Specialized Search Engine with AI

PHILIP THORN

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LECS-2004-3

Inst för Mikroelektronik och
Informationsteknik
Kungl Tekniska Högskolan
100 44 STOCKHOLM

Dept of Microelectronics and
Information Technology
Royal Institute of Technology
SE-100 44 Stockholm, SWEDEN

Specialized Search Engine with AI

Master of Science Thesis

PHILIP THORN

IMIT/LECS-2004-3

Master's Thesis in Internetworking (20 credits)
at the Department of Microelectronics and Information Technology,
Royal Institute of Technology, Stockholm, Sweden, January 2004

Examiner and advisor is Vladmir Vlassov
at Department of Microelectronics and Information Technology

Abstract

Today's search engines focus mainly on the static content of the web. Since more and more websites are being created with dynamic pages, i.e. with server side scripts that queries a database and creates web pages "on the fly" based on some input criteria, this is a problem that is growing rapidly.

This project consists of designing, developing and evaluating a search engine for dynamic data in the form of second hand products available on the Internet in Sweden. Another part of the project is to improve the quality of the search results by equipping the search engine with artificial intelligence.

This report describes and compares different search engines and the technologies behind them. It gives a thorough description of the components in the system and motivates the choice of solution and platform.

The prototype search engine worked so well that I decided to make it available to the public. The search engine can be found at http://www.annonsguiden.nu/. If you have any questions or comments I can be reached at philip@evision.nu.

Acknowledgements

Building this search engine has been an interesting and educational experience throughout the entire project. I would like to thank all users for their positive feedback; it really motivated me to push this search engine to its limits. I also would like to thank my academic advisor and examiner Vladimir Vlassov for guiding me when writing the report.

Table of Contents

1 Introduction

Today there exist about 50 different websites in Sweden where you can buy and sell second hand products. As a consumer you would want to search among all the products these sites can offer, but it is very troublesome and time consuming to visit all of these sites, and the average consumer probably only knows one or two of them.

If you use a typical search engine to find a second hand product you probably won't find any of these products since they reside on dynamic pages. Even if you did (some search engines like Google have started to index dynamic pages) you would get a lot of irrelevant search results. To find such things with good accuracy you'll need a specialized search engine, i.e. a search engine that indexes dynamic data and focuses only on a sub-set of the World Wide Web to answer a subject specific query.

The goal of this project is to design, develop and evaluate a search engine for all second hand products available on the Internet in Sweden. I chose second hand products because it addresses three of the biggest problems with today's standard search engines:

- Indexing of dynamic data
- Irrelevant search results
- Frequently expiring pages

The search engine will be written as a website for ads of second hand products, i.e. with the possibility to add your own ad to the system. It will also be equipped with a special kind of artificial intelligence to improve the search results. The search engine will be optimized for speed and accuracy and be able to handle many simultaneous users. Since database connectivity is a big bottleneck, methods that relieve the database from as much workload as possible will be implemented.

2 Background

When most people talk about Internet search engines, they really mean World Wide Web search engines. The Internet is older than the Web and there have existed search engines long before the Web became the most visible part of the Internet. The early search engines kept indexes of files stored on different servers connected to the Internet. The ones that dominated were search engines with names like Gopher and Archie.

Today, most Internet users limit their searches to the Web. There exists a variety of implementations of different types of search engines, each customized to serve its purpose best. The search engines that dominate today are search engines with names like Google and Yahoo.

This chapter gives an overview of different types of search engines. It is a summary of a case study I conducted [1], [2], [3], [4], [5], [6], [7] to get inspiration and ideas before I designed the prototype search engine for this project.

2.1 Typical search engines

A typical search engine is a place on the Internet where you can search for just about anything you want. It is what you normally think of when the word ‘search engine’ is mentioned.

From the users’ perspective it works as follows:
You type some keywords in an input field at the search engine’s homepage and press a submit button. Then you’ll receive search results in form of a list of links to pages on the Internet where the information you are looking for hopefully exists. There are usually a couple of lines of text next to the links that is taken from the pages the links point to, so you can make better decisions on which links you want to visit.

From the server’s perspective it’s a bit more complicated. Before I go into details I must mention that finding in-depth information about these types of search engines is very difficult, since this kind of information is sensitive and secret. Sergey Brin and Lawrence Page, the creators of the world’s most popular search engine – Google, describe how their system works in their paper [1]

“The Anatomy of a Large-Scale Hypertextual Web Search Engine”. Most of this overview is based on that paper, but for a more general case.

Figure 1 shows an overview of how the components are connected. First a short description is given and then all of the components are described in more detail.

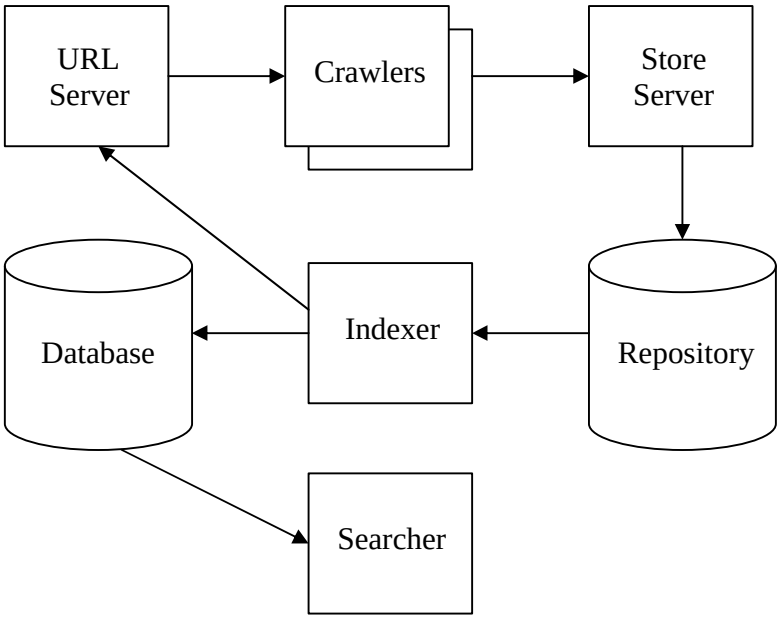

Figure 1. High level system architecture of a typical search engine.

The web crawling (downloading of web pages) is done by several distributed crawlers. There is a URL server that feeds the crawlers with new URLs to be fetched. The web pages that are fetched are then sent to the store server. The store server compresses and stores the web pages into a repository. The indexer performs a number of functions. It reads the repository, uncompresses the documents*, and parses them. Then it builds a database of all the words that are found in the documents along with some info about where the word was found, font size etc. It also extracts all the links found within each document. These links are sent to the URL server. The database holds a searchable index of the entire repository. The searcher queries the database with keywords submitted by users, builds hit lists from the search results and presents them to the users.

(*) I will refer to a downloaded page as a document.

2.1.1 URL Server

The URL server has one single purpose – to feed the crawlers with new URLs to crawl. The URL server holds a queue of URLs which will be visited by the crawlers. URLs are added to the queue from different parts of the system. Most of the URLs come from the indexer, but a lot also come from web page owners who submit their pages' URLs to the search engine. Before the search engine runs for the first time the queue needs to be filled with some starting pages that hopefully will keep it busy for a long time. Normally, the administrators will add URLs of very popular sites and heavy used servers. This way, the crawlers will quickly spread out across the most widely used portions of the web.

2.1.2 Crawlers

Crawlers are also known as 'spiders' or 'robots'. Running a web crawler is a challenging task; there are tricky performance and reliability issues, and even more importantly, there are social issues. It's also the most fragile application since it involves interacting with hundreds of thousands of web servers and various name servers which are all beyond the control of the system.

In order to scale to hundreds of millions of web pages, search engines normally use several distributed crawlers that run in parallel. This is necessary to retrieve web pages at a fast enough pace. Google's system uses four crawlers. Each crawler keeps roughly 300 connections open at once. At peak speeds, the system can crawl over 100 web pages per second using all four crawlers. This amounts to roughly 600 kB of data per second.

A major performance stress is DNS lookup. To reduce this bottleneck, a crawler usually maintains its own DNS cache, so it does not need to do a DNS lookup before crawling each page. Each of a crawler's hundreds of connections can be in a number of different states: looking up DNS, connecting to host, sending request and receiving response. These factors make the crawler a complex component of the system.

If a web page contains sensitive information and the page owner doesn't want the page to show up on major search engines, he can indicate which parts of the site that should not be crawled, by

providing a specially formatted file called ‘*the robot exclusion protocol’*, or use a special meta tag called ‘*the robots meta tag’*. All major search engines support this protocol and the crawlers will respect it.

Crawlers usually avoid pages with URLs that contain escape characters like &, ?, =, etc. These pages often lead to dynamic pages with recursive links that can easily trap crawlers in a maze of data. Sometimes this poses a threat to the crawlers, but more often than not, a trapped crawler will simply bring a server to its knees in a couple of minutes.

2.1.3 Store Server

The store server receives fully downloaded web pages from the crawlers. Each page is assigned a unique ID, compressed and put into the repository. The choice of compression technique is a tradeoff between speed and compression ratio. Google uses a compression library called ‘zlib’, which is a choice of speed over compression.

2.1.4 Repository

The repository holds the full HTML code of every web page the crawlers have fetched. The goal of every typical search engine is to have every web page in the world in its repository. This may sound impossible but Google is actually not that far away as it currently holds over 3.5 billion pages in its repository. AltaVista currently holds over 500 million pages in its repository. In the repository the documents are stored by ID, length, URL and the compressed HTML page. All data structures in the system can be rebuilt from just the repository. The repository is stored on disc; preferably in such a way that a record can be fetched in just one disc seek.

2.1.5 Indexer

The indexer makes the documents in the repository searchable. To do so, it fetches them from the repository, uncompresses them and parses their content, i.e. filters out all HTML tags and extracts every word in the document. Each word that is found is recorded along with some information about the word, such as its position in the document, its font size, if it was found in the headline, the title or in the meta-tag. This extra information about each word is used in order to calculate the best search result when a search is made.

Each search engine stores different information about the words. Some may, for example, ignore the font size, and some may treat all words equal regardless of where in the document the word was found. This is one of the reasons why a search for the same word on different search engines will produce different results.

