Indexing and Search Algorithms for Web shops
Indexering och sök algoritmer för webshoppar

ISAK GUSTAFSSON
AXEL REIMERS
Indexing and Search Algorithms for Web shops

Indexering och sök algoritmer för webshoppar

Isak Gustafsson och Axel Reimers
Abstract

Web shops today need to be more and more responsive, where one part of this responsiveness is fast product searches. One way of getting faster searches are by searching against an index instead of directly against a database.

Network Expertise Sweden AB (Net Exp) wants to explore different methods of implementing an index in their future web shop, building upon the open-source web shop platform SmartStore.NET. Since SmartStore.NET does all of its searches directly against its database, it will not scale well and will wear more on the database. The aim was therefore to find different solutions to offload the database by using an index instead.

A prototype that retrieved products from a database and made them searchable through an index was developed, evaluated and implemented. The prototype indexed the data with an inverted index algorithm, and was made searchable with a search algorithm that mixed type boolean queries with normal queries.

Keywords
Index, Data Mining, Knowledge Discovery in Databases, Lucene.NET, Inverted Index
Sammanfattning

Webbutiker idag behöver vara mer och mer responsiva, en del av denna responsivitet är snabb produkt sökningar. Ett sätt att skaffa snabbar sökningar är genom att söka mot ett index istället för att söka direkt mot en databas.

Network Expertise Sweden AB vill utforska olika metoder för att implementera ett index i deras framtida webbutik, byggt ovanpå SmartStore.NET som är öppen käll-kod. Då SmartStore.NET gör alla av sina sökningar direkt mot sin databas, kommer den inte att skala bra och kommer slita mer på databasen. Målsättningen var därför att hitta olika lösningar som avlastar databasen genom att använda ett index istället.

En prototyp som hämtade produkter från en databas och gjorde dom sökbara genom ett index var utvecklad, utvärderad och implementerad. Prototypen indexerade datan med en inverterad indexerings algoritm, och gjordes sökbara med en sök algoritm som blandar boolska frågor med normala frågor.

Nyckelord
Index, data utvinning, kunskaps upptäckelse i databaser, Lucene.NET, inverterad index
Acknowledgements

We would like to give our thanks to Bo Wisenfelt and Robert Andersson from the company Network Expertise Sweden AB for providing us with things that we have needed, answers and for allowing us to carry out this thesis.
# Table of contents

1 Introduction .......................................................................................................................... 1
   1.1 Problem statement ........................................................................................................ 1
   1.2 Goals ............................................................................................................................. 1
   1.3 Delimitations ................................................................................................................ 2
   1.4 Statement of authorship ............................................................................................... 2

2 Theory and Background ....................................................................................................... 3
   2.1 The construction of a search engine ........................................................................... 3
   2.2 Knowledge discovery in databases ............................................................................ 4
   2.3 Indexing ....................................................................................................................... 4
      2.3.1 Inverted index ....................................................................................................... 5
      2.3.2 Suffix array .......................................................................................................... 6
      2.3.3 N-gram index ...................................................................................................... 8
      2.3.4 Lucene.NET ......................................................................................................... 8
      2.3.5 Evaluate ............................................................................................................... 9
   2.4 Store index ................................................................................................................... 9
   2.5 Update index ............................................................................................................... 9
   2.6 Searching ..................................................................................................................... 9
      2.6.1 Suffix array binary search .................................................................................. 9
      2.6.2 Boolean query ..................................................................................................... 10
      2.6.3 Normal query ..................................................................................................... 10
      2.6.4 Vector model ....................................................................................................... 10
      2.6.5 Evaluate ............................................................................................................... 10

3 Methods and Results ......................................................................................................... 13
   3.1 Indexing ....................................................................................................................... 13
      3.1.1 Inverted index .................................................................................................... 13
      3.1.2 Hash-table ......................................................................................................... 14
      3.1.3 Alternative methods for indexing ..................................................................... 14
   3.2 Search .......................................................................................................................... 15
      3.2.1 Vector model and Boolean query ....................................................................... 15
      3.2.2 Alternative methods for searching .................................................................... 15
3.3 Implementation ................................................................. 16
  3.3.1 API library .................................................................. 16
  3.3.2 ASP.NET Server .......................................................... 18
3.4 Tests ................................................................................. 18
  3.4.1 Test on queries in different formats ................................ 18
  3.4.2 Tests on how fast a query result was returned ............... 19
  3.4.3 Test on storage size ...................................................... 19
  3.4.4 Generating random documents ...................................... 19
  3.4.5 Hardware .................................................................... 19
3.5 Compilation of results ....................................................... 19
  3.5.1 Format on queries ....................................................... 20
  3.5.2 Comparison of returned result times ............................. 20
4 Analysis and discussion ....................................................... 25
  4.1 Difference in results ....................................................... 25
  4.2 Suffix and postfix ........................................................... 25
  4.3 Managing database changes .......................................... 25
  4.4 Storing index to files ...................................................... 26
  4.5 Social and economic aspects ......................................... 26
  4.6 Environmental aspects .................................................. 26
5 Conclusion ........................................................................... 27
References ............................................................................... 29
Appendix ................................................................................. 32
1 Knowledge Discovery in Databases .................................... 32
  1.1 Cleaning ......................................................................... 32
    1.1.1 Regression technique ................................................ 33
    1.1.2 Clustering technique .................................................. 33
    1.1.3 Binning technique ...................................................... 34
  1.2 Integration ........................................................................ 34
  1.3 Transformation ............................................................. 35
  1.4 Data mining ..................................................................... 36
    1.4.1 Models ..................................................................... 36
    1.4.2 Classification ............................................................ 36
<table>
<thead>
<tr>
<th>1.4.3</th>
<th>Relational databases</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.4</td>
<td>Multiple data sources</td>
<td>37</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Problem statement
Searching for products on web shopping sites is a common thing to do nowadays on the Internet. The products are stored in a database, and each site often provides their own search engine to enable customers to search for products.

If there are multiple customers browsing the web shop at once and if the web shop also contains a lot of products, then it can decrease the response time when the customers are browsing it and make the customers shopping experience worse. Delivering a bad shopping experience can lead to lost sales. This can also lead to the worst case scenarios that the database cannot deal with the pressure and crash, and be at a security risk since a denial of service attack can be made by making too many calls to the database to make the database unreachable. A solution to this problem is to create an index from the database and send all searches to the index instead.

An index is a data structure that is constructed to improve the speed of data retrieval operations of a database, and is used to locate data without having to search through every row in a database table. An index can also be used to together with a proxy to further split the load of customers searching the web shop onto multiple indexes.

