* is set to true all characters that are part of the Lucene query semantics are escaped. For example, if the query contains parentheses they will be escaped and not interpreted by Lucene as a part of an expression in the Lucene query syntax. Normally parentheses are used to group terms together. The last search property is *boost time relevance* which is used to increase the weight on a date field, so that a date closer to the current date is weighted higher and there by ends up higher in the search result. The attribute for this property is simply an integer from one and above. How this is done internally in explained later. When the method `search(string, int, String)` is invoked a `SearchFieldSelector`'s method `getFields` is called. This method returns an array of document field names. The query is to be matched to these fields in each Lucene document in the index. Lucene will search the index and sort the result by relevance to the search query and return a list of document. The `search`-method will not return the document as they are, since there is quite a lot of text for each document. A set of document fields are selected that is relevant, so that one can distinguish the hits from each other. This set is quite small so that it becomes easy to overview the search result and pick the hits that seem most relevant. The class `StandardLuceneResultCreator` is used to extract the data in these fields for each document and then encode in JSON format. Listing 5.4 explains the algorithm in the `search`-method.

Listing 5.4: `search` method

```
1. Apply the search properties to the search string
2. Get the field names to search in
3. Search in the index in the chosen fields
4. For each document (or until max is reached)
       4.1 Extract a the values for a small set of fields
       4.2 Encode in JSON format
       4.3 Add encoded result to a result array
5. Return the JSON encoded array of search hits.
```

The method `getTR(String)` is used to fetch one TR. The argument for this method is a TR's unique identifier. This identifier comes from MHDB and is the reference used throughout all systems to identify a specific TR. This method is similar to the search method, but it does only search in one field, the field holding the unique identifier, and it does not extract just the values from a small set of fields, it extracts them all and encode the values in JSON format. This is where the class `ExtendedLuceneResultCreator` is used to extract the values from a hit, and encode it in JSON format.

The method `getTotalNumberOfDocs()` returns the total number of documents (TRs) stored in the index, and a timestamp of when the index was last modified. The timestamp can be useful to make sure the index is up to date.

Boost time relevance

The search property for increasing relevance if a date field is close to the current date is applied on a field carrying the date of when a TR was last modified. Lucene allow you to match documents whose field values are between a lower and upper bound. The syntax looks like this: `field:[lower bound TO upper bound]`. For example, `age:[18 TO 25]` will return results where the field named "age" contains values between 18 and 25. Lucene also lets you boost a search term, so that it has higher relevance in the search result. For example, `term^x` will boost the term by a factor of x, where x is a rational number. These two features are used to increase the time relevance of a TR in the result. By setting the upper bound to the current date and the lower bound to the current date minus seven days and boosting this search term with an appropriate factor, one can increase the relevance for TRs modified in the last seven days.

This technique is applied for five date ranges:

– One week old

– One month old

– Six month old

– A year old

– First of January 1970

This is the lower bound for the five ranges and the upper bound of each range is the preceding range's lower bound. Since the first range does not have a preceding range the upper bound is set to the current date. Each range is assigned a boost factor where the first range has the highest number and the preceding have lower and lower.

By using this technique one can increase the relevance of TRs that were recently modified. The limitation of this technique is that you place the date range in a quite large span. A TR that was last modified two months ago will get the same boost as a TR last modified four month ago. However the idea is not that the result should be sorted by the last modified date, the result is still to be sorted by relevance to the search query, which makes this technique a good compromise.

The original query is simply extended with these boosted ranges if this search property is used.

5.3.2 JSON

JSON stands for JavaScript Object Notation and is a lightweight, text-based, human readable, language independent data interchange format [8]. It enables you to represent simple structured data and associative arrays in in a human readable text. It resembles XML in some ways, but is more lightweight and less complex. JSON support the following basic data types:

– Number

– String

– Boolean

– Null

– Object

– Array

Objects are a set of name/value pairs. An object begins with "{" and ends with "}". Each name is followed by a colon, ":", and then the value. Each name/value pair is separated by a comma, ",".