Google also uses something called '*page rank*'. This is a rank that is given to each document based on how many other pages link to the document. The more pages that link to it, the higher rank it gets. The page rank algorithm also considers the rank of the pages that link to a document. A link from a page with high page rank weights more than a link from a page with low page rank.

The indexer builds an index of all the words in the repository and stores it in the database. An index has a single purpose; to allow information to be found as quickly as possible. There are quite a few ways for an index to be built, but one of the most effective ways is to build a '*hash table'*. A hash table is a container with an index. The index maps '*keys'* to '*values'*. If you have the key you get the value directly. This is similar to looking up a person in the white pages. If you know his name you don't have to start at A and work your way to Z. In this case, the key would be the name and the value would be the phone number.

The actual keys and values are never stored in the hash table. Instead, a formula is applied to attach a numerical value to each key, a so called '*hashed number'*. The formula calculates the hashed number based on the actual value of the key and the pre determined number of entries in the hash table. The hashed number will be stored instead of the actual key, and a pointer to the value will be stored instead of the actual value. When a search is made, the hash formula is applied to the key, giving it a numerical value which corresponds directly to the position in the

hash table where that entry resides. This means that not only will the hash table be very small in size since the actual keys and values are never stored, but it will also be very fast.

The index is built as follows. For each word in a document, the indexer calculates its hashed number. With this key it gets a list of all the documents that possess that word from the database. Each object in the list contains a documentID and information about how that word was represented in that document. The indexer creates a new object, containing the current document's ID and information about the word, and appends it to the list. The list can be sorted in different ways depending on how the search engine will represent the result.

The indexer also builds an index of the document. It maps the documentID to information about the document, such as URL, title, some text and a pointer to its location in the repository.

Besides building indexes, the indexer also extracts all the links found within the documents and sends them to the URL server. This way, the crawlers will recursively index every page on a website and spread out across the Internet.

2.1.6 Database

The database is used for real time searches. Therefore it needs to be as fast as possible. A disc seek normally takes about 10 ms to complete, compared to a memory seek which takes about 50 ns. For this reason, the relevant parts of the database are preferably kept in main memory. To scale this to a major search engine like Google, a cluster of hundreds of servers is used [8]. Each server holds a part of the database in memory and some on disc. The servers are also used to process search queries. The database is also stored on several backup drives, so whenever a server crashes, a new one takes its place by loading its data from the backup drive. This solution makes the system robust; an engineer can replace the broken server whenever he finds the time.

The database has two indexes: a document index and a word index. The document index maps documentID to information about the document, such as URL, title, some text and a pointer to its location in the repository. This index is very big – several terabytes, and is therefore kept on disc. It will only be called once the search result is computed and the result page needs to be written; that is present the URL, title and some short text to the user.

The word index maps words to lists of objects. Each object holds a documentID and information about the word for that document, such as font size and location. Some search engines, like Google, doesn't apply a hash formula directly on the words to calculate the keys in the index, but instead they convert the words to wordIDs, which they get from a lexicon that is kept in main memory. To do this efficient they have divided the database into different "barrels" where each barrel holds a range of wordIDs. This way, the wordIDs stored in the barrels will just be an offset from the minimum value each barrel can hold; hence the size of the word index will be reduced. The word index is much smaller than the document index, but still too big to fit in main memory, even for a cluster of hundreds of machines. Therefore the relevant part of the word index is kept in main memory, which consists of the most recently searched words and the most frequently searched words.

2.1.7 Searcher

The searcher processes search queries from online users. A search consists of keywords and Boolean operators. The Boolean operators can be different for each search engine, but these are the most commonly used:

AND - All the terms joined by "AND" must appear in the pages or documents. Some search engines substitute the operator "+" for the word AND.
OR - At least one of the terms joined by "OR" must appear in the pages or documents.
NOT - The term or terms following "NOT" must not appear in the pages or documents. Some search engines substitute the operator "-" for the word NOT.
FOLLOWED BY - One of the terms must be directly followed by the other.
NEAR - One of the terms must be within a specified number of words of the other.
EXACT PHRASE - All words are treated as a phrase, and that exact phrase must be found within the document. A quotation mark '"' is normally used to indicate the beginning and end of the phrase.

When a search is executed all the words and their Boolean operators are sent to the searcher. The searcher checks which documents contain those words based on their operators. If no operators are submitted the AND operator is normally used as default.

The process for doing this works as follows:

1) The searcher receives a search query string from the user.
2) The search string is parsed, i.e. the words and their Boolean operators are extracted.
3) The words are converted to their hashed values (or wordIDs).
4) The searcher calls the database (the word index) with these values and receives a list of documents that contain these words.
5) Each document in the lists is checked if it fulfills the search condition and is given a rank based on how the word was represented in the document.
6) All documents that matched all search terms are sorted by rank.
7) URL, title etc. for the N first documents are extracted from the database (from the document index).
8) The result is presented to the user (N first documents and links to more results).

2.2 Peer-to-Peer (P2P) search engines

P2P search engines [6] are quite different from typical search engines. Typical search engines searches among web pages on the whole Internet, while a P2P search engine searches among files located on all the peers connected to the P2P network. To access a typical search engine a user only needs a browser, but to access a P2P search engine a user needs to install and run a third party program in order to join the P2P network.

There are two categories of P2P networks:

- Centralized P2P networks
- Decentralized P2P networks

2.2.1 Centralized P2P networks

In a centralized P2P network there exists a single central server, which maintains an index of the shared files stored on the respective computers of every user that is connected to the network. When a user searches for a file, the central server creates a list of files matching the search request by cross-checking the request with the server's database of files belonging to users who are currently connected to the network. The central server then sends that list to the requesting user. The requesting user can then choose files from the list and make direct connections to the individual computers which currently posses that file.

Advantages of a centralized architecture

The main advantage of the centralized architecture is the central index which locates files quickly and efficiently. Also, because all clients have to be registered as part of the network, search requests reach the files for all logged on clients, which ensures that the search is as thorough as possible.

Disadvantages of a centralized architecture

The central server system provides a single point of failure and a visible target for attacks on the network. Also, because the central server index is only updated periodically, there is a possibility that a client receives outdated information.

One of the most famous centralized server networks was Napster, which was shut down due to legal aspects. It has been replaced by OpenNap, which is an open source network of servers running the Napster server-client protocol.

2.2.2 Decentralized P2P networks

The concept of decentralization is to remove the central structure of a network such that each peer can communicate as an equal to any other peer. The peers are connected together in a tree-like structure. When a peer sends a request, it is sent to the peer's child and parent nodes. They in turn do the same thing, and the search is propagated out through the net. Although this theoretically allows for an infinite network, in practice a time to live (TTL) is used to control the number of nodes a request can reach.

If a peer has a file that matches the request, a reply is propagated back the same route to the peer that sent the request. The file can then be downloaded by establishing a direct connection between these two peers.

Advantages of a decentralized architecture
They are more robust, because a single point of failure is eliminated. They are also harder to shut down (which may be a disadvantage to some).

Disadvantages of a decentralized architecture
Searching a decentralized network is slower. You are not guaranteed to find a file even if it is on the network because it may be so far away that the TTL expires before it reaches a peer that has it.

KaZaA and Morpheus are two of today's most popular decentralized network applications.

2.3 Directory search engines

Directory search engines divide all their content in different categories in a ‘directory like’ structure.

Instead of crawlers, directory search engines are edited by humans. They are the ones who decide which sites to list and to which categories. Before they submit a page to the directory database they review it and decide which category the page is most suitable for. Depending on the content of the page the editors will also provide keywords which don’t necessarily reflect the keywords in the page, making keyword searches more accurate than in a typical search engine.

To add a web site to a directory search engine you enter the site’s URL and provide a short description of it. Before the site is added to the directory an editor will review it and decide if it is appropriate for the search engine. If so, the editor chooses the appropriate category and the most suitable keywords for the site.

The main advantage of directory search engines is that they often provide much more targeted results than typical search engines. You can also find pages that match your interest without submitting any keywords, by just simply browsing different categories in the directory structure.

The main weakness is that they are hard to keep up to date since humans must filter and maintain each addition. Broken links are usually identified by using robots, but sometimes a site changes its theme or topic which makes its entry in the database incorrect. It is too expensive for humans to continuously check each link in the database, so this frequently happens.

2.4 Meta-search engines

Meta-search engines [7] provide the ability to search in several search engines at once. This makes it possible to exhaust one search and retrieve results from many different search engines in one go.
Unlike the typical search engines and directories, meta-search engines do not have their own databases, they do not collect web pages, they do not accept URL additions, and they do not classify or review web sites. Instead, they send queries simultaneously to multiple web search engines and/or web directories. In many meta-search engines it is possible to choose which search engines are to be included in your search.

Successful use of a meta-search engine depends on the status of each of the individual search engines used. Some may be heavily loaded at the time; some may be unreachable. A serious problem with many of the meta-search engines is slow response time since they are dependent of the speed of each individual search engine. Many of them, therefore, have a timeout period, so that attempts to work with a particular search engine can be abandoned if no response comes from it within a set period of time.

Another disadvantage of the meta-search engines is that they can’t take advantage of all the features of the individual search engines. Since a meta-search engine has a uniform search interface and syntax, it is difficult to apply this against the diversity of individual search engines. Boolean searches, for example, may produce varied results, and some Boolean operators may not be supported.

Moreover, meta-search engines generally do not conduct exhaustive searches: they do not bring back all the pages from each of the individual search engines. They only make use of the top 10 to 100 hits from each of them. While this is sufficient for most searches, individual search engines must be consulted if one needs to go beyond the top hits as determined by the meta-search engines. Some meta-search engines facilitate this need by providing query links back to the individual search engines.

2.5 Specialized search engines

A specialized search engine is a search engine that only focuses on specific information. There exists a variety of implementations of specialized search engines which can be divided into two categories; crawler based and non crawler based.

The majority of all specialized search engines are non crawler based. An example of a non crawler based specialized search engine is Amazon.com, which keeps a searchable database of their inventory. An example of a crawler based specialized search engine – similar to the one that I'm going to build, is Pricegrabber.com, which has a searchable database of new products, that is updated by crawlers.

3 Method

The goal of this project is to design, develop and evaluate a search engine for all second hand products available on the Internet in Sweden. In addition to being a search engine, it should be possible to add your own ads to the system. The search engine should support searching by keywords, Boolean operators and by other criteria such as category, type and region.