1.2 Goals
The goal is to research and develop a server with an outward HTTP interface that can index databases and make them searchable along metadata, attributes and text for a web shop. The solution should from the company's side preferably be based on open-source algorithms and data structures, be scalable (how many products that can be indexed), give quick search results and not contain functionality that is not needed for this task. The index should preferably support the functionality of being kept updated if changes occurs to the database. This task was divided into the following goals:

1. Research and document the subject indexes.
   a. Research and document different indexing and search algorithms for indexes.
   b. Research and document advantages and disadvantages of different indexing and search algorithms based on requirements from Net Exp.
2. Research and document how to keep track of changes in a database.
3. Develop a server application prototype.
   a. Implement the indexing and searching algorithms that were considered most fitting from the research.
   b. Implement the method for keeping track of database changes that were considered easiest to maintain.
4. Evaluate the prototype.
   a. Performance test on the prototype.
   b. Storage space test on the prototype.
   c. Swedish ISO and special character test on the prototype.
   d. Compare the performance and storage results to an existing solution.
1.3 Delimitations
The thesis work:

- Were to be developed in .NET with C#.
- A MS SQL database were to be the main source to use and gather data from externally.
- Were to only use one database.
- Would only be searchable with attributes and regular text.
- Would not be to design databases for web shopping sites.
- Would be a smaller part of a bigger project, where it would only touch the search and index functionality of the web shop.

1.4 Statement of authorship
The writing of this thesis, including the results from the development of the prototype, have been undertaken as a mutual collaboration by Axel Reimers and Isak Gustafsson.
2 Theory and Background

This chapter presents the fundamental information on how a search engine works, together with indexing and searching an index. A number of algorithms and data structures that can be used to achieve this are examined and documented. It also describes how web search engines work which has similarities to the subject of this thesis. Every subchapter will contain general as well as technical information on these techniques and technologies.

Algorithms are what lets computers solve a problem automatically. An algorithm can be described as a solution method that can take many forms. They are often underlying based on mathematics, but depending on the problem there are algorithms that are not directly connected to mathematics.

A search engine is a tool on the web or in an application that exists to help users find what they are looking for by entering search terms. A web search engine usually searches through web pages online to find its results, but can also use other data sources such as databases. The results that are returned to the user are called hits and is usually sorted in a list before presented.

2.1 The construction of a search engine

According to the creators of google Sergey Brin and Lawrence Page, search engines today work primarily in three steps [1]:

- Web crawling, gathering of data
- Indexing, structuring of data
- Searching, getting results from the index

In their study “The Anatomy of a Large-Scale Hypertextual Web Search Engine” [1] they present an in-depth description of how large-scale web search engines work, and how a practical large-scale system can be built. It also touches on problems that arises when dealing with the web which is a large collection of uncontrolled heterogeneous documents, and not a well-controlled collection like a database. If a database is used as the source collection the information is not gathered and added by a crawler, the information is instead added to the database by a user. In this case, the focus is more directed towards the indexing of the information that resides in the database.

To create a prototype, Sergey Brin and Lawrence Page put up a few design goals which were to:

- Improve the search quality of search engines.
- Build a system that a reasonable number of people could actually use.
- Use and architecture that could support novel research activities on large-scale web data.

In their prototype, the web was data mined by a set of distributed web crawlers where the retrieved information was then parsed into documents, each with a separate document ID. Documents were then converted into a set of word occurrences called hits, which records
the word, position in document, and information like approximation of font size and capitalization. The hits are then stored into a set of storage areas called barrels by the indexer, which generates a forward index partially sorted by document ID. It then creates an inverted index to be able to return query results quickly by making the indexer take the barrels and resort them by word ID, it also uses the list of word IDs to create a lexicon that a searcher can use together with the inverted index to answer queries.

The results that the finished prototype returned were better than other major search engines that existed at the time the study was conducted. It returned high quality pages and none of them were broken links, which was one of the major design goals of the prototype. The performance of the system was shown to be very efficient by managing to mine 26 million pages in roughly 9 days at first start, with the current hardware at the time. Once the system was running smoothly the speed of the mining process actually increased mining 11 million pages in just 63 hours, which was about 48.5 pages per second. The indexer ran about 54 pages per second, with the sorters running in parallel who finished their jobs in about 24 hours. Queries were answered in between 1 and 10 seconds in the prototype version at the time.

2.2 Knowledge discovery in databases
Knowledge Discovery in Databases or KDD for short is the process of discovering useful information out of a set of data from one or multiple databases. KDD can be separated into different phases that needs to be carried out in order. These different phases can for some applications be very simple and for others more complex. The phases are in the following order [2, 3, 4]:

- **Cleaning**, for removing noise and inconsistent data from sources,
- **Integration**, for merging multiple sources to one,
- **Selection**, for choosing which data to be used,
- **Transformation**, for combining or changing the form of data,
- **Data mining**, for finding patterns in the data and lastly,
- **Presentation**, for presenting the result in a way that can be understood by humans.

KDD is used in a wide variety of different business areas like scientific research in astronomy and medicine, government security for combating crime and terrorism, and in financial business like banking and retail [5, 6]. In the case of this thesis KDD was studied, but ruled out since the data from the sources were not inconsistent and no intricate patterns had to be found. The main focus of this thesis was the indexing and searching algorithms of the server. To read more about the KDD study, please refer to appendix 1.

2.3 Indexing
Indexes are used for optimizing speed and performance in finding relevant documents for a search query. Documents is the entity that contains all the information that will be saved inside the index. The information is represented by individual fields that works much like columns in a relational database. A field can contain different kinds of data, such as strings, integers, booleans and so on. All fields inside of a document are not required to be searchable and can also be stored as is for easy retrieval. When creating a document, the developer needs to divide the raw content of files into documents and fields. An email for example can be expressed as one document, where the sender, receiver, topic and body can each be a
separate field in the document. You could of course sequentially scan each file for the given word or phrase. This would however be very inefficient since it doesn’t scale to larger file sets or where files are generally very large. To be able to search large amounts of text quickly, one must first index the contents of the files and convert it to a format that allows rapid searching, instead of using a slow sequential scanning technique. Indexes can be seen as a form of data structure that are stored as index files on the file system and allows for fast random access to content of words that are stored inside [7].

Depending on the design factors and architecture of a system, the way indexing is performed and how it is stored vary. Some implementations use existing API libraries such as Lucene, others create their own solutions with other data structures. Various indexes do certain things better than others, because of this they are often used in different areas and not just in search engines, for example bioinformatics. A couple of these will be explained in this section. Some of the index methods that are used today are [8, 9]:

- Inverted indexes
- Suffix arrays
- N-gram indexes

2.3.1 Inverted index

An inverted index is a modern approach for search engines, which can be combined with different data structures to tailor it for the needs presented. [10] In an inverted index there exists a lexicon containing all words used in all of the documents within the index. Each word listed in the index have a list pointing to all documents that mentions it. Just storing the documents who mentions a word can be used for normal queries. To make more sophisticated queries such as phrase and proximity searches also require positional data of words in documents. Depending on what type of queries should be supported, the positional data can for example be the position in the document or the frequency of a given word in a document. [11, 12]

If a string S1 = “Carrots should be orange, and bananas should be yellow” with id 1 is to be indexed, then an inverted index containing it will look like table 1 if word frequency should be the only additional information in the index. The list of ids in the index is of the form \((d_1:f_1, d_2:f_2, \ldots, d_n:f_n)\) where \(d_i\) is the document id and \(f_i\) is the frequency of the given word. Word frequency can help create better rankings in the search results.
Table 1: String S1

<table>
<thead>
<tr>
<th>Word</th>
<th>List of ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carrots</td>
<td>1:1</td>
</tr>
<tr>
<td>Should</td>
<td>1:2</td>
</tr>
<tr>
<td>Be</td>
<td>1:2</td>
</tr>
<tr>
<td>Orange</td>
<td>1:1</td>
</tr>
<tr>
<td>And</td>
<td>1:1</td>
</tr>
<tr>
<td>Bananas</td>
<td>1:1</td>
</tr>
<tr>
<td>yellow</td>
<td>1:1</td>
</tr>
</tbody>
</table>

The lexicon in an inverted index can be constructed with numerous different data structures, such as hash tables or trees. With a hash table insert and search operations would have complexity around O(1). [13] This enables finding documents by value more quickly.