An array is unordered collections of values. A value is any of the basic data types. An array begins with "[" and ends with "]". The values are separated by a comma, ",".

Listing 5.5 shows an object that describes a book.

Listing 5.5: JSON representation of a book

```
{
        "title": "Fancy book title",
        "year": 1999,
        "authors": [
                {
                        "firstName": "Arch",
                        "sureName": "Stanton",
                        "isAlive": false
                },
                {
                        "firstName": "Jack",
                        "sureName": "Smith",
                        "isAlive": true
                }
        ]
}
```

5.3.3 Session Handling

The web service supports session handling which means that a session can be maintained for more than just one method call to the web service. The class `LuceneSearcher` (see figure 5.4) implements an interface called `ServiceLifecycle`, which is an interface provided by Apache Axis framework. Any class implementing this interface must have two methods, `init(Object)` and `destroy()`. The method `init(Object)` is called when a session is initialized and `destroy()` is called when the session is terminated. The session mechanism provided by the Axis framework must be used both by the client and server in order for the session to work. If this is the case, a call to the web service will create an instance of the class `LuceneSearcher` and the Axis framework will invoke the method `init(Object)` and access to the index will be set up. When calling the web service again the same instance of the `LuceneSearcher` will be used. This means that the access to the index is already set up and the call can be handled directly, which increases performance. The session will be destroyed if the session is terminated or it times out.

5.4 Website

To access the web service exposing the index, a website has been developed that offers an easy way to enter a search query and overview the search result. The site is built up using Java Server Pages (JSP), which is a technology for creating dynamically generated web pages based on HTML. JSP uses Java programming language embedded in regular HTML which provide a powerful way of building websites. The site also uses JavaScript, to move over processing of data to the client side, and there by relieve the server of this work.

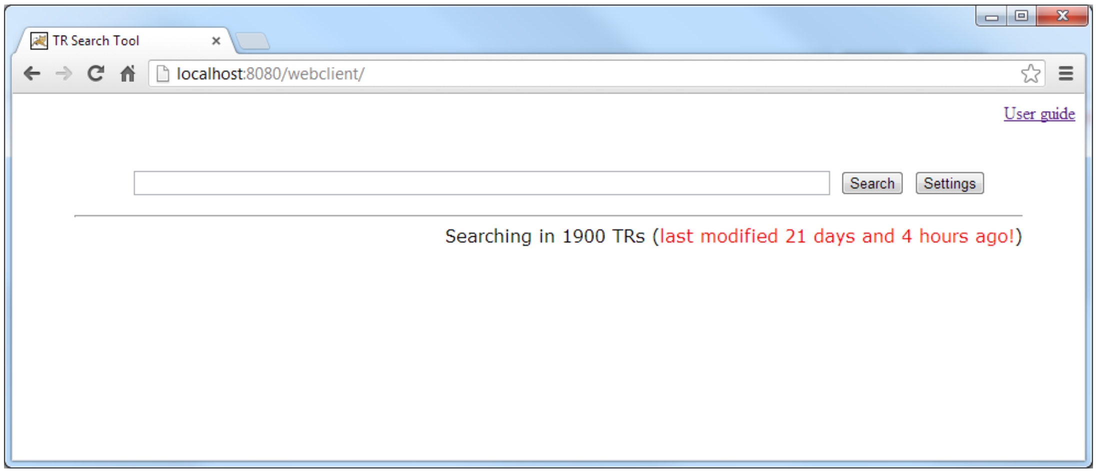
5.4.1 Structure

The site is build up of three JSP pages, `index`, `searcher` and `usage`. The `usage` page contains description of how the search function works and the syntax that can be used when creating a query. The `index` page is the main page and the one used for entering a search query and viewing the search results. The `searcher` page is used to create a session against the web service, invoke the methods provided by the service and then return the result. The `index` page uses JavaScript to create an AJAX (Asynchronous JavaScript and XML) call to the `searcher` page and then dynamically fills the `index` page with the data from those call. By using AJAX to send the query the index page does not need to be reloaded when performing a search and since it is asynchronous the page does not become locked while waiting for the result to come back.