The system should have some form of artificial intelligence to improve the quality of the search results. Also, the whole system should be optimized for speed and accuracy.

This chapter describes my choice of underlying structure and how I implemented the different aspects of the search engine in order to accomplish the goals I have set out.

3.1 Technology

This search engine focuses only on second hand products; hence it is a specialized search engine. In addition to that, it uses a combination of all the technologies from the different search engines that are described in the background section.

Like a meta-search engine, this search engine searches through several sites' databases. To make this fast and efficient I used the centralized structure of a P2P search engine to store the entire index of these databases locally on one centralized server. The index is built by robots, similar to the web crawlers of a typical search engine. Like a directory search engine, where humans decide which pages they should index and under which categories they should be put, the robots were told which information should be retrieved and how it should be categorized.

3.2 Platform

3.2.1 Hardware

I decided to use a dedicated server for this project. The decision of what type of hardware it should be equipped with was based on two things; the type of solution I was going for and my budget. Since I am going to use a concept called '*caching*', which is described in detail in chapter 4.3.5, a relatively fast processor and a lot of main memory was needed. I went for a 1.4 GHz AMD processor with 1.5 GB of main memory, which should be adequate. The server has a 10/100 Mbit network card that is connected to a 2.0 Mbit Internet connection. The server uses two 40 GB hard drives; one for live data and one for backup data.

3.2.2 Software

The server runs on a Windows 2000 server (service pack 3). I chose between Linux and Windows 2000 server. I decided to go for Windows since it is more convenient to work with and it supports more third party software. The web server is the one that came with the installation of Windows, which is Internet Information Server (IIS) 5.0. It works well as long as you remember to patch it on a regular basis. The database is Microsoft's SQL 2000. It is a very fast database and has a nice interface to work with and supports stored procedures and such.

The obvious choice for program language was Java. This is because it is an advanced program language that I could build the robots and the search engine with. It is also possible to build dynamic web pages; so called Java Server Pages (JSP) or Java Servlets [11]. The caching could also be built very efficient; the robots, the search engine and the servlets will all share the same memory space, and since Java is an object oriented program language, it is possible to build classes that correspond directly to the SQL tables. The system could also easily be ported to a different operating system because Java is a platform independent language. I chose the latest SDK from Sun, which is version 1.4.2.

Microsoft recently provided a free class 4 JDBC driver that I am using for communication between the Java Virtual Machine and the database. Most web servers, like IIS, don't support Java by themselves; a third party java servlet engine is needed. I chose Resin 2.16 as Java servlet engine because it is one of the fastest servlet engines, it supports IIS and you can use it for free if you are a student or a start-up company without substantial income.

3.3 Architecture

This section describes the architecture of the prototype search engine. Figure 2 shows a diagram of the system and how the components are connected. First, the system is described in general, and in the next section, a detailed description of how each component is implemented will be provided.

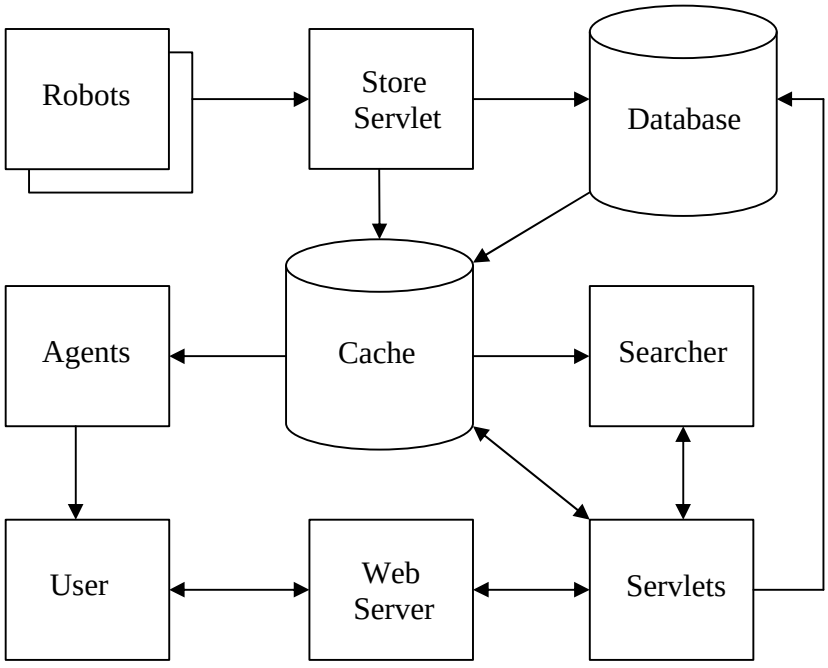

Figure 2. High level system architecture of the prototype search engine.

The collecting of new ads is done by several distributed robots. The robots visit a number of pre-programmed websites where they collect specific information about the ads. The collected ads are sent to the store servlet which stores them into the database. The store servlet also builds indexes

of the ads and their keywords and stores this information in the cache. The cache holds all the relevant information from the database in the main memory. The agents are programs that help users find what they are looking for by informing them when ads they are interested in have arrived. The web server presents a user interface to the users. The servlets provide the web server with dynamic data and handles all communication to the lower layers of the system. The searcher is responsible for handling search queries and is called by the servlets in behalf of the users whenever a search has been executed.

3.4 Implementation

All software in the system is implemented in Java (SDK 1.4.2). The system uses 160 Java classes which are divided into 22 base classes, 34 Java Server Pages (JSPs), 20 Servlets and 84 Threads. Even though there are a lot of threads, only four threads are daemon threads, i.e. run constantly in the background. These threads are the MailDaemon (handles emails), the RobotDaemon (handles robots), the ExpireDaemon (handles expired ads) and the AgentDaemon (handles agents). The rest of the threads are subclasses of the abstract classes RobotThread and ExpireThread.

There is one RobotThread and one ExpireThread for every site that is indexed. The RobotThreads are started by the RobotDaemon every five minutes. They have short lifecycles; they collect new ads from their sites and then they die. The ExpireThreads are started by the ExpireDaemon once every night. They have a bit longer lifecycles; they check every ad that is indexed by the system from their site, and then they die.

3.4.1 Robots

I was going to call them crawlers, but since they will visit a pre determined number of sites over and over, a more suitable name for them is ‘robots’. Now, how are these robots going to collect the ads from every ad site in Sweden? There are two possible solutions to this problem: 1) I could ask each ad site to give me permission to access their database or make them create a custom page where the robots could find the information they need or 2) The robots could collect the ads directly from the existing pages of the ad sites.

The first solution would require minimum work for me, but it is not a scalable solution since it would require that I get a thumb up from every site, which is not very likely. I therefore went for the second approach. This could, however, violate some legal aspects in forms of copyright if it isn't done with great caution. I've done a thorough research about these aspects with the help of some experts and I'm convinced that this is legal, but that's beyond the scope of this report.

The websites from which the robots will collect the ads have a similar structure. They have a search engine where you can search for ads, and the search results are displayed as a list with short info about the ads and links to pages with detailed information. Even though they are similar in this sense, the code is very different, and therefore the robots need customized instructions for every site.

First I implemented a lot of methods for parsing and extracting information from a HTML page. These methods were compiled into a class 'Tools', which is used by all the robots. This example shows some HTML code and the Java code for extracting the text 'Volvo 850' from that HTML code:

<TABLE width="100%"><TR><TD>Volvo 850</TD></TR></TABLE>
Tools.skipTo(is, "<TD>"); String text = Tools.readTo(is, "</TD>"); // is = input stream

All robots are subclasses of the abstract class RobotThread, which is a subclass of Thread. The robots collect ads by opening HTTP connections to the websites and extracting the information about the ads. All common functionality among the robots, such as opening HTTP connections, is stored in the RobotThread class. The actual extraction is different for each site and therefore this functionality is located in the subclasses of the RobotThread. Thus, there is one robot for each site that is indexed.

The robots are quite different from the web crawlers of a typical search engine in the sense that they don't collect whole pages. Instead they sweep through the pages' HTML code and extract only the relevant information about the ads that is needed. To reduce the load on the servers, the robots only read information that is necessary; if they stumble across an ad that already has been collected they will close the connection. To get new ads in a relatively fast pace without flooding the servers, the robots re-visit the sites every five minutes. This is controlled by the RobotDaemon, which is a daemon thread that runs in the background. The RobotDaemon starts

the RipperThreads every five minutes. Before starting the threads it checks the state of the threads from the previous run; if they are timed out or some other problem occurs, it takes proper actions, such as closing the connection.

A few sites offer the ability to change an ad after it has been published. I didn't implement functions that keep track of updated ads because this is very rarely used; in most cases, if an ad needs to be changed, the old one is removed and a new one is added. However, the ads do expire, and functions had to be implemented to keep track of expired ads in order to keep the database up to date, i.e. no dead links. I solved this by adding another type of robots; expiration robots. These robots are subclasses of the abstract class ExpireThread. Since each site shows its expired/removed ads differently, there is one expiration robot for each site. The expiration robots visit all the ads from their respective website to check if they are still available. The ExpireDaemon is the thread that controls the expiration robots. The robots are run once every night since traffic is low at that time.

The first version of the robots was quite unstable. Sometimes sites go down, their pages change structure or they have a slow response time. This sometimes caused the robots to hang or crash. To solve this I added timeouts to all the robots and made them report structural changes in the pages. This made the system very stable.

3.4.2 Store Servlet

The store servlet provides both the storage and the indexing functionality. When a robot has collected an ad it sends the information to the store servlet. The information consists of a siteID, the ad's URL, headline, text, category, type, region, and if it has a picture. The category, type and region are mapped to the search engine's corresponding IDs. All information about the ad is saved in the database where it will be given a unique ID. Since the database will be very small in size compared to a typical search engine, there is no need to use a compression other than the one built into SQL 2000.

The indexing of the keywords works in the same way as of a typical search engine, i.e. by building hash tables. The keywords consist of the words located in the ad's headline and text. There is no need to make a distinction between these words and they are therefore treated as

equal. Insignificant words like ‘and’, ‘if’ and ‘I’ will be ignored since they will be of no use in this search engine. Font size and such are also of no importance. However, the positions of the words are stored. The reason for this is to support exact phrase searching. For example, if someone searches for the phrase “second hand”, the searcher will first get all ads containing the word ‘second’ and their position in the text. Then it will check if these ads also have the word ‘hand’ at a position that corresponds to the first position incremented by one.