2.3.2 Suffix array

The suffix array is a data structure that can be used in full text indexes and data compression algorithms. It was introduced by Udi Manabers and Gene Myers in 1990 to be used as an alternative to the suffix tree data structure. According to the creators [14], it enables online string searches like “Is W a substring of S?” in O(P + logN) time, where P is the length of W and N is the length of S, and can be constructed in O(N log N) time. It works as an array that takes, sorts and stores all suffixes of A, and uses information about the longest common prefixes of adjacent elements to conduct string searches. For example, to index the string $S = \text{banana}\$, each character will have an own index in the string array as shown in table 2:

Table 2: String S.

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>S[i]</td>
<td>b</td>
<td>a</td>
<td>n</td>
<td>a</td>
<td>n</td>
<td>a</td>
<td>$</td>
</tr>
</tbody>
</table>

Where the character $ is unique and lexicographically smaller than the other characters in S. S will have the following suffixes as shown in table 3:
Table 3: The suffixes of string S.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>banana$</td>
<td>1</td>
</tr>
<tr>
<td>ana$</td>
<td>2</td>
</tr>
<tr>
<td>nana$</td>
<td>3</td>
</tr>
<tr>
<td>ana$</td>
<td>4</td>
</tr>
<tr>
<td>na$</td>
<td>5</td>
</tr>
<tr>
<td>a$</td>
<td>6</td>
</tr>
<tr>
<td>$</td>
<td>7</td>
</tr>
</tbody>
</table>

These suffixes are sorted in ascending order as shown in table 4:

Table 4: The sorted suffixes of string S.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>7</td>
</tr>
<tr>
<td>a$</td>
<td>6</td>
</tr>
<tr>
<td>ana$</td>
<td>4</td>
</tr>
<tr>
<td>anana$</td>
<td>2</td>
</tr>
<tr>
<td>banana$</td>
<td>1</td>
</tr>
<tr>
<td>na$</td>
<td>5</td>
</tr>
<tr>
<td>nana$</td>
<td>3</td>
</tr>
</tbody>
</table>

A suffix array A is created to store the starting position of each sorted suffix as shown in table 5:
Table 5: Suffix array A with indexes and suffixes shown vertically for better understanding.

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>$</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>$</td>
<td>n</td>
<td>n</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>a</td>
<td>n</td>
<td>$</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$</td>
<td>n</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>n</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

According to its creators the suffix array stores n integers [14]. Assuming an integer requires 4 bytes means that the suffix array requires 4n bytes.

2.3.3 N-gram index
N-gram models are widely used in statistical natural language processing and probability, but is also a popular index for search applications where phrases are used, such as when doing plagiarism detection on documents. Documents are described using a set of n-grams, one n-gram is a sequence of n words. Creating a set is done by moving a window of size n from the beginning to the end of a document, where each location for the window gives one n-gram. Before n-grams are created they are processed with stemming algorithms and more, which decreases the number of words in the n-grams. The index is created by creating tuples of the n-grams of the format \{n-gram, id, frequency\}, the n-gram acts as the key of the tuple and all tuples are contained in a chosen data structure. The chosen data structure also gives roughly the time for searching the index. [15]

2.3.4 Lucene.NET
Lucene is an open source search engine library for doing full-text searches, Lucene.Net is a port of the original Java-based library to .Net and C# [16]. The core parts of the Lucene library are its analysis of acquired content, indexing and storing the content, and lastly searching. Lucene.NET uses an inverted index as its underlying index data structure, which is populated by tokens. Before the inverted index is filled with tokens, the tokens first have to be created by analyzing the documents. A token is an atomic element that correlates roughly to a word. It is used since an index cannot index text directly, the textual fields that exists in a document has to be divided into separate tokens first. These tokens are then used to create query representations. The analyzation is done in three steps, filtering of optional characters and normalize the content, create tokens and lastly filter the tokens [17].
2.3.5 Evaluate
Today, most search engines and search applications are using inverted indexes due to the popularity it has gained from the Apache Lucene Library \[18\], and its ability to find documents by value very quickly. For an index being used in a web shop, the source of information is predetermined and should be structured in the index such that it can easily be matched against each search term separately. This also means that the order of the terms appearance in the source is not of importance.

When it comes to finding patterns of analytical purpose in strings, suffix arrays really exceed since they store all the suffixes of the words they are given, which can really benefit features like auto-completion. N-gram index is most useful when transcending large amounts of text and the query terms must be matched in the correct order.

2.4 Store index
To be able to persist the state of the index, it has to be retained to some computer data storage or memory. Almost all computers use a hierarchy of storage consisting of mainly primary storage and secondary storage \[19\]. The primary storage contains RAM (random access memory) which is directly accessible by the CPU by a memory bus, this increases the bandwidth and decreases the latency when being accessed. The RAM that is used for primary storage is a volatile component where its memory is uninitialized at startup. The secondary storage is on the contrary non-volatile, and does not lose stored data when powered down. It is not directly connected to the CPU, which in turn decreases bandwidth and increases the latency when reading and writing data, but is more cost-efficient since it can store a large amount data at a lesser cost per byte compared to primary memory \[19\].

2.5 Update index
In Microsoft SQL Server there exists built in features for monitoring changes in chosen database tables. Change Tracking is a feature that can answer an application what rows in a table has changed or if a row has changed, it does not save how many times a row has been changed only that a change has occurred and those changes can be retrieved from the table concerned directly. Change Tracking gives a lightweight option for monitoring changes in chosen tables with great flexibility on what should be saved with the tracking \[20\].

2.6 Searching
Gathering information from a database is accomplished by using regular SQL-queries against a database. Querying an index is however a lot more different and depending on the underlying data structure often needs an algorithm for retrieving data. Depending on how results should be presented, in group or single exact matches, the developer can make use of normal, Boolean queries and binary searches. This section will present and evaluate different search and query techniques that can be used together with the algorithms and data structures explained earlier in 2.3.