5.4.2 Design

The design of the site is focused on being easy to use and understand. The functionality of the site should be straight forward and be usable by any one.

When entering the main page the user is presented with an input field to enter the query in, a search button to run the query, a settings button for setting special search properties and a link to the `usage` page. There is also information of how many TRs there is in the database and when they were last updated. The main page can be seen in figure 5.5.

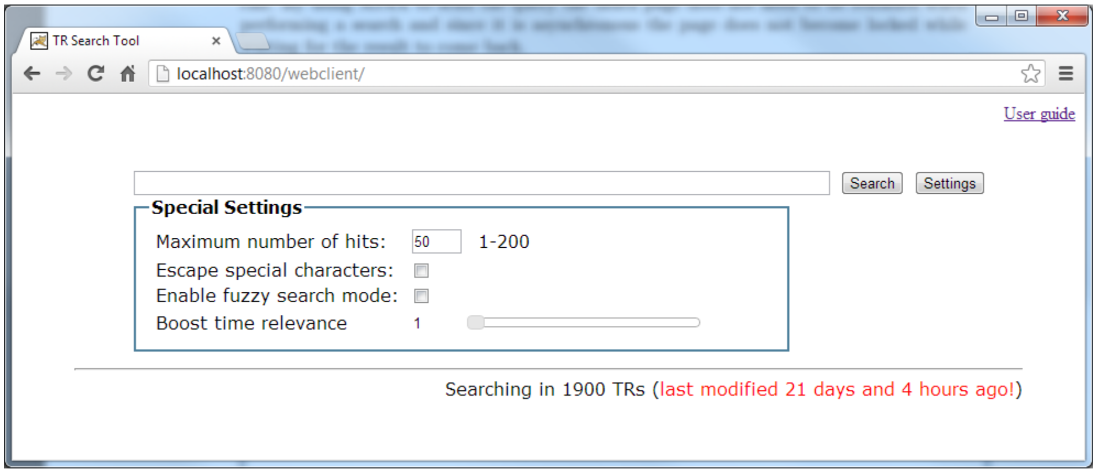


Searching in 1900 TRs (last modified 21 days and 4 hours ago!)

Figure 5.5: Main web page of LTE TR Tool

When clicking the settings button a container will appear where the user can set the search properties that were described under section 5.3.1. This can be seen in figure 5.6.

To search in the database the user enters his/her query in the input field and a hit enter or pushes the search button. This will create an asynchronous call to the web service method `search`. The web service will match the search query to all documents (TRs) in the database and return an array of search hits where each hit contains a small set of the available fields each document has. This data is then put in a table on the main page and


Special Settings

Maximum number of hits: 50 1-200

Escape special characters:

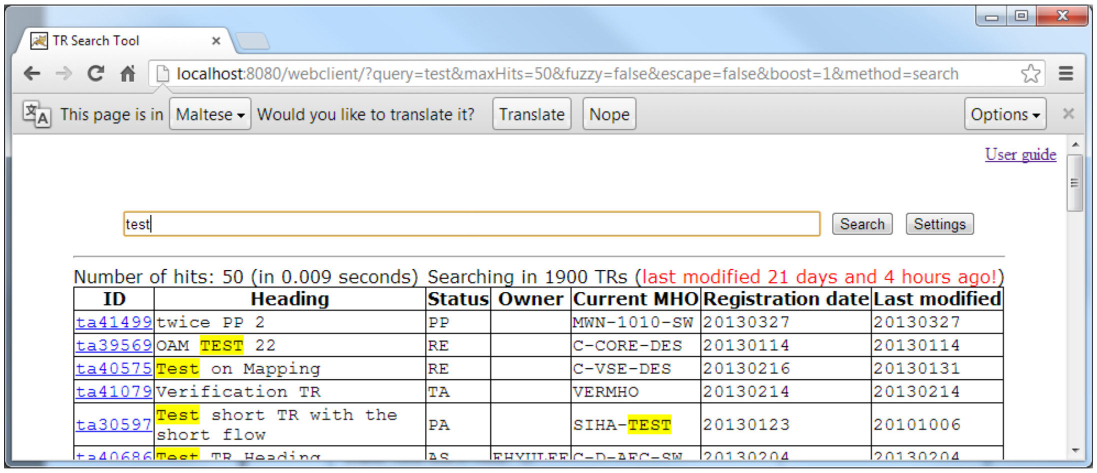
Enable fuzzy search mode:

Boost time relevance 1

Searching in 1900 TRs (last modified 21 days and 4 hours ago!)

Figure 5.6: Settings on main page for LTE TR Tool

the user can see the result. The hits are ordered by relevance to the search query. Figure 5.7 shows a search on the term "test".


localhost:8080/webclient/?query=test&maxHits=50&fuzzy=false&escape=false&boost=1&method=search

This page is in Maltese Would you like to translate it?

test

Number of hits: 50 (in 0.009 seconds) Searching in 1900 TRs (last modified 21 days and 4 hours ago!)

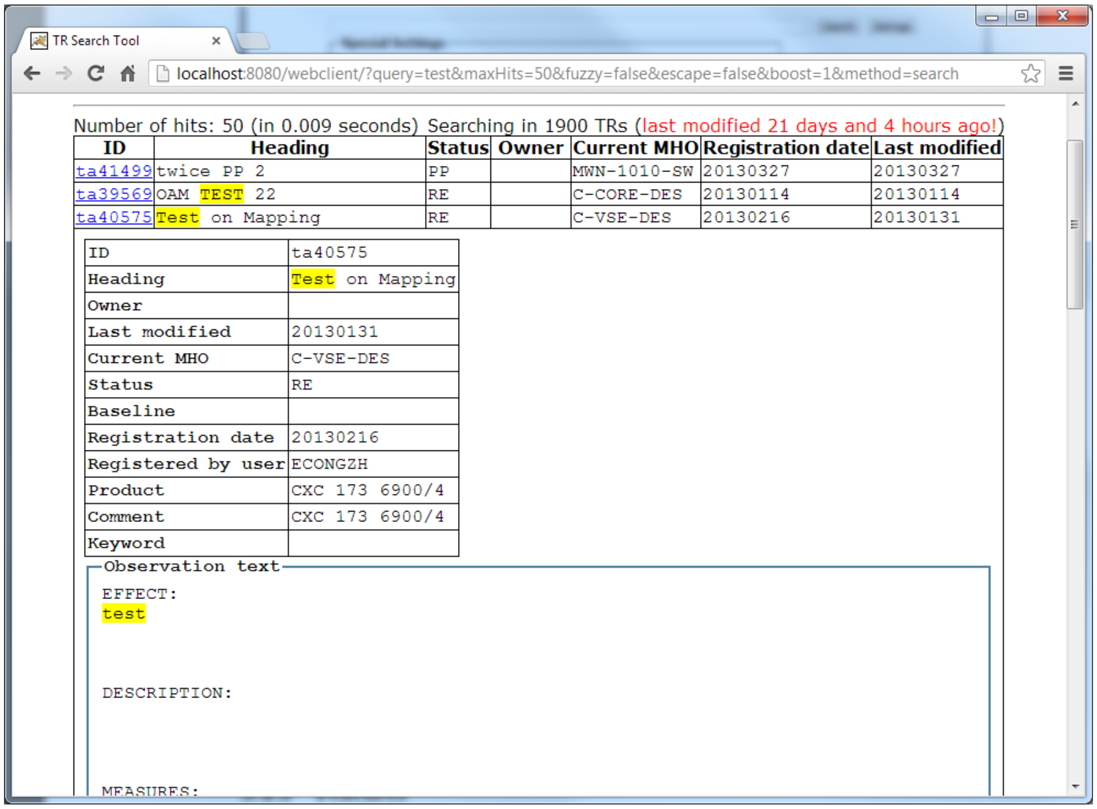
ID	Heading	Status	Owner	Current MHO	Registration date	Last modified
ta41499	twice PP 2	PP		MWN-1010-SW	20130327	20130327
ta39569	OAM TEST 22	RE		C-CORE-DES	20130114	20130114
ta40575	Test on Mapping	RE		C-VSE-DES	20130216	20130131
ta41079	Verification TR	TA		VERMHO	20130214	20130214
ta30597	Test short TR with the short flow	PA		SIHA-TEST	20130123	20101006
ta40686	Test TR Heading	AS	EHYULEE	C-D-AFC-SW	20130204	20130204