A Java object of the ad is created. The object, which is an instance of the class Ad, contains all information about the ad. A Java object of each keyword is also created. These objects are all instances of the class Word. A Word has two variables; a reference to an ad object and an integer for the word’s position in the text.

Two indexes will be built: an ad index and a word index and, similar to a typical search engine’s document index and word index. The ad index uses the ad’s ID as key and a reference to the ad object as value. The word index uses the hashed values of the keywords as keys and references to the word objects as values. Both these indexes are stored in the cache. A more thorough description is provided in the ‘Implementation’ chapter. Read the description of the cache for more information about how the indexes are implemented.

The indexing of the other criteria, such as category, type and region is done in correlation with creating the ad index; the ad index also contains a linked list with all the ads, and since the criteria are only integers, an iterate search for them through the list will be very fast, and there is very little to gain to build hash tables.

The biggest memory consumption is the ad’s text since it can contain several thousand characters. Only a few lines of text need to be displayed in the search results, because the users will be redirected to the ad’s origin site. Since the text is already indexed (i.e. made searchable by keywords) there is no need to keep the entire text in the non local ad objects (i.e. ads that have been collected by the robots). Therefore these ads’ text is reduced to a maximum of 150 characters. This will, however, not be applied to the local ads since this information needs to be displayed when someone clicks on a local ad.

3.4.3 Artificial Intelligence

After running the system for a while, i.e. letting the robots collect a few thousand ads, I found that the database had a lot of duplicate ads. The reason for this is that people want to get as much exposure as possible and therefore they submit their ad to several different ad sites. Sometimes they even place the same ad multiple times on the same site. This problem had to be fixed since it degraded the service severely; in some extreme cases when you executed a search, as much as half of the hits could be the same ad.

To solve this problem I equipped the store servlet with a tweaked Bayesian AI [2], [12] filter that every new ad had to pass before they were stored in the system. The method for this works as follows. When the store servlet has extracted the keywords from the ad, they are sent to the filter along with the criteria of the ad. The filter checks which ads possess any of these words by conducting a Boolean OR search with all the keywords. To reduce the load on the server a maximum of 50 keywords are used. This is enough since this is 50 unique and significant words; about 3/4 of the words in one ad's text are not unique nor significant, which makes the filter handle ads with up to 200 words without skipping any keywords, which is about 99% of the ads. For the remaining 1%, 50 keywords will be enough to determine if there is a duplicate.

Each ad that matched the search is given one point for each word it possesses. A threshold is calculated based on how well the criteria (category, type and region) are matched and the number of keywords in the ad. The worse the criteria match, the higher the threshold will be. Some of the criteria are also mandatory.

The number of words is compared to a range of number of words. For each different range there exists a percentage that will be multiplied with the number of words. For example, if an ad has 13 significant words, it will be in the range 11-15 words. That range has a percentage of 85% if all criteria match, and a bit higher if some criteria are different. This means that the threshold will be 13 x 0.85 = 11 for ads with the same criteria, which means that an ad with matching criteria has to have at least 11 or more matching keywords to be considered a duplicate. The total number of

keywords in the ad is also taken into account. If the total number of keywords in the found duplicate differs too much from the ad, it is not considered a duplicate. This is because ads with a large amount of keywords have a high probability to match all keywords from ads with much less keywords, and therefore be mistaken for duplicates.

At first all criteria had to match. The problem with this solution is that there exists a lot of "spammers" who submit the exact same ad in different categories or regions just to get more exposure. By just keeping the type criterion mandatory and by increasing the thresholds for ads that don't match the other criteria, I was able to remove all "spam ads" and still keep the same accuracy of the filter. I did some minor tweaking of the thresholds and I am still today astonished of how well this filter works. It manages to remove 99.9% of all duplicate ads.

3.4.4 Database

The database contains all the data in the search engine. It is implemented as an SQL 2000 database. The database has 16 tables and 48 stored procedures. The database has tables for each of the different data types. For instance, there are tables with names like Ads, Agents, Categories, Types, Regions and Keywords. The database works like a real time backup system; it is only accessed when data has to be changed or when the cache is loaded for the first time. When the cache is loaded, all relevant data is transferred from the database to the main memory. From then on, all read operations are done in the cache.

There is no need to use a compression scheme other than the one built into SQL 2000 since the database will be much smaller than a typical search engine's repository. The current size of the database (~200.000 ads) is 500 MB. All data structures in the system can be rebuilt from the database.

3.4.5 Cache

The cache holds the relevant parts of the database in the main memory. The purpose of the cache is to optimize the overall performance of the search engine. The cache is implemented as a

singleton class, which means that there is only one instance of that class. All data in the cache is kept in different data structures [10] – each optimized for how the data will be accessed.

The keywords and ads are stored in a '*HashList*', which is a special type of hash table that I have designed. It is described more thoroughly in the chapter 'Optimizations'. Other data such as categories, types and regions are stored in fixed size arrays. All insignificant words are kept in a HashMap. The agents are kept in a Vector. The cache also keeps the hashed value of the URLs from the most recent collected ads and the most recent duplicate ads in a Hashtable. This is so that an ad won't be indexed again nor be checked again by the AI filter the next time the robots run.

Since all data in the cache is kept in main memory, bottlenecks like disc seeks are avoided. Also, communication between the web server and the database is only needed when data has to be changed, because all read operations are done in the cache.

Even though the cache consumes a lot of memory (~200 MB per 100.000 ads) it also saves memory; no data needs to be fetched from the database and no temporary objects needs to be created since references (pointers) can be passed between the various parts of the system. Also, full search results can be stored in the memory as session variables in forms of arrays with references.

Before starting the system, the cache needs to be loaded. The cache loads its entire data structure from the database. This operation takes about three minutes with a database of 200.000 ads. This is only needed once; all updates of the data will occur both in the cache and the database. This is controlled by the servlets, which update both the database and the cache whenever a write operation occurs (see figure 2).

3.4.6 Agents

The agent service helps users find ads they are interested in. A user can create an agent by specifying what types of ads he is interested in, and then the system will search for these ads as they arrive to the system. The users can specify keywords and all the other criteria the search engine supports.

Each agent is an instance of the class Agent. It contains a creation date and a user's search criteria, keywords, email and password. The password is used when a user wants to modify or remove his agent. All agents are stored in a vector in the cache.

The agents are controlled by the '*AgentDaemon*', which is a daemon thread that runs in the background. The thread sleeps until an ad arrives to its queue of ads. When a new ad arrives, the thread wakes up and compares the ad with all the agents in the vector. If there is a match, it will alert the user by sending him an email with a link to the ad. The AgentDaemon also removes expired agents, and sends expire notification emails to the users whose agents are about to expire.

The agents guarantee that users will be notified of a match within five minutes after an ad has been published to any of the indexed sites. This is because five minutes is the exact timeframe in which new ads arrive.

3.4.7 Web Server

The web server is Microsoft's Internet Information Server (IIS) 5.0. The web server presents the web interface to the users' browsers. The web server communicates with the servlets through a servlet engine (Resin 2.1.6). The servlets provide the web server with the dynamic data.

3.4.8 Servlets

The servlets consist of Java Servlets and Java Server Pages (JSP). They are used in order to create dynamic web pages. Servlets and JSPs are basically the same thing; JSPs are automatically compiled into servlets. The difference is that JSPs are preferable when the major part of the code is HTML code.

The web server, the servlets and the cache can be thought of as three layers; the cache is the lowest layer with all the raw data, the servlets are the middle layer that do all the communication between the other two layers and the web server the highest layer that presents the data to the users. See the chapter 'Use cases' for a detailed description of such communication.

3.4.9 Searcher

The searcher [9] is the core part of the search engine that does the actual searching. It is implemented as a class with static methods that contain all search algorithms. These methods are invoked by the '*SearchServlet'*, which is the servlet that handles search queries from the users. Before the SearchServlet calls the searcher, the search query string is formatted; the words are converted to lowercase and all illegal characters are removed. Also, if the query contains insignificant words that the search engine doesn't index, such as 'if' and 'or', they are discarded. The remaining words are separated based on their Boolean operators. The operators that are supported are AND, NOT, OR and EXACT PHRASE. The words will be separated according to the first three operators; the last operator is a sub-operator of the other operators, and therefore the words within the exact phrase are treated as one word and separated according to the other operators.

If no operator is present, the AND operator will be default. This is because the majority of all people doesn't bother to read the instructions of how to use the search engine, or are unfamiliar with Boolean operators and such, and they don't want to receive search results on a "Volvo 240" if they search for a "Volvo V70".

The searcher takes the separated keywords and search criteria as parameters and returns a search result in form of an array with references (pointers) to the matching ads. The ads are sorted by best match and date, in that order. The search result is stored temporarily as a session variable on the server. This way, a user can iterate back and forth within the search result without executing any more searches.

The search engine supports three types of searches: keyword searches, criteria searches, e.g. 'cars', 'for sale', 'in Stockholm', or combined keyword/criteria searches.

A criteria search works as follows. The searcher iterates through the list of ads and selects those ads that match all criteria. Since all criteria need to match, all matching ads are equal in rank and the result is therefore sorted by date.

A keyword search works as follows. All ads that match the keywords are collected. To do this efficient, it first gets all the ads that match the AND words, and from these ads, remove those that match any of the NOT words. From the remaining matches, the ads are given points for each of the OR words that are matched. The ads are sorted by points; the ads that have the same amount of points are sorted by date.

A combined search is similar to a keyword search. The difference is that the matching ads are filtered out according to a criteria search before the OR words are given points.

3.5 User Interface

The search engine uses a web user interface. The interface is created by different JSPs that produce dynamic HTML code. Figure 3 shows the start page of the search engine. The page is divided into five different frames (from top to bottom): top frame, menu frame, search frame, main frame and bottom frame.

The top frame shows the logo of the search engine and a commercial banner. The commercial banner will change depending on what the user searches for, so that the user will receive more targeted commercials.

The menu frame contains links to different services the search engine offers. For instance, there is a link to a page where you can create your own ad and a link to a page where you can create your own agent that helps you find what you are looking for. All links open in the main frame.