2.6.1 Suffix array binary search
The common way to search a suffix array is by using the binary search algorithm, by using binary search the complexity of searching the suffix array is $O(\log( |T|))$ where $|T|$ is the size of the suffix array. Binary search can be used because the suffix array is sorted and therefore grouping together suffixes with the same prefix. The result of a query will then be the suffixes between two positions in the array. \[9\]
Binary search searches a set by dividing the search array repeatedly until the correct prefix is found. At every iteration it compares the middle of the set with the query, the compare is a simple bigger than, smaller than or equal to comparison. If the bigger than comparison is true it will continue with the half with higher value in the next iteration, the smaller than comparison continues with the other half and if the equal to comparison is true then it has found the prefix. Two rounds of binary search are necessary to find the interval for the prefix being searched for. [21]

2.6.2 Boolean query
Boolean queries use operators from Boolean logic to construct Boolean expressions with the search terms. An OR operator can be used to find all documents containing at least one of the search terms, while an AND operator finds all documents containing all of the search terms. The OR operator cannot be used by itself and must therefore be used together with OR and AND operators. By combining these different operators, a customized logic can be built for the query available to the user. [11]

2.6.3 Normal query
A normal query is a clear method to query multiple documents with a single term. It is not explicitly indicated to be a specialized query like the boolean query described in 2.5.2. When single term queries are made it matches all documents that contains that specific term. However, it becomes quite unclear when matching documents with multi-word queries occur, and for example treats it as an implicit boolean query by inserting AND operators between each term. This can however make the query too strict and not include results where some terms are missing even though their relevance is very high. Because of this, other implementations use result ranking which allows some terms to be missing in the query and then ranks the result by how many of the search terms were found in each document. [11]

2.6.4 Vector model
The vector model is a method of searching an inverted index by calculating a score of the similarity between two weights, one weight for the query and one for the document. The score is then utilized when processing the query. When processing the query, an array with one accumulator for every document is created. Each term is searched separately, and for each term the list of documents is retrieved. The similarity score is then calculated and added to the accumulator. After every term has been gone through, the documents with the highest accumulated score are retrieved. [22]

2.6.5 Evaluate
Search engines for web shops does not require phrase search capabilities due to search terms consisting of adjectives and nouns from products, for example “Intel Core i7-6700K 8 threads”. It should also be able to display multiple results that are relevant to the user’s search query. Binary searches are usually not used in search engines since they only provide a user with an exact match to their search query. Boolean queries are a more popular choice for search engines since they are very customizable, though they can miss search results if they are not sophisticated enough. If the AND operator is used and a user searches with three terms and only two or less of the search terms actually exist in the result the user wants, then that result can be left out and considered not relevant. To make it smarter it can
be combined with normal queries which specializes in matching a search term with multiple documents, and a result ranking method to present a sorted list of more relevant results. Vector Model is another choice that is easier to implement than boolean queries because of not using complex boolean expressions, but it requires more pre-processing of documents since the weights used to get a similarity score must be calculated. The vector model also ranks the results and can be used to easily retrieve the best matching results. Because of creating a list of accumulators for all documents in the index, it does require more memory when a query is made.
3 Methods and Results

This chapter describes the methods and algorithms that were used to develop a prototype that can index databases and make them searchable for a web shop. Discussions about alternative methods and algorithms are also described.

The solution should from the company's side preferably be based on open-source algorithms and data structures, be scalable (how many products that can be indexed), give quick search results and not contain functionality that is not needed for this task.

The research in this thesis was carried out using books and literature studies from Google Scholar and KTH primo, together with product websites and source code from developer websites. A number of algorithms were researched and compared against each other. From this analysis, the algorithms considered most fitting towards the needs of the company explained in 1.2 were chosen. A prototype using the chosen algorithms was then developed in the form of a API library. The prototype was then tested after it had been filled with dummy data to evaluate the time it takes for a response to a query to arrive, storage space and different query format.

3.1 Indexing

Since creating testbeds and environments to test each algorithm and data structure would take too much time. To make a decision on what indexing algorithm and data structure to use, a lot of the research done was studying studies made by others, and what other big applications like Google [1], Lucene [18] and other implementations were using, how they were using them and why. After evaluating the different indexing algorithms against each other based on studies, their inner workings and functionality, a solution that used an inverted index combined with a hash-table as the underlying data structure was considered the best alternative for a problem such as this.

3.1.1 Inverted index

Inverted indexes are used by other big applications because of its easy implementation, high customizability for result ranking and its ability to search for words without scanning through whole documents which decreases both the search area and disc access. It is also very applicable in distributed applications. Inverted index construction is straightforward where text must be processed in document number order into a data structure that can be written to disc in term order. The only constraint is that it must have a reasonable amount of memory available. When it comes to scalability the inverted index is also very prominent. The amount of disk accesses for the inverted file index is independent of scale, so the processing cost is the amount of data that can actually be transferred from the index. It is also in favor when it comes to extensibility, it supports a wide range of text query types and is fast at handling them since it only requires a fraction of the number of disk accesses. All of this concluded in the studies "Distribuerte Inverterte Indekser" by Simon Jonassen [22], and "Inverted Files Versus Signature Files for Text Indexing" by Justin Zobel, Alistair Moffat and Kotagiri Ramamo-hanarao [23]. Where the authors implemented the inverted index in a database system which performed very well compared to other methods, since the current trend today in computer technology where the difference in processor speed compared to disc access time is increasing.
3.1.2 Hash-table
The reason why a hash-table was chosen as the underlying data structure was because it uses direct argument. Explained in the book “Introduction to Algorithms” [13], a hash table utilizes the $O(1)$ speed of an normal array’s direct addressing to get quick lookup times. According to the book, each value in a hash-table have an associated key, an index in the array for the given key is calculated with a hash function. Each index is usually called a bucket and there can be occasions when more than one key leads to the same bucket with the hash function, while ideally there should only be one value per bucket.

Another reason was that there exists a fast and reliable implementation of a generic hash-table in C# .NET with concurrency features called ConcurrentDictionary [24] which was used as the inner data structure of the index. This data structure was made for multiple read and write actions simultaneously, where multiple reads simultaneously are necessary in a web shop.

3.1.3 Alternative methods for indexing
In this section the alternative methods for indexing are presented. These methods were not chosen since they proved to achieve different goals than the ones set for this thesis, but they can however be used to build upon the chosen index algorithm. Since the requirements of the company was that the system should answer queries fast and be lightweight, meaning that it would only focus on easily matching search terms with the documents stored inside the index, the other algorithms were not chosen since they had features that were deemed unnecessary and abundant.

3.1.3.1 Suffix array
Suffix arrays enables string matching but is more useful for finding patterns and suffixes of strings for analytical purposes, rather than just matching strings [9]. Keeping track of the suffixes of every indexed word would ease the implementation of auto-completion, which would require keeping track of and storing a lot of additional data being the substrings of every stored string [25]. But this was not a demand from the company so it was not chosen for implementation in the final prototype.

3.1.3.2 N-gram index
N-grams are applied in search applications, but its focus is largely on phrases, for example in plagiarism detection [15]. In the study about n-gram search engines by Satoshi Sekine [26], the study explains that n-grams is mainly used for NLP (Natural Language Processing) applications where the source data need a lot of pre-processing before being indexed. And that this also gives N-gram search engines great flexibility in how searches are made, with the ability to use wildcards for example. But since web shops do not have a need for extra flexibility such as wild cards, it was deemed to be abundant and was not implemented in the final prototype. The study also explains that a lot of preprocessing computation also makes the build time for an N-gram index slow, and since search engines also needs to be rebuilt frequently in a short amount of time, it was also deemed not good enough for this application.