Figure 5.7: Search result for the term "test"

Each row in the table represents a hit. Each hit has a link to the original TR at MHWeb. If you click on a row (not the link to MHWeb) a call to the web service method `getTR` will be created. The call will fetch all available fields of the specific TR and show this data right under the clicked row. Figure 5.8 shows how this looks.

5.4.3 Query Syntax

The tool has lots of search features and knowing how to use them can really improve the the information seeking process. In this section we will look at the features related to the query syntax. All these features can be used in the input field on the main page.


TR Search Tool
localhost:8080/webclient/?query=test&maxHits=50&fuzzy=false&escape=false&boost=1&method=search

Number of hits: 50 (in 0.009 seconds) Searching in 1900 TRs (last modified 21 days and 4 hours ago!)

ID	Heading	Status	Owner	Current MHO	Registration date	Last modified
ta41499	twice PP 2	PP		MWN-1010-SW	20130327	20130327
ta39569	OAM TEST 22	RE		C-CORE-DES	20130114	20130114
ta40575	Test on Mapping	RE		C-VSE-DES	20130216	20130131

ID	ta40575
Heading	Test on Mapping
Owner	
Last modified	20130131
Current MHO	C-VSE-DES
Status	RE
Baseline	
Registration date	20130216
Registered by user	ECONGZH
Product	CXC 173 6900/4
Comment	CXC 173 6900/4
Keyword	

```
Observation text
EFFECT:
test


DESCRIPTION:


MEASURES:
```

Figure 5.8: Get all information for one TR

Terms

A query is broken up into terms and operators. there are two types of terms: single terms and phrases. A single term is a single word such as "cake" or "animals". A phrase is a group of words surrounded by double quotes such as "hello world". Multiple terms can be combined together with boolean operators to form a more complex query (boolean operators will be described later).

Fields

The tool supports searches targeted against a specific field. All documents are built up by fields, where each field holds some text. To search for a term in a specific field one writes the field name followed by a colon, ":", and then followed by the search term. For example, the search query "heading:error" will search for the term "error" in the field named "heading".

Wildcard Searches

The search tool supports wildcard searches which means that the users can use a wildcard, denoted with "*", to express a sequence of unknown characters. For example, the query "app*" would return documents containing "apples", "apple", "application" and so on. The tool also supports the use of wildcard limited to just one character, denoted by "?". The query "da?" could return documents containing "dad", "day", "dam" and so on.

Fuzzy Searches

The search tool supports fuzzy searches. To do a fuzzy search one uses the tilde sign, "∼", direct after a single term to indicate that the term should utilize the fuzzy search function provided by the Lucene library. Fuzzy search will search for words that are close to the original search term. For example, the query "goat∼" could return documents containing the term "goats" or "boat".

Proximity Searches

The search tool supports finding words that are within a specific distance from each other. The distance is measured in whole words. To do a proximity search one uses the tilde symbol, "∼", at the end of a phrase, followed by an integer. For example, the query '"data problem"∼5' will get hits where the term "data" and "problem" has a maximum of five words between each other.