The search frame shows the search interface. Here you can specify what type of ads you are interested in by selecting among different criteria. You can select category, region and type with drop-down menus. You can select if you want ads in nearby regions and if the ads must have a picture with checkboxes. In the input field you can add keywords that are compared with the ads' headline and text. Boolean operators can be used with the keywords to narrow down the ads even more.

The main frame shows the 50 latest ads in descending date order, and some links to more ads. The ads are shown by date, picture, headline, category, type, region and source. The source is the name of the ad site from which the ad was collected. If you click on an ad's headline, the ad will open on the original site in a new browser window. When a search has been made, the search result will be presented in this frame instead of the latest ads.

The bottom frame shows the total number of visitors today, the number of visitors at the moment, and the number of ads currently indexed in the system.

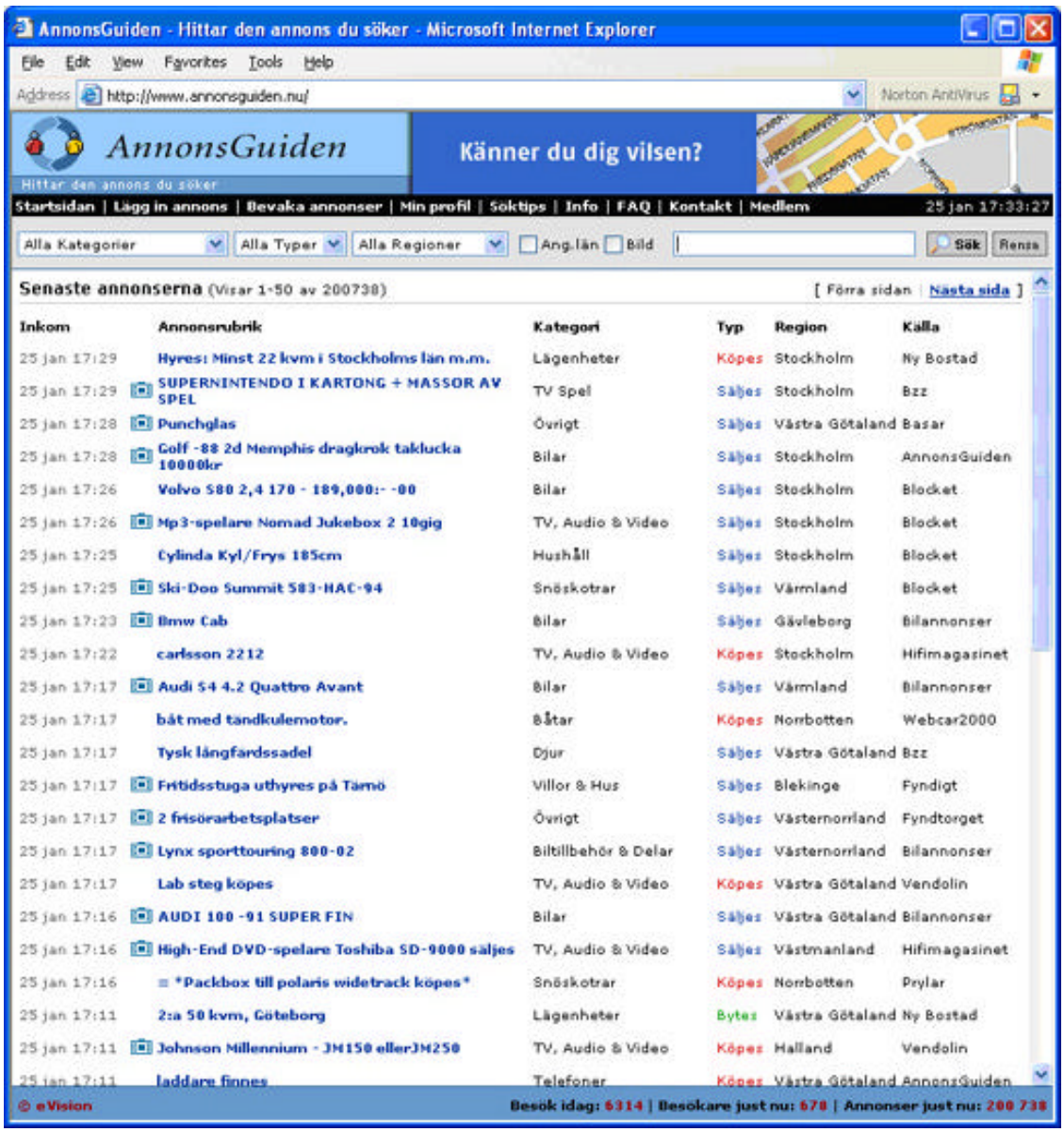

Figure 3. Web interface of the prototype search engine. The page shows the latest 50 ads.

The interface is built by a number of Java Server Pages (JSPs). Each of these files has a reference to the cache. The JSPs get their reference to the cache by calling its getInstance() method. To generate the different data they call different methods from the cache. For instance, there is a method called getCategories() which returns all the categories in form of a reference to an array with category objects. The category class has two variables; ID (int) and name (String). The drop-down menu with categories is created by iterating through the array and printing the result.

3.6 Use Cases

3.6.1 Searching

In this example a user searches for cars for sale in Stockholm. He has added the keyword ‘volvo’ with the Boolean operator AND, which means that this word must be a part of the ad’s headline or text. The keywords ‘855’ and ‘850’ are added with the Boolean operator OR, which means that they don’t have to be a part of the text, but those ads that contain them will be in the top of the search result. The word ‘white’ is added with the Boolean operator NOT, which means that all ads that contain this word are excluded from the search result.

Figure 4 shows the search result. Notice that the commercial banner has changed to a topic that matches the user’s search.

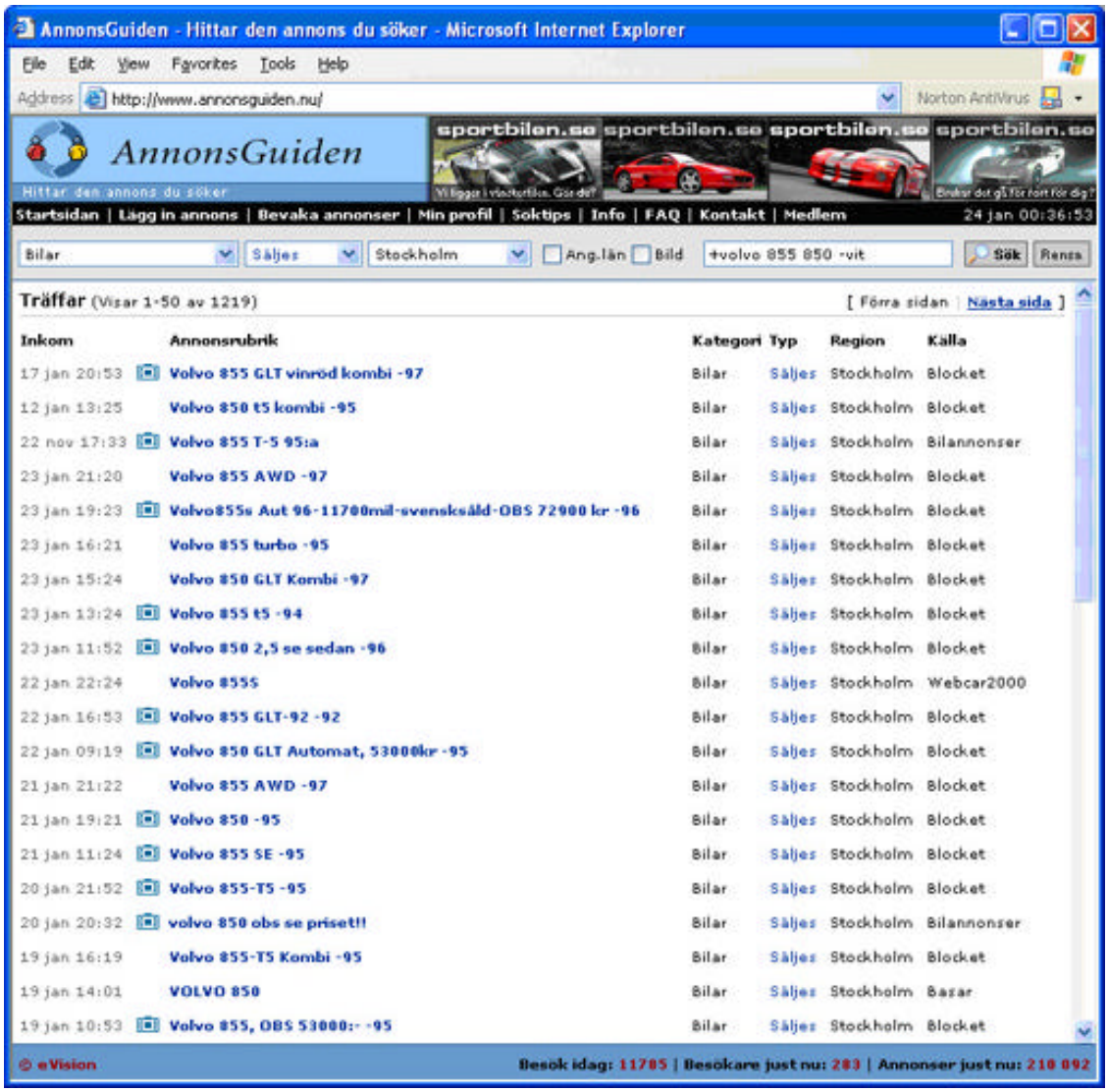

Figure 4. Search result after searching for cars for sale in Stockholm with some keywords.

The whole scenario works as follows:

1. The search frame is generated by a file called search.jsp. The frame consists of a HTML form with different parameters, such as categories, types and regions. The values of the parameters are their corresponding IDs in the database. The user selects different parameters such as criteria and keywords and presses the submit button. The page has the main frame as base target, which means that the response will show up in the main frame.
2. The parameters are sent to the SearchServlet from the user's browser with a HTML POST method.