3.1.3.3 Lucene.NET
Lucene.NET is a ready to use API library with finished functions for indexing and searching, as well as spellchecking and tokenization capabilities ready to use in a .NET environ-
The homepage of Lucene.NET explains that it uses an inverted index as its underlying data structure for indexing its documents. Being a ready to use API library, it was not possible to use in our solution, since the purpose of this thesis was to research and develop our own solution to the stated problem using open source algorithms.

### 3.2 Search

The algorithm used for searching in this thesis was created by basing it on other known and used algorithms, to fit the company’s needs. When researching and documenting different algorithms based on studies and their functionality, the most important properties of the algorithms to take into account were: as low complexity as possible, be customizable, support ranked results and multi-term queries.

After evaluating the different search algorithms against each other based on their inner workings functionality, a solution based on the combination of the vector model and the boolean query was considered the best alternative for the problem handled in this thesis.

#### 3.2.1 Vector model and Boolean query

In the study “Inverted indexes: Types and techniques” by Ajit Kumar Ma hapatra and Sitanath Biswas, it is explained that Boolean queries are great for indicating if a search term exists or not in a document and is a good strategy to use when implementing term frequency, but using it alone has the potentiality of expanding or shrinking the result set too much.

The vector model is explained in the study "Distribuerte Inverterte Indekser" by Simon Jonassen as simple to implement and very flexible. The flexibility comes from the weighting functions used that can be customized after what is desired.

By combining the weights of boolean queries with the vector model it is concluded that all terms are of equal value and can give good results, as stated in the paper “A Statistical Approach to Mechanized Encoding and Searching of Literary Information” by Hans Peter Luhn that the frequency of a term is relative to its importance.

#### 3.2.2 Alternative methods for searching

In this section the alternative methods for searching the index are presented. These methods were not chosen since they proved to achieve different goals than the ones set for this thesis or not meeting all the properties desired in the resulting search algorithm.

##### 3.2.2.1 Binary search

As stated by the National Institute of Standards and Technology of how the binary search algorithm works, binary search only matches the exact phrase searched for, this would add extra complexity for the query since multi-term queries were a requirement, which in turn also adds extra complexity for achieving ranked results. It also requires a sorted list as source since the binary search creates smaller and smaller search areas at each iteration by finding the center of the search area.

##### 3.2.2.2 Normal query

According to the study “Inverted indexes: Types and techniques” by Ajit Kumar Ma hapatra and Sitanath Biswas, normal queries are great when using single term queries
when searching for documents with a specific term, but was not chosen for this thesis because the study also mentions that there is no clear defined way of how to handle multi-term queries and ranked results in normal queries.

3.3 Implementation
In this subchapter, the implementation of the prototype is described. The prototype was divided into two parts. One being a API library, and the other a ASP.NET project that uses the separately developed API library. This split made unit testing easier and also gave the API library the opportunity to continue being developed and used by others after this thesis was finished without being tied to a specific environment.

3.3.1 API library
An API stands for “Application Programming Interface” and contains a set of tools for building software and applications. It works as an interface related to a software library with a set of classes and methods written in one language, but can be used in other languages too.

The main part of the API library developed in this thesis is its indexing and searching functionality. The underlying data structures utilized in the API library when indexing and searching is made up of two hash-tables, one used for the lexicon and one that contains all the documents in the index with a unique string key as an identifier. The hash-table used by the lexicon uses terms as keys and the values are lists of all documents searchable by that term represented by their respective key.

3.3.1.1 Indexing
The data that will be added to the index needs to be preprocessed and packaged into the correct structure before the API is called. To achieve the correct structure, data that are going to be searchable containing more than one word separated by blank spaces are divided into individual terms. The data that are not going to be searchable is added to a hash-table as a string value together with a string key as the identifier.

When a document is added to the index, a unique key chosen by the programmer together with the string of terms that are going to be searchable and the accompanying hash-table of non-searchable data are passed to a function called \texttt{AddDoc(String, String, Dictionary<String, String>)}.

By letting the programmer choose a unique key for the document, the programmer is able to use the same key for the document as the key used in the source of the data, such as a database primary key. The key of a document must be known by the programmer if there is a need to modify or remove a document.

The API library creates a document object containing all information that has been passed, which is stored inside a hash-table with the given unique key as a identifier.

The whole index resides inside volatile memory for the best performance and non-volatile memory was only used for backup.
In the step to make the document searchable the API library takes the string containing the terms that should be used to find the given document and splits it and stores all terms in an array. All terms in the array are added to a hash-table that is used as a lexicon with the document key as an identifier.

Each term in the lexicon has a list of posts in the form of

\[
\{ (d_x, f_x), (d_y, f_y), \ldots \}
\]

where each element is a post in the form of \((d, f)\), where \(d\) is the document key and \(f\) is the frequency of the given term in that document.

### 3.3.1.2 Searching

When a search query is made, a separator character for splitting the search string and a maximum number of documents to returned are passed. The algorithm then creates a result list in the form of

\[
\{ (d_x, h_x), (d_y, h_y), \ldots \}
\]

where \(d\) is the found document and \(h\) is a term hit counter. All terms are searched against the index one at a time and all results are recorded in the result list common for all of the term results. Each list of posts for a search term is retrieved and the document IDs of the posts are added to the result with a counter. If a document is then encountered additional times, the counter is incremented by one instead of adding another entry to the list. This list therefore only contains documents with term matches and can afterwards be ranked by sorting them by number of term hits per document, with the highest first.

\[
Sort((p1, p2) \Rightarrow p2.Value.CompareTo(p1.Value));
\]

A new list is created containing only the documents objects and only contains up to the desired amount of documents to be returned to the user.

### 3.3.1.3 Removing

When removing a document, its unique key is needed for finding it. The document is taken from the hash table containing all of the documents and then removed, thereafter all searchable terms for the document is looped through, and each term is deleted from the lexicon. If a term in the lexicon have posts for more than just the document to be deleted, then only the post for the document will be removed. If a term has no posts left, the term itself will also be removed from the lexicon.

### 3.3.1.4 Updating

When updating a document, the update method of the API called UpdateDoc(String, String, Dictionary<String, String>) is called. It takes the same parameters as the add method, but should contain the new updated information of an entry in the index. The method uses both the remove and add functions of the API library explained in section 3.3.1.1 and 3.3.1.3. The unique key for the document is saved and stays the same after the update is complete.
3.3.2 ASP.NET Server

To give the API library an HTTP interface it needs to be used in a programming environment that has those features, in the case of this thesis ASP.NET was used to achieve this. The API library therefore does not have a HTTP interface of its own, this gives the user of the API library more options and enables them to use the outward network interface of their choice. The ASP.NET server uses an HTTP interface together with a basic html page for easy testing of the search functionality of the API library.

Developers implementing the API library also have more options regarding populating and keeping the index up to date. The API library’s methods lets the implementer choose the update option that is considered most fitting and supported in their environment. In the case of this thesis, EntityFramework could be used for both populating and keeping the index up to date with the database, but instead predetermined documents that were hardcoded was used to populate the index for consistent testing with available resources. The index itself was contained in a static singleton handler class for easy use.