Range Searches

Range queries allow one to match documents whose field values are between a lower and upper bound. Range queries can be inclusive or exclusive of the bounds. If the bounds are to be included one use hard brackets, "[]", if the bounds are to be excluded one uses curly brackets, "{}". For example, the query "year:[1999 TO 2005]" will get hits where the field "year" is between "1999" and "2005". Sorting is done lexicographically, which means that one can do range queries not only on numbers but also on words.

Boosting a term

The tool support boosting terms to increase the relevance of individual terms. To boost a term one uses the caret symbol, "ˆ", after a single or phrase term followed by a boost factor. For example, the query "dataˆ2 problem" will boost the relevance of the term "data" by a factor of two.

Boolean operators

Boolean operators allow terms to be combined through logic operators. The tool supports `AND`, "+", `OR`, `NOT` and "-" as boolean operators.

The `OR` operator is the default conjunction operator. This means that if there is no boolean operator between two terms, the `OR` operator is implied. The `OR` operator links two terms and finds a matching document if either of the terms exists in a document. The symbols "||" can be used in place of the word "OR".

The `AND` operator matches documents where both terms exist anywhere in the text of a single document. For example, the query "dog AND cat" will find documents where both terms "dog" and "cat" are present. The symbols "&&" can be used in place of the word "AND". The "+" operator requires that the term after the "+" symbol exists somewhere in a field of a single document.

The `NOT` operator excludes documents that contain the term after "NOT". The sign "-" has the same function.

Grouping

One can group expressions together by putting the expression inside parentheses, "()".

5.4.4 Features

In addition to the search function, the search tool offers some extra features to make the search experience greater. All terms from the query will be highlighted in the result with yellow background. This helps the user to faster see where the hit occurred. When hovering the mouse over a row in the result table the row's background will turn light blue. This will make it easier to read the information in the table. On the page the user can see how many TR the database has and when the database was last modified. If the last modified time was more than 24 hours ago the text will turn red to indicate that something maybe wrong with the connection to MHDB. It is very unlikely that no TR is added or changes for such a long time and emphasizing this can help detect a failure. Another feature is that the time of all searches are measured and displayed to the user. The search time is measured on the server side.

5.5 Hosting LTE TR Tool

Hosting of the web service and the website is done with Apache Tomcat [4] which is an open source web server and servlet container. The data from MHDB is collected using the JMS-client provided by MHWeb and stored in a Lucene index on disk. The location of the web service and website is free of choice. The web service must be able to access the Lucene index and preferable not over network since this will have higher latency compared to being able to access the index direct from the disk where the service is running from.

Both the web service and website project has build scripts that creates a war-file (Web application ARchive) for easy deployment. War-files are easy to add to a Tomcat server, one just copies the files to a base catalogue where Tomcat is running and Tomcat will unpack the archive and make the application available. This makes it easy to deploy the system once a web server is set up.

Chapter 6

Conclusions

In this chapter we will look at the original goals of the project to see if the produced result meets these goals. Some of the desired functionality was not implemented. The reason for this will be examined and motivated. Last we will look at possible future work that could be done.

6.1 Goals

Some of the goals described under section 2.2 are non-functional goals, so to see if they are met requires motivation and reflection. The reason for having non-functional goal was that Tieto did not have clear functional goals and the project's scope were very dynamic. One could say their requirements were low and their expectations were high. So instead of forcing functional requirements in to the goals a more agile approach was taken. A constant discussion were held towards Tieto to ensure that the goals was interpreted in the right way and that the result reached an acceptable level. Table 6.1 shows the reached goals of the project.

6.2 Limitations

One functionality that the system lacks is the ability to match errands automatically. This function was not originally part of the project. It was indented to be implemented if the time permitted it. The reason for not implementing this type of functionality was that there was simply too little time to do it in. To implement a feature of this kind requires lots domain knowledge both in context analysis and in the products the TRs addresses. Determining if two errands originates from the same problem is a hard challenge even for experienced developers and making this task automatic would be very difficult. So instead of implementing such a feature, which would have small changes of being able to produce trustworthy results, the time was spent on make the rest of the system better and more reliable.