3. The SearchServlet extracts and validates the parameters. It creates a session variable called ‘commercial’, which is an array with the submitted categoryID, typeID and regionID. This variable is later used to present targeted commercials. If logging is enabled (which is a parameter that can be set by the administrator), the SearchServlet writes the search data to a log file called ‘search.log’, which is used for statistical analysis. The keywords are sent as a string called ‘keywords’. The SearchServlet extracts the keywords from the string and places them in different arrays based on their Boolean operators. It also removes insignificant words such as ‘if’ and ‘the’, because they are not indexed by the system. Now it calls a static method of the Searcher class with the search criteria and the keyword arrays.
4. The Searcher class contains all methods for turning search queries into search results. It calls the Cache’s getWordIndex() method. This method returns a reference to a Hashtable that maps words to lists of ads that possess these words. The Searcher calls the Hastable’s get() method with each keyword, and for each keyword it receives an object in form of a HashList (read chapter 3.7.4 for details about this class) that contains all the ads that possess that keyword. Each object in the HashList contains a reference to an ad and a number that represents the word’s position in the ad’s text. The number is only used when EXACT PHRASE keywords are present. The Searcher first chooses the HashLists that maps the AND operator keywords. It takes the HashList that has the least number of ads. In this case, ‘volvo’ is the only AND keyword. The Searcher iterates through the HashList and checks which ads that match the criteria, e.g. by calling ad.getCategory().getID() and compare that ID with the categoryID that was submitted, etc. Those ads that match the criteria are checked that they don’t contain any words matching the NOT operator keywords. It does so by calling the get() method of the HashLists that represent the NOT words. The method takes a reference to the ad as a parameter and returns null if the ad didn’t exist in the HashList or a reference to the same ad object if it did. If the word wasn’t present the ad is added to an ArrayList that contains the search result. The list is already sorted by date, however if OR words are present the list needs to be sorted by best hits, i.e. the ads that possess the most OR words (e.g. 855 and 850) will be given the highest rank. The OR words are checked in a similar fashion as the NOT words. For each word that is present the ad is given one point (the ArrayList contains objects of the class SortAd; it implements the Comparable interface and has a

reference to an ad and an integer that represents the points). The ArrayList is sorted with the Arrays.sort() method. The ArrayList is returned to the SearchServlet.

5. The SearchServlet stores the ArrayList in a session variable ‘result’. It redirects the user to a JSP file called ads.jsp with a parameter ‘result’ that tells the JSP that it should show the search result instead of the 50 latest ads.
6. The ads.jsp file checks the ‘result’ parameter. If it exists the JSP gets the session variables ‘commercial’ and ‘result’. The commercial array is compared to the currently stored commercial banners, and if there is a match the top frame is reloaded (with a JavaScript call).
7. Topframe.jsp checks the same session variable and shows the corresponding banner. If several banners are mapped to the same criteria, it randomly chooses among them.
8. Ads.jsp shows the total number of hits and the span of the search result that is displayed; by default it shows the 50 first ads from the search result. It does so by iterating through the ArrayList ‘result’. It also adds links to more ads in the result.

3.6.2 Creating an ad

In this example a user creates an ad. Figure 5 shows the interface for creating an ad. After submitting the ad the user receives an activation email where he clicks on a link that makes the ad active in the system.

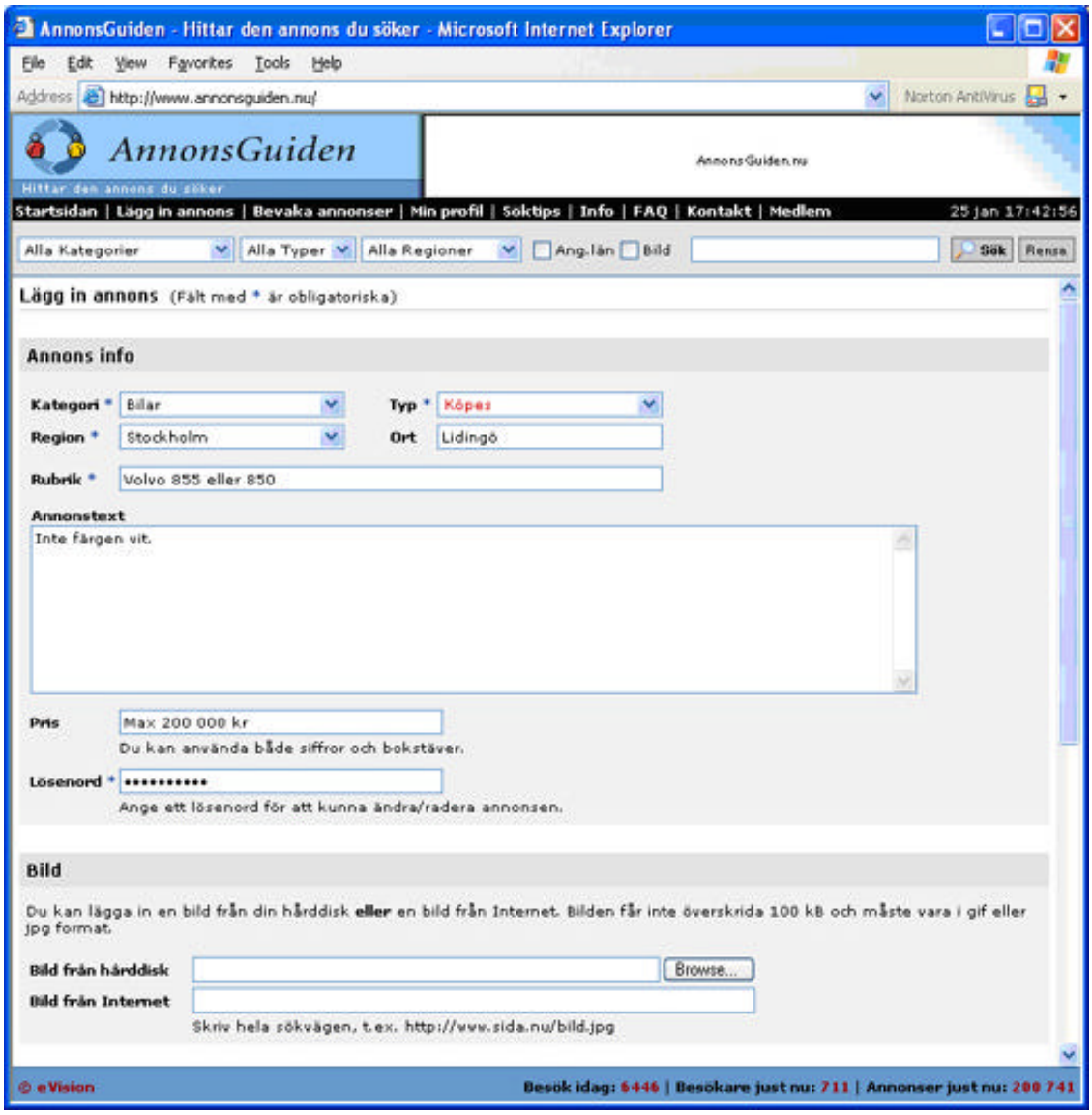

Figure 5. Web interface of the page where a user can create an ad.

The procedure for this works as follows:

1. The user clicks on the “create ad” link in the menu frame. The link’s anchor points to the file ‘ad.jsp’. The menu frame has specified the main frame as base class in the HTML code, which means that all links will be opened in the main frame.
2. The user’s browser requests the ad.jsp file from the web server with a HTTP GET call.
3. The web server recognizes the .jsp file type and redirects the call to the servlet engine.
4. The servlet engine executes the JSP file ad.jsp.

5. The JSP creates the dynamic HTML code which builds the interface as seen by figure 5. The code is generated the same way as described in 3.5.
6. The user fills out the form and presses the submit button. The form's 'action' value is set to 'AdServlet'.
7. The user's browser sends the form's parameters to the AdServlet with a HTTP POST call.
8. The web server recognizes the URL and redirects the call to the servlet engine.
9. The servlet engine executes the servlet AdServlet.
10. The AdServlet receives the parameters and validates them. If some parameters are missing or are incorrect the servlet redirects the user back to ad.jsp with some error text. The servlet extracts all unique keywords from the ad's headline and text. A randomly key of 10 characters is generated by the servlet. The servlet requests a database connection from the ConnectionPool.
11. The ConnectionPool is a singleton class which holds a number of pre-allocated database connections. The ConnectionPool finds a free connection, marks it as busy and returns it to the AdServlet.
12. The AdServlet makes a database call to the stored procedure 'AddAd' with the ad's parameters, keywords and key.
13. The JDBC driver handles the call between the JVM and the database.
14. The database executes the stored procedure; it puts the ad under the table 'Ads', with the column 'Active' set to 0, which means that the ad is inactive in the system. The keywords are stored in the table 'Keywords', which has two columns; AdID and Keywords. The keywords are separated with spaces and stored as an nvarchar. The key is added to the table 'InactiveAdKeys', which maps keys to ad IDs. The stored procedure returns the unique ID that was automatically created for the ad.
15. The JDBC driver returns the result to the servlet.
16. The AdServlet receives the unique ID and returns the database connection to the ConnectionPool.
17. The ConnectionPool marks the connection as free.
18. The AdServlet calls Mail.createAdActivationMail() with the user's email address, the key and some other parameters such as the user's name and the ad's headline.
19. The Mail class has static methods for creating HTML formatted mails. The methods create Email objects. Email is a class for holding email relevant information. In this case the email is a confirmation email to the user with a link that activates his ad. The link's

anchor points to the AdServlet with the randomly generated key as parameter. The reason for this activation email is to verify that the user's email address is correct. A reference to the email object is returned to the AdServlet.

20. The AdServlet calls the MailDaemon's sendMail() method with the email as parameter.
21. MailDaemon is a daemon thread that handles all emails. The MailDaemon sends the email content to the SMTP server via SMTP calls.
22. The SMTP server sends the email to the user's email address.
23. The AdServlet generates a page where the user is told that an email with an activation link has been sent to his email address.
24. The web server sends the page to the user's browser.
25. The user receives the message and opens his email client. He receives the email and presses the activation link. The link opens the user's browser, which requests the AdServlet from the web server with a HTTP GET call. The key is sent in the query string of the same call.
26. The web server recognizes the URL and redirects the call to the servlet engine.
27. The servlet engine executes the servlet AdServlet.
28. The AdServlet receives the key and requests a database connection.
29. The ConnectionPool finds a free connection, marks it as busy and returns it to the AdServlet.
30. The AdServlet makes a database call to the stored procedure 'ActivateAd' with the key as parameter.
31. The JDBC driver handles the call to the database.
32. The database executes the stored procedure ActivateAd. The procedure selects the AdID from the table 'InactivatedAdKeys' that has the corresponding key. If an AdID was found the procedure updates the table 'Ads' by setting the column 'Active' to 1. The procedure returns the data about the ad from the table Ads and the table Keywords.
33. The JDBC driver creates a ResultSet object and returns it to the servlet.
34. The AdServlet extracts the ad's data and keywords from the ResultSet object. The servlet returns the database connection to the ConnectionPool.
35. The ConnectionPool marks the connection as free.
36. From this data, the AdServlet creates a Java object and calls the Cache's addAd() method with the ad object as parameter.
37. The Cache stores the ad in the ad index, which is a HashList that maps adIDs to ads.