3.4 Tests

This section introduces the different tests that was carried out on the prototype to get an understanding of how well it performed. These tests were also carried out on a different already existing solution, to put the prototype into a perspective of how well it compared to a popular API library that had the ability of indexing and retrieving information from an index. The chosen solution to test against was Lucene.Net which was discussed in 2.3.4 and is a popular [18] solution for enabling search capabilities in web sites. Lucene was implemented in a copy of the same ASP.NET project used with the API library developed in this thesis. By using the same ASP.NET code to give both an HTTP interface, both solutions could be put under more equal conditions. The ASP.NET code also included the code for randomly generating a set of documents to populate the two indexes with, and also a couple of hard coded documents for special case testing.

3.4.1 Test on queries in different formats

Since Network Expertise is a Swedish company, searches that contained Swedish ISO characters had to be compatible. Other characters and common symbols that were requested by the company also had to be compatible. Therefore, a list of tests with different query formats was compiled. These queries would also be evaluated to make sure results were returned correctly and were accurate. The queries that would be tested were queries that contained:

- Swedish ISO characters “å”, “ä”, “ö”.
- Whitespace (“ ”).
- Hyphen (“-“).
- Underscore (“_”).
- Different letter cases such as “THESIS” and “thesis”.

In table X the two hard coded documents used for testing different queries are listed together with the queries used to test different query formats. Each test query listed in table X are inside a pair of quotation marks, during the testing of the queries the quotation marks were left out.
Table 6: Documents used in test with the different test queries.

<table>
<thead>
<tr>
<th>Entry in index</th>
<th>Test search queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>En åsna kom gåendes på en ödslig väg</td>
<td>”åsna”, ”åsna på”, ”ödslig”, ”ÅSNA”</td>
</tr>
<tr>
<td>Hej-på-dig vad heter_du</td>
<td>”hej-på-dig”, ”heter_du”</td>
</tr>
</tbody>
</table>

3.4.2 Tests on how fast a query result was returned
To compare the performance of both our prototype and the Lucene.NET implementation, a test to get an average of how long it would take to get a result returned from a query was planned. The test would be carried out on both implementations with three different index sizes: 10 000, 50 000 and 100 000 number of documents. Each index size would also be tested with different levels of load, meaning that one to four separate threads would do a pre-determined query on the index 100 times in sequence and then calculate an average time for that index size with a given number of threads.

3.4.3 Test on storage size
To compare the size of how much disk space the stored index of both our prototype and the Lucene.NET implementation would occupy, the amount of occupied disk space would be recorded and compared on three different index sizes: 10 000, 50 000 and 100 000 number of documents.

3.4.4 Generating random documents
The random generated documents all had strings built from randomly choosing a given number of characters. These characters were the whole English alphabet, numbers one to nine and the whitespace character (“ “). Each random generated document had 5 different fields stored for easy retrieval.

- **Name**, consisting of two strings of 5 characters each separated by a whitespace.
- **Sku** (Stock-keeping unit), consisting of 5 random characters.
- **Price**, a random number between zero and a thousand.
- **Short description**, consisting of 30 random characters.
- **Full description**, consisting of 100 random characters.

All of these fields were also combined into one long string used to make the document searchable, all combined fields were separated by a whitespace character.

3.4.5 Hardware
The test queries would be carried out on a computer with an Intel i7-2600k at 3.80GHz with Hyper-Threading and 16GB of RAM. This computer also hosted the index from inside Visual Studio. Both hosting and querying from the same computer was done in order to eliminate random time losses that can occur when testing between two computers over the internet.

3.5 Compilation of results
This section follows a compilation of the results of the tests outlined in section 3.4.
3.5.1 Format on queries
After the first test with different types of queries on the thesis prototype explained in section 3.4.1, it was shown that queries containing:

- Swedish ISO characters “å”, “ä”, “ö”.
- Whitespace (“ “).
- Hyphen (“-“).
- Underscore (“_“).
- Different letter cases such as “THESIS” and “thesis”.

Were all supported by the prototype developed in this thesis and returned the expected results.

3.5.2 Comparison of returned result times
Presented below in figure 1 is the second test that compared the time it took for a query result to arrive, for the thesis prototype and the Lucene solution. Each dot with their confidence interval included represents the average time it took for a query to get a result returned back, when a given number of threads sequentially makes one hundred HTTP GET calls to the index. In figure 1, both indexes had a size of 10 000 documents. The gap in time between the index developed in this thesis and Lucene is similar when queried by one thread, but continues to grow apart from two threads all the way up to four threads. With each additional thread the gap continues to widen, and at four threads difference between the two is at 45.7%.

![10k documents speed comparison at different loads](image)

Figure 1: Average result response time and confidence interval comparison between the prototype and Lucene at different loads with 10k documents. The load is between one and four threads, where each thread makes 100 sequential queries.
In the third test with an index size at 50 000 documents it can be seen that the trend from 10 000 documents also appears here, see figure 2.

Figure 2: Average result response time and confidence interval comparison between the prototype and Lucene at different loads with 50k documents. The load is between one and four threads, where each thread makes 100 sequential queries.
In the fourth test with an index at the size of 100,000 documents shown in figure 3, the index developed in this thesis still exceeds Lucene’s index when queried by multiple threads. Lucene’s index is however faster when queried by one thread by 22.6%. Lucene were also faster than when queried with 50,000 documents.

Figure 3: Average result response time and confidence interval comparison between the prototype and Lucene at different loads with 100k documents. The load is between one and four threads, where each thread makes 100 sequential queries.
The fifth test was a storage size comparison for the index when populated with 10,000, 50,000 and 100,000 documents. As can be seen in figure 4, the prototype developed in this thesis increases drastically the more documents are added to the index. Between 10k and 50k the increase is around five times the size, and between 50k and 100k it increases even further to twice the size.

Meanwhile Lucene is not taking up much space, at 50k the size is just 50MB, which is 65% smaller than the prototype. Even at 100k documents, Lucene was just shy of 75MB and a whole 74% smaller than the prototype at that index size.

![Storage size comparison different sizes](image)

Figure 4: Comparison between the prototype and Lucene's storage requirements at different index sizes.
4 Analysis and discussion

This chapter is composed of the students’ own interpretations and evaluations. The chapter covers an analysis of the implemented prototypes and their results, as well as their social, economic and environmental aspects.

4.1 Difference in results

The results of how fast a response to a query arrive was clearly in favor of the prototype developed in this thesis. The gap between the prototype and Lucene became greater and greater the more documents that were added, and also became more apparent by increasing number of queries made simultaneously. This big gap can likely be because of Lucene being a lot more feature rich when it comes to its searching capabilities, such as being able to search for part of words. The prototype does not have these features however, and therefore becomes more streamlined with less overhead when carrying out search operations.

Another reason can be that Lucene also compress its information since it stores and mainly work against non-volatile memory during run time, while the prototype does not compress any information and stores everything in volatile memory during runtime. The prototype also works directly with volatile memory and only saves the index to a non-volatile memory if asked. This gives Lucene a huge advantage when it comes to space efficiency, which is an important aspect when it comes to indexes and could also become a problem for the prototype if it has to store more information than the amount that was tested in this thesis.