6.3 Future work

The next step to ease the process of matching similar errands together would be to look at some techniques for matching errand together automatically. Preferably the web service

Table 6.1: Reached goals in the project

Goal	**Reached**	**Motivation**
Create a system that helps developers match similar errands through free text search.	Yes	The created system has the ability to free text search in errands and can be used to find errands with the help of keyword search.
Make the system configurable to fit the needs of different developing teams.	Yes	The system can be set up and be configured to include all kinds of TRs. The system uses generic interfaces which enables the creation of custom classes if special needs would emerge.
In an easy manner be able to specify which errands should be part of the local database.	Yes	Since the system uses the JMS-client provided by MHWeb, one can easily create rules that dictate which errands should be included in the local database.
Create a mechanism that ensures the local database is up-to-date with the real database.	Yes	By running the JMS-client which regularly checks for changes the local database will be up to date. How often the client checks the queue can be configured to match different requirements.
Free text search in up to 100 000 errands with real time feeling.	Yes/No	The system has not been tested with 100 000 errands so strictly this has not been confirmed. This was not tested because of the load it would create on MHDB, which was unwanted. However the system has been tested with 50 000 errands with very satisfying results and there is no restriction of how many errands the system can handle. The system should be able to handle up to one million errands and still give satisfactory results.
Get useful feedback on the search result.	Yes	The website in the system offers a good overview of the search result. The highlighting function offers extra feedback for the user.
Access of the system should be easy and platform independent.	Yes	Since the search can be done on the website and all code is written in Java all parts are platform independent.
The design of the system should enable further development of the product.	Yes	The usage of Lucene gives the system ability to be customized and extended. The system uses interfaces instead of classes at various locations and there by offers an easy way to replace classes if needed.
Make the system easy to set up.	Yes	The created projects have build scripts and configurations files which makes the deployment of the system fairly easy.

would simply have a method that takes the ID of a TR as input argument and returns a list of possible matches. Exactly how to realize this functionality would require further research.

Another thing that could need some more work in the future is adding more data from each TR in to the index. Each TR carries lots of information an all is not stored in the index.

Only the field significant to matching errand together has been chosen. It is a delicate task of selecting the right ones and there may be a need to complement the index with more data. The reason for not selecting all is that it would create unwanted noise and the precision could be lowered.

Chapter 7

Acknowledgements

I would like to thank my supervisors, Jan-Erik Moström and Daniel Rönström, and all people at Tieto who has taken time to help me and answered my questions. Special thanks go to Daniel Rönstrom whom really has helped me and guided me in this project.

References

[1] Apache. Axis web service. http://axis.apache.org/axis/, 2013. [Online; accessed 2013-04-25].

[2] Apache. Lucene search engine library. http://lucene.apache.org/core/, 2013. [Online; accessed 2013-04-25].

[3] Apache. Solr search server. http://lucene.apache.org/solr/, 2013. [Online; accessed 2013-05-02].

[4] Apache. Tomcat web server. http://tomcat.apache.org/, 2013. [Online; accessed 2013-05-13].

[5] Rick Cattell. Scalable sql and nosql data stores. Technical report, Cattel.Net Software, 2010.

[6] Compass. Compass project. http://www.compass-project.org/, 2013. [Online; accessed 2013-05-02].

[7] T.M. Connolly and C.E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*, chapter The Relational Model. International computer science series. Addison-Wesley Longman, Incorporated, 2005.

[8] D. Crockford. The application/json media type for javascript object notation (json). http://tools.ietf.org/html/rfc4627, 2006. [Online; accessed 2013-05-10].

[9] ElasticSearch. Elasticsearch distributed real-time search engine. http://www.elasticsearch.org/, 2013. [Online; accessed 2013-05-02].