38. The AdServlet calls the Cache's addKeywords() method with the keywords and a the ad object as parameters.
39. The Cache stores all keywords in the word index, which is a Hashtable that maps words to Hashlists with ads. The Cache adds the ad and its keywords to the AgentDaemon's queue.
40. The AgentDaemon wakes up. It requests the Caches' getAgents() method.
41. The Cache returns a reference to the Vector containing all the agents.
42. The AgentDaemon compares the ad(s) in the queue with the agents; if there is a match the AgentDaemon calls the MailDaemon's sendAgentHitMail() method.
43. The AdServlet generates a confirmation page with a link to the ad.
44. The web server sends the page to the user's browser.

3.6.3 Creating an agent

In this example a user creates an agent. Figure 6 shows the interface for creating an agent. The agent service is only available for members; hence there is no need to send activation mails since the members' emails have already been verified to be correct in correlation with signing up as members.

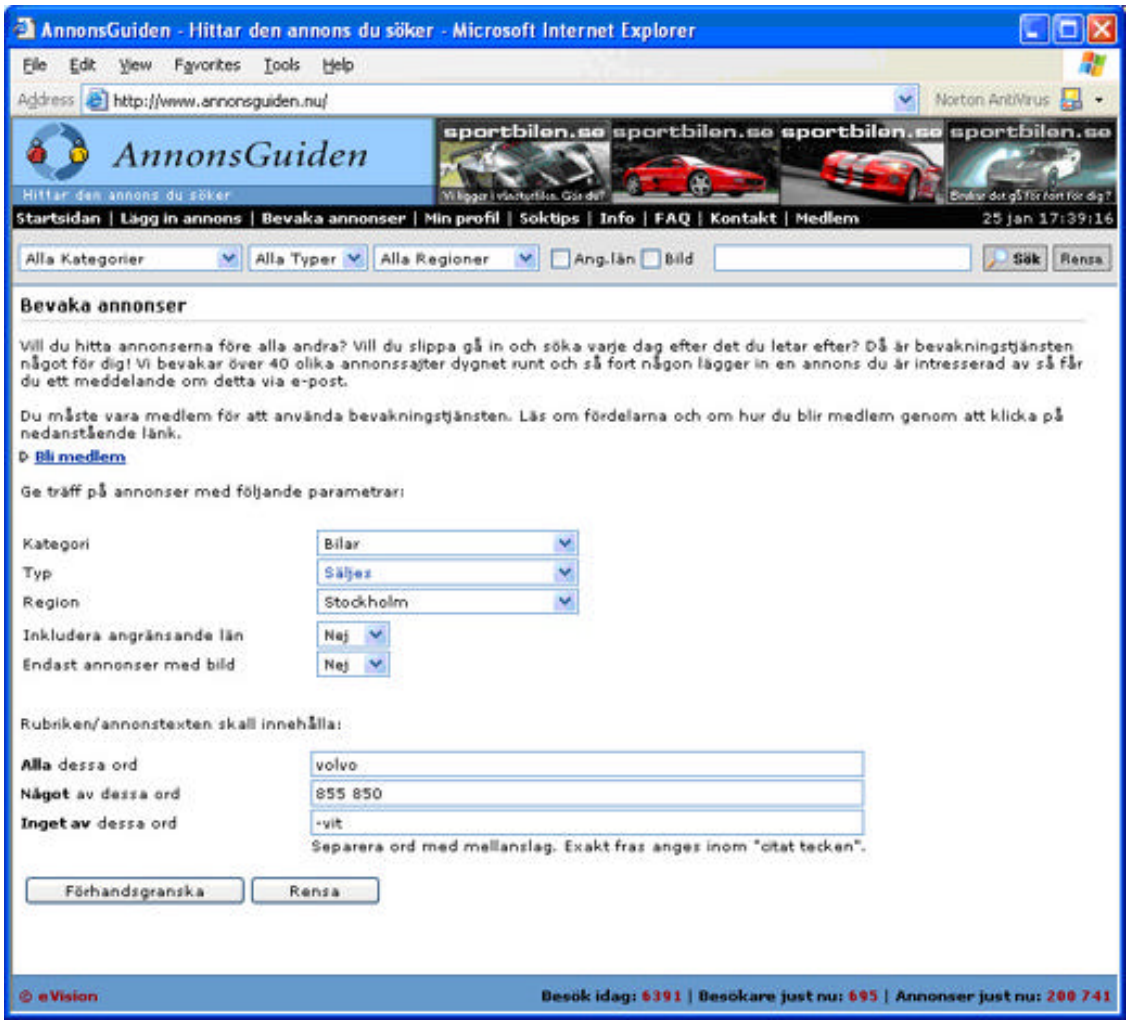

Figure 6. Web interface of the page where a user can create an agent.

The procedure for this works as follows:

1. The user clicks on the “create agent” link in the menu frame. The link’s anchor points to the file ‘agent.jsp’. The menu frame has specified the main frame as base class in the HTML code, which means that all links will be opened in the main frame.
2. The user’s browser requests the agent.jsp file from the web server with a HTTP GET call.
3. The web server recognizes the .jsp file type and redirects the call to the servlet engine.
4. The servlet engine executes the JSP file agent.jsp.

5. The JSP creates the dynamic HTML code which builds the interface as seen by figure 6. The code is generated the same way as described in 3.5.
6. The user fills out the form and presses the submit button. The form's 'action' value is set to 'AgentServlet'.
7. The user's browser sends the form's parameters to the AgentServlet with a HTTP POST call.
8. The web server recognizes the URL and redirects the call to the servlet engine.
9. The servlet engine executes the servlet AgentServlet.
10. The AgentServlet receives the parameters and validates them. If some parameters are missing or are incorrect the servlet redirects the user back to agent.jsp with some error text. The servlet extracts all unique keywords from the ad's headline and text. The servlet requests a database connection from the ConnectionPool.
11. The ConnectionPool finds a free connection, marks it as busy and returns it to the AgentServlet.
12. The AgentServlet makes a database call to the stored procedure 'AddAgent' with the agent's parameters and key.
13. The JDBC driver handles the call between the JVM and the database.
14. The database executes the stored procedure; it puts the agent under the table 'Agents'. The stored procedure returns the unique ID that was automatically created for the agent.
15. The JDBC driver returns the result to the servlet.
16. The AgentServlet receives the unique ID and returns the database connection to the ConnectionPool.
17. The ConnectionPool marks the connection as free.
18. The AgentServlet creates a Java object of the agent containing the data and ID, and calls the Cache's addAgent() method, with a reference to the agent object as parameter.
19. The Cache stores the agent in the vector 'agents'.
20. The AgentServlet generates a confirmation page to the user.
21. The web server sends the page to the user's browser.

3.7 Optimization methods

There are quite a few ways of increasing the performance of a system. Keeping in mind that the system will be used by a large number of simultaneous users who mainly execute searches, I tried to identify the biggest bottlenecks and eliminate them.

3.7.1 Cache

One of the biggest bottlenecks of today's computer systems is the hard drive. Even high performance RAID disc clusters don't come near the performance of a main memory. Therefore I tried to use as little hard drive as possible and instead use the main where possible, which lead to the development of the '*cache*'. The cache holds the entire relevant part of the database in the main memory. Whenever an update occurs in the cache, the database will be updated as well; a so called '*write through cache*'.

The cache consists of different containers; each one is chosen to be optimized for its data. For instance, hash tables are used where individual data needs to be fetched from a large container, arrays are used for data that is frequently iterated, and if the size is known in advance, fixed size arrays are preferable. I created a special container called '*HashList*', which I will describe in detail later, which is used as a container for all the ads and keywords.

It may seem a bit overhead to optimize such things as disc seeks when the clients will be connected to the server via the Internet, but if you consider the effect of thousands of simultaneous users processing search queries, every millisecond you can reduce will have a great impact on the overall performance.

3.7.2 Connection pooling

Sometimes however, data has to be written to disc because a main memory is not a permanent storage medium. An effective way of storing data to disc is by using a SQL database. There are however a few bottlenecks here that can be avoided. First of all, the database has to be built in an effective manner; that is using indexes at the right places, keeping the data types as small as possible, avoid varchars where not necessary, etc. Assuming we done all this correct, there is still one big bottleneck that can be avoided.

A Java Virtual Machine (JVM) can communicate with a database through the JDBC (Java DataBase Connectivity) API, which translates incoming JDBC calls to outgoing database native protocol calls. The actual implementations are provided by third party vendors in form of JDBC drivers. Before any calls can be sent, a connection has to be established between the JVM and the database. Once the connection has been established, it can be used for as long as you want.

The biggest bottleneck of database connectivity is the actual setup time of this connection. I did some tests and it takes about 250 ms for it to complete in my test environment. The positive thing is that this delay can be totally eliminated by creating a '*connection pool*'. Since a connection only has to be established once, a pool of a number of pre-allocated connections can be used. Whenever a connection is needed, it will be handed out by the pool and be marked as busy. This will also reduce the number of needed database connections, since one connection can be used by several users (threads) assuming they don't do the update at the exact same time.

3.7.3 Avoid synchronization as much as possible

Since this is a multithreaded system, synchronization is needed to keep the shared resources in a safe manner, i.e. avoid deadlocks and such. Synchronization will reduce the performance and therefore it is very important to use it only where it is needed. Old Java classes such as Vector and Hashtable have built in synchronization for all their methods. If no synchronization is needed, then newer classes such as ArrayList and HashMap should be used instead, since they are unsynchronized and the synchronization is left for the programmer to take care of if needed.

3.7.4 HashList

The main container for all ads needs to be as optimized as possible because it will be the most frequent used container. I developed an own class for this called '*HashList*'. It has the features of both a hash table and a linked list. A hash table is fast when you want to look up a specific object, but slow for iterating through its objects, and vice versa for a linked list. Also, if you iterate through a hash table, there is no guarantee in which order the objects will appear, while iteration through a linked list will return the objects exactly as they were inserted into the list.