4.2 Suffix and postfix

Searching the index with suffixes and allowing wildcards for postfix searches are something that is missing from the prototype developed in this thesis. This could enhance the user experience further and could also be combined with suggestion to even further enhance the experience.

4.3 Managing database changes

The API library does not support automatic detection of database changes since this would limit its uses and what data sources it would support. Instead the functionality of updating was developed in the API, but the responsibility for handling changes in the source was handed over to the person implementing the library in a bigger project. Because of time constraints this functionality was not added to the ASP.NET project implementing the API library in this thesis. Instead code was used to populate the index and this also made it easier to test different index sizes compared to populating a database first, which was much slower than populating a data structure inside the same application. But there is nothing in the API stopping someone from using whichever source they desire and a more general API library makes for more use cases.
4.4 Storing index to files
When saving and loading the index from files on a non-volatile memory, DataContractSerialize was used due to time constraints. Since DataContractSerialize uses XML which is human readable and since there is a lot of data structures in the index, there will be a lot of overhead because of this. This overhead also made it so that the index took up a lot of disk space which increased very quickly. This can also cause problems when loading the index into ram from non-volatile memory, such as crashes. This problem could be reduced by creating a custom file handler which does not use as much overhead.

4.5 Social and economic aspects
During the planning and execution of this thesis both the social and economic aspects have been considered. The underlying goal of this thesis has been to set up a back-end that would allow to give humans a more positive user experience, and to fulfill the demands that exist today. The idea of collecting products and storing them in a single accessible location is to reduce the amount manual workload customers would otherwise have to go through when searching different website for what they need. This would also contribute to the economic growth of the company. All of the development in this thesis has been carried out in a sustainable way, and with future purposes in mind.

4.6 Environmental aspects
One of the major objectives of this thesis was to decrease the strain on the database. Hardware is prone to break at some point in time. If the database is constantly under a big load from users, the hardware will fail faster and needs to be replaced more often. By decreasing the load on the databases, components do not need to replace as often. This therefore leads to less components needing to be disposed of, and would decrease the demand for newly manufactured components. This will be advantageous for the environment since computer components contain rare minerals, and it would also decrease the emission of greenhouse gases and other toxic waste that the shipping and manufacturing of new components would have otherwise created.
5 Conclusion

A working server application that was separated between an API and a ASP.NET project containing algorithms and associated data structure was developed to be able to index the products of a web shop, and make the products searchable from the index. The goals of the final prototype together with the study of the underlying algorithms and data structures of the implementation were achieved. The final prototype was able to return results when queried while under heavy load 45.7% faster than the popular solution that it was compared with. Looking at the overall purpose of the server application it was able to deliver relevant query results. With continued work on index compression and storing procedure, together with furthering its search capabilities to enable wildcard searches for postfix and searches with suffixes, the algorithms of the prototype could be improved to further enhance the user experience.
References


Appendix

1 Knowledge Discovery in Databases

Knowledge Discovery in Databases or KDD for short is the process of discovering useful information out of a set of data from one or multiple databases. KDD can be separated into different phases that needs to be carried out in order. These different phases can for some applications be very simple and for others more complex. The phases are in the following order:

Cleaning, for removing noise and inconsistent data from sources,

Integration, for merging multiple sources to one,

Selection, for choosing which data to be used,

Transformation, for combining or changing the form of data,

Data mining, for finding patterns in the data and lastly,

Presentation, for presenting the result in a way that can be understood by humans. [1, 2, 3]

KDD is used in a wide variety of different business areas like scientific research in astronomy and medicine, government security for combating crime and terrorism, and in financial business like banking and retail. [4, 5]

1.1 Cleaning

Before data can be mined it has to be cleaned from noise and inconsistent data with missing values. In the case of missing data, where for example several attributes in a tuple has no recorded value one can either ignore the tuple, manually fill in the value, use a global constant, use a mean value of the attribute, use the attribute mean for all samples in the same class of the tuple, or use the most probable value for the missing value. To use the most probable value for the missing value is the most common method, since it is based on the information of present data and therefore increases the chances of preserving the relationships between other attributes.

Noise is a common occurrence where a measured variable has a random variance. To reduce noise one has to use a method to smooth out the data. This can be achieved by several techniques, such as Regression, where data is modeled and smoothed to fit a straight line, Clustering, the data is grouped into clusters of similar data, Binning, where data values are sorted into so called bins and are smoothed by other neighboring values. [1]
1.1.1 Regression technique
In regression a function is used to find the most appropriate line to fit two attributes as shown in figure 1.

Figure 5: Example of regression where data is modeled to fit a straight line.

Doing this enables one attribute to be used to predict the other. In function 1, a response variable $y$ can be modeled linear together with another random predictor value $x$. Both $x$ and $y$ are numerical attributes that exist in a database, and the variance of $y$ is assumed to be constant. The regression coefficients $w$ and $b$ determines the line of the function and the $Y$-intercept respectively.

$$y = wx + b$$

Regression can be used on sparse data, though it can be a little limited and should rather be used on skewed data. It is also very computation heavy when applied to high-dimensional data. It is best used when investigating the relationship between multiple variables. For example, when predicting the velocities of wind and its directions as a function of temperature, humidity and air pressure. [1, 2]

1.1.2 Clustering technique
In large databases, it is quite common to have large sets of data objects where the class labels are unknown since the process of labeling large amount of objects can be costly. When mining data of this kind, clustering is used. During the process, data is grouped into classes or so called clusters, where each cluster contain objects with high similarity. Each cluster is very dissimilar from each other, where the dissimilarities are determined by the attribute values of the objects. After objects have been partitioned into smaller groups with clustering, the groups can be assigned labels which is much more cost efficient. Values that fall
outside the clusters are called outliers and can be used for later outlier analysis if needed. [1]

Clustering is difficult to execute on categorical data, since most algorithms are designed around numerical attributes, this makes identification of categorical attributes expensive. Databases can also contain attributes such as nominal, ordinal and binary, these attributes needs to be converted to categorical to simplify identification. If the number of class labels is not known beforehand, an extensive analysis has to be made by an algorithm. Identifying and deciding different class labels for databases with mostly numerical attributes is really fast because of its natural use with arithmetics. [6]

1.1.3 Binning technique
When using the binning technique, the set of data is split into smaller subsets at a given interval. If the number of elements in each subset is the same for all subsets, then it is called Equal Width Binning. [7]

If the subsets are divided into k subsets with almost equal number of elements, then it is called Equal Frequency Binning.