[10] R.A. Elmasri and S.B. Navathe. *Fundamentals of Database Systems [With Access Code]*, chapter Introduction to Information Retrieval and Web Search. ADDISON WESLEY Publishing Company Incorporated, 2010.

[11] A. Goker and J. Davies. *Information Retrieval: Searching in the 21st Century*, chapter Information Retrieval Models. Wiley, 2009.

[12] Google. Google search engine. https://www.google.com/, 2013. [Online; accessed 2013-04-23].

[13] Hibernate. Hibernate search. http://www.hibernate.org/subprojects/search.html, 2013. [Online; accessed 2013-05-02].

[14] IARPA. Intelligence advanced research projects activity website. http://www.iarpa.gov/, 2013. [Online; accessed 2013-04-30].

[15] Yufeng Jing and W. Bruce Croft. An association thesaurus for information retrieval. Technical report, University of Massachusetts at Amherst - Department of Computer Sciences, 1994.

[16] M. MacCandless, E. Hatcher, and O. Gospodnetić. *Lucene in action*, chapter Building a search index. Manning Pubs Co Series. Manning Publications Company, 2010.

[17] Microsoft. Bing search engine. `http://www.bing.com/`, 2013. [Online; accessed 2013-04-23].

[18] MongoDB. Mongodb nosql database. `http://www.mongodb.org/`, 2013. [Online; accessed 2013-05-03].

[19] MySQL. Mysql database. `http://www.mysql.com/`, 2013. [Online; accessed 2013-05-02].

[20] NIST. National institute of standards and technology website. `http://www.nist.gov/`, 2013. [Online; accessed 2013-04-30].

[21] NIST. Text retrieval conference (trec). `http://trec.nist.gov/`, 2013. [Online; accessed 2013-04-30].

[22] U.S National Library of Medicine. Medline medical online library history. `http://www.nlm.nih.gov/news/medline_35th_birthday.html`, 2006. [Online; accessed 2013-04-23].

[23] PostgresSQL. Postgresql database. `http://www.postgresql.org/`, 2013. [Online; accessed 2013-05-03].

[24] Sphinx. Sphinx open source search server. `http://sphinxsearch.com/`, 2013. [Online; accessed 2013-05-03].

[25] Robert S. Taylor. The process of asking questions. `http://zaphod.mindlab.umd.edu/docSeminar/pdfs/16863553.pdf`, 1962. [Online; accessed 2013-04-23].

[26] W3C. World wide web consortium soap. `http://www.w3.org/TR/soap/`, 2004. [Online; accessed 2013-04-25].

[27] Wikipedia. B-tree — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/B%2B_tree`, 2013. [Online; accessed 2013-04-24].

[28] Wikipedia. Bayes' theorem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Bayes'_theorem`, 2013. [Online; accessed 2013-04-24].

[29] Wikipedia. Cap theorem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/CAP_theorem`, 2013. [Online; accessed 2013-05-02].

[30] Wikipedia. F1 score — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/F1_score`, 2013. [Online; accessed 2013-04-30].

[31] Wikipedia. Levenshtein distance — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Levenshtein_distance`, 2013. [Online; accessed 2013-05-05].

[32] Wikipedia. Stemming — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Stemming`, 2013. [Online; accessed 2013-04-24].

[33] I. Xie. *Interactive information retrieval in digital environments*, chapter User-Oriented IR Research Approaches. NetLibrary, Inc. IGI Pub., 2008.

[34] P. Zaitsev, V. Tkachenko, J.D. Zawodny, A. Lentz, and D.J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and More*, chapter Advanced MySQL Features. O'Reilly Media, 2008.

[35] P. Zaitsev, V. Tkachenko, J.D. Zawodny, A. Lentz, and D.J. Balling. *High Performance MySQL: Optimization, Backups, Replication, and More*, chapter Schema Optimization and Indexing. O'Reilly Media, 2008.