Binning uses categorical data, alas numerical attributes are converted into categorical attributes. This makes binning more suited for databases where the majority of data is categorical data as opposed to Clustering where numerical data is best suited. [8]

1.2 Integration
If the data to be analyzed is made up of multiple sources, then the sources will be combined into one data set in the integration phase of KDD. When integrating different sources, there might be problems with entity identification. This problem arises when sources have different names for the same attributes in entities that should be equivalent. Using metadata for every attribute could help mitigate the number of errors occurring when doing schema integration and object matching. [1, 9]

One other problem that could arise during integration is redundancy, meaning that one attribute can be derived by using other attributes and can therefore become redundant. Correlation analysis can be used to find some redundancies by analyzing how much an attribute implies on other attributes. [1, 9]

\[
r_{A,B} = \frac{\sum_{i=1}^{N} (a_i - \bar{A})(b_i - \bar{B})}{N \sigma_A \sigma_B} = \frac{\sum_{i=1}^{N} (a_i b_i) - N \bar{A} \bar{B}}{N \sigma_A \sigma_B}
\]  

(2)

If the attributes are numerical they can be analyzed by calculating the correlation coefficient between two attributes A and B. In the equation \(r_{A,B}\) N is the number of tuples, \(a_i\) and \(b_i\) are the values from set A and set B in tuple i. \(\bar{A}\) and \(\bar{B}\) is the average values of set A and set B, \(\sigma_A\) and \(\sigma_B\) is the standard deviations of set A and set B respectively, and \(\sum(a_i b_i)\) is the product between \(a_i\) and \(b_i\) for every tuple in set A and set B.
The result from $r_{A,B}$ has a range of $-1 \leq r_{A,B} \leq +1$ and shows that the set $A$ and set $B$ are independent of each other if the result is 0. Meanwhile if the result is positive then the values of $A$ increase together with the values of $B$, and if the result is negative then the the values of $A$ decrease together with the values of $B$. $r_{A,B}$ does not necessarily show that $B$ correlates with $A$, only that $A$ correlates with $B$.

$$\chi^2 = \sum_{i=1}^{c} \sum_{j=1}^{r} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}$$

For attributes that are categorical data or discrete data, they can be analyzed by calculating the $X^2$ (chi-square) between set $A$ and set $B$. Suppose that set $A$ has $c$ discrete values and set $B$ has $r$ discrete values. With a contingency table, tuples by set $A$ and set $B$ can be described, where the discrete values of $c$ from set $A$ is the columns and the discrete values of $r$ from set $B$ is making up the rows. The contingency table is filled with elements of the form $(A_i, B_j)$.

$X^2$ uses the observed frequency $o_{ij}$ and the expected frequency $e_{ij}$ of $(A_i, B_j)$.

$$e_{ij} = \frac{\text{count}(A = a_i) \times \text{count}(B = b_j)}{N}$$

The expected frequency uses the number of tuples $N$, how many of the tuples in set $A$ with the value of $a_i$, count($A = a_i$), and the number of tuples in set $B$ with the value of $b_j$, count($B = b_j$). [1]

### 1.3 Transformation

Data transformation is the process in KDD where data are transformed or combined into forms more suitable for data mining. This reduces the amount of data and makes it easier to analyze. [10]

The techniques that are involved in data transformation are:

- **Smoothing**, which is explained in section 2.2.1.
- **Aggregation**, when computing the sum of elements for example, and when building data cubes for analysis of data. [1]
- **Generalization**, where concept hierarchies are used for replacing raw data with higher-level concepts. [10]
- **Attribute construction**, which builds and adds new attributes from an existing set of attributes to assist the mining process. [10]
- **Normalization**, which is used when scaling data so it can fall within a certain range, such as 0.5 to 1.0. This can be useful for distance measurement in clustering and nearest-neighbor classification. For example, one of the methods in normalization is called **min-max normalization**, which performs a linear transformation on the data. Suppose you have an attribute $A$, where $\text{min}_A$ and $\text{max}_A$ are its minimum and maximum values. With **min-max normalization**
tion you are able to map a value \(v\), of the attribute \(A\) to \(v'\) in the range \([\text{new}_\text{min}_A, \text{new}_\text{max}_A]\) by calculating:

\[
v' = \frac{v - \text{min}_A}{\text{max}_A - \text{min}_A}(\text{new}_\text{max}_A - \text{new}_\text{min}_A) + \text{new}_\text{min}_A
\]

This preserves the relationships between the original data values, and will give an error if an input falls outside the original range of \(A\). [1]

### 1.4 Data mining

This subchapter explains the process of data mining, how the sources that contain the data work, and how models and classification techniques can be used by the data mining process to extract useful information when dealing huge amounts of data.

Data mining is the action of extracting the knowledge from a database containing sets of data with chosen algorithms and methods. During data mining these algorithms and methods are used to find patterns in the data set. The patterns acquired can also go through the whole process again to further analyze the data.

There are multiple use cases for the data mining part of KDD, discovery and verification. Verification is used to verify a hypothesis made by the user, while discovery is used to find new patterns.

Discovery can be split into prediction and description, where prediction is used to find patterns that tell about the behavior of entities in the future, such as patterns used in the technical analysis for the financial market. Description is when patterns are used for presentations understandable by humans. [1, 2]

#### 1.4.1 Models

Models are used in data mining to apply already known knowledge learned from data that has been already mined onto new data to generate predictions and make inferences about relationships. Models can be applied in a statistical or logical way. When applying it statistically, the model does not have to stay the same between every time it is being applied to the data, while applying it logically the model stays the same. The statically way is used most of the time because of the uncertainty in most real-world applications.

The data mining process uses methods to apply these models to the data, the methods consist of model representation, model evaluation and search as its primary algorithmic components. These methods are based on techniques that uses machine learning, pattern recognition and statistics. Such techniques are for instance classification, regression and clustering. Regression and Clustering are covered in 2.1.1.1 and 2.1.1.2. [2]

#### 1.4.2 Classification

Classification is one of the most commonly used data mining technique, it involves predicting data based on a given input. With a classification technique there are several pre classified classes that data items get mapped to, a model gets constructed by basing it on pre classified examples. Classification by decision tree induction and neural networks are some types of classification models often used. The pre classified examples are analyzed by the
classification algorithm to figure out the accuracy of the algorithm. If the classification algorithm is accurate enough it will be employed on new data items. A set of required parameters created by the classification algorithm is then encoded into the model to properly distinguish the data items. [11]

1.4.3 Relational databases
A relational database is one of the sources that data can be extracted from during data mining. It is a database that uses the relational model to store data, and is the most commonly used model today. It uses tables with tuples and attributes to store its data.

In the relational model, a relation is equivalent to a table. Every tuple in a relation contains data of a specific item which can for example be a product that a store provides, where the attributes signifies a property the product can have, for example price. The tuples in a relation constitutes a set and does not have to be in a specific order, but needs to be unique. This data can then be accessed by relational queries written in a relational query language, for example SQL.

Relational queries enable specific sub-sets of data to be retrieved, for example a query can be formed to show a list of all products within a certain price range. A relational language also enables the use of aggregate functions which can be used to calculate average, maximum, minimum, number of elements and sum of elements. Aggregate functions are very useful when mining relational databases. It allows deeper searching for trends and data patterns, and lets you extract parts of information from the data that is considered useful for further analyzation or processing. [12]

1.4.4 Multiple data sources
If a large company has many branches with lots of databases spread out on each branch, getting relevant data from each database is a difficult task to accomplish. A solution to this problem is to use something called a data warehouse. A data warehouse works as a repository for data that has been mined from multiple sources, and stores this data in a single unified schema at one central location. The data in a data warehouse goes through a process of cleaning, integration, transformation, loading and periodic refreshing, and is organized around a certain major subject, for example the range of products a company provides. [1]
References Appendix 1