The HashList contains a built in HashMap and a special designed linked list. The linked list contains objects (ads) with references to the next and previous object in the list. I did some benchmarks where I compared the HashList with the native Java classes 'HashMap' and 'LinkedList'. In this benchmark I created 500.000 objects and iterated through them. The following table shows the allocated memory needed for the classes to hold the objects (excluding the memory allocated for the objects themselves), and the time taken for a complete iteration through the objects.

Class	Memory (MB)	Time (ms)
HashList	26,1	63
LinkedList	18,2	125
HashMap	13,8	156

The total memory if you combine the two native classes will be 32 MB. Since the HashList contains both a hash table and a linked list it outperforms the two native classes in both speed and memory consumption. The reason for this is that I could avoid much of the overhead in the native classes by building my own class very slim; I only implemented the things needed in a way that was optimal for the usage in this project.

Another very big issue with the HashList is that I managed to make the iteration of it safe and unsynchronized even in a multi threaded environment. This is impossible for the native classes since they will automatically throw a ConcurrentModificationException if a write operation occurs while another thread iterates through the list if it isn't synchronized. I utilized the garbage collector for this. The garbage collector is a part of the Java Virtual Machine and takes care of all memory deletions. It checks if an object cannot be reached through any references, and if so it is

deleted. By changing the references for a removed object in such a way that the object keeps its references and all references to the object are removed, the object will remain in memory as long as some thread have a reference to it, and thus, if a thread should be in the middle of an iteration at the same location as this object, the object will not be deleted since the iterating thread will have a reference to it, and the iterating thread will be able to continue its iteration since the object has kept its references to the next object in the list. The write operations of the HashList need to be synchronized though, since two or more write operations cannot occur simultaneously because that could make the references point in an undetermined way.

3.7.5 Reduce the size of the result page

No matter how good your hardware is or how much you have optimized the search engine, in the end what you will send to the end user is a page with the search results. If the end user has low bandwidth, e.g. a telephone modem, every kilobyte you can reduce from the result page will have a great impact on the time it will take to download the page. For instance, I managed to reduce the result page from 50kB to just 7kB, which means that the load time for a user with a 56k modem was reduced from seven seconds to just one second.

I did this by storing as much static data as possible on the client side and only send the actual dynamic data. I stored the static data in a JavaScript file that is downloaded the first time a user visits the page, and then it will be cached on the user's hard drive. I use the same JavaScript to build up dynamic HTML pages from the data which is sent from the server. For example, the result page will just contain calls to a JavaScript function which will produce the HTML code necessary to build up HTML table structures, etc. This requires some extra processing on the client's machine, but that delay is insignificant compared to the download time, especially if the user has a somewhat modern computer, then this delay is not even noticeable.

4 Analysis

4.1 Evaluation testbed

For the tests I used the configuration that is described in section 3.2. The system held about 214.000 ads at the time when the tests were performed. I tested the functionality, the performance and accuracy of the search engine and how it behaved under heavy stress. I performed four different tests to evaluate the different aspects of the search engine:

- **Functionality test**
  This was the actual “bug” test which checked that everything works as it should. The test consisted of creating, changing and removing ads and agents, monitor that the robots collect new ads correct, that the AI removes duplicates correct, that the agents give correct hits, and that the search engine gives correct results.

- **Performance test**
  This test checks how fast the search engine can produce different search results. I conducted different types of searches and measured the time between receiving a search query by the search engine and producing a complete result set.

- **Accuracy test**
  This test checks how well the search engine gives results for advanced Boolean searches. I also compared searches on this search engine with corresponding searches on other search engines whose ads are indexed by this search engine.

- **Stress test**
  In this test I checked how the search engine behaved under heavy stress. I simulated online users with threads that executed searches.

4.2 Results of evaluation

4.2.1 Functionality test

The ads and agents were created/changed/removed correctly. To check the AI I let the robots run for 24 hours, which produced 5272 new ads. The AI classified 583 of those ads as duplicates, i.e. currently indexed in the system from some other site. The AI writes the URL of every new ad that is classified as a duplicate into a log file, along with the URL of the ad that was found to be its duplicate. I visited each of these URLs manually and compared the ads, and I'm proud to say that every one was correctly classified as a duplicate.

4.2.2 Performance test

The database held 211.448 ads when this test was performed. The JVM measures time in units of tens of milliseconds, i.e. time 0 means that the search took less than ten milliseconds to complete and time 10 means that the search took between: 10 <= time < 20 ms.

Keywords	**Criteria**	**Hits**	**Time (ms)**
Saab	-	4.421	0
Saab *AND* 9000 *AND* Turbo	-	269	0
Saab *NOT* 9000 *NOT* Turbo	-	2.239	10
Saab *OR* 9000 *OR* Turbo	-	4.421	20
Volvo	-	11.374	0
Volvo *AND* 850 *AND* GLT	-	220	0
Volvo *NOT* 850 *NOT* GLT	-	9.490	30
Volvo *OR* 850 *OR* GLT	-	11.374	50
-	Stockholm	50.789	90
-	Computers in Stockholm	1.183	90
-	Cars for sale in Stockholm with picture	4.139	90
Ford *AND* Escort	Cars for sale in Stockholm with picture	48	0

As seen by the results, keyword searches are extremely fast. They're also not affected by the number of keywords. Keywords with the AND operator are the fastest because they will reduce

the result set very rapidly. Keywords with the NOT operator are a bit slower because they will not reduce the result set as significant as the AND operator, and thus, the time will depend much on the size of the starting result set, i.e. the set before the NOT words are removed from the set. The Boolean operator OR is the slowest. The reason for this is that such a search result will not only be sorted by date but also by best possible hit, i.e. the ads that contain most OR words will be at the highest position in the search result list. For this reason, the time will depend on the size of the result set.

When a criteria search is executed, the whole list is iterated and each ad is checked if it fulfills the criteria. This takes about 90ms to complete regardless of how many criteria that are submitted. If a keyword is added to a criteria search, the search engine will first execute a keyword search and then remove the ads that doesn't fit the criteria from the result, and hence, the time will be the same as a pure keyword search.

4.2.3 Accuracy test

Keywords	Criteria	Hits	Sites
Nissan *AND* 200sx	Located in Stockholm	16	4

This test gave 16 hits distributed among 4 different sites. The first site gave 7 hits, the second 1 hit, the third 7 hits and the fourth 1 hit. A search on the first site only gave 3 hits, which all were hits among these 7. It turned out that this site's search engine missed the other 4 ads because they had a space between 200 and sx. The second site gave 1 hit according to the one that was found. The third site gave 6 hits, which all were among the 7 found. It missed the 7th ad for the same reason as the first site. The fourth site gave 1 hit according to the one that was found.

Result: 100% correct hits with even better accuracy than the original sites; it found 5 ads that the original search engines had missed.

Keywords	Criteria	Hits	Sites
T610	Phones for sale in Gothenburg area	23	7

This test gave 23 hits distributed among 7 different sites. The first site was this local site and gave 4 hits, the second 14 hits and the other 5 sites had one hit each. No need to examine the local hits. The second site gave 11 hits, where 10 of these were according to the ones that were found. The 3 other ads were missed by the original search engine because the ads had text like T610i and T-610. The ad that was missed was about 2 months old. According to the logs, it had been replaced by a duplicate ad located on another site. This ad had had later been removed from that site, hence the ad didn't turn up in the search result. The other sites gave hits according to the ones that were found.

Result: 96% correct hits. It also found 3 ads that the original search engines had missed. The ad that was missed was due to a duplicate that had been removed from the original site, which is a scenario that can happen, and thus no bug.

Keywords	**Criteria**	**Hits**	**Sites**
Porsche *AND* 911	Cars	43	6

This test gave 43 hits distributed among 6 different sites. The first site gave 29 hits, the second gave 8 hits, the third gave 2 hits, the forth and fifth gave 1 hit and the sixth gave 2 hits. The first site gave 26 hits, all corresponding to the ones that were found. The ones that the original search engine missed had text like '911/964', '911T' and '911sc'. The second gave all correct hits. The third didn't support the AND operator (it defaulted to OR), but from the 6 hits that were found with an OR search, the 2 ads that were found were also the ones that had both the words Porsche and 911.
The remaining search engines gave correct hits according to the search result.

Result: 100% correct hits. It also found 3 ads that one of the original search engines had missed.

4.2.4 Stress test

I keep a log file of all searches that have been executed on the server (the server has been running live for some time). At peak times there are about 600 simultaneous users. During this time an average of 30 searches is executed per minute i.e. one search every other second. Half of these searches are keyword searches and the other half are criteria searches. I simulated online users by

running threads that executed similar searches with the same search per user ratio. The searches were conducted directly to the search engine, thus bottlenecks like the web server could be avoided.

Simultaneous users	**Searches / second**	**CPU load**	**Increased search time**
1000	0,8	2 %	+ 0 ms
10.000	8,3	7 %	+ 0 ms
50.000	42	41 %	+ 0 ms
100.000	83	74 %	+ 0-10 ms
300.000	250	89 %	+ 0-70 ms
500.000	417	99 %	+ 0-140 ms

As shown by the results the search engine can handle up to 100.000 simultaneous users before the search time is affected. The bottleneck here is the CPU. Of course if such an amount of users would use the live system the bandwidth and the web server would probably be the biggest bottlenecks, even with a far less amount of users.

What this tests shows is that if you remove bottlenecks like bandwidth and web server by using a high bandwidth cluster of web servers, the core search engine can still be used as a central system for the whole cluster.

5 Conclusions and future work

I accomplished everything I set out to do. I have developed a search engine for second hand products available on the Internet in Sweden, which I have made available for the public. The search engine can be found at http://www.annonsguiden.nu/. I have received an enormous positive feedback from the users, because it's a great tool for finding a product that you are interested in and it saves you a lot of time and effort.

I had good programming knowledge in Java and SQL before I started, which I think was mandatory for this type of project. But even so, I have learned a lot more from all this, especially by doing all the optimizations. I have also gained in-depth knowledge about search engines, how they work, the problems involved and so forth.

As seen by the test results, this search engine can handle far more users than a normal website probably ever will receive. Scaling this system to a large amount of users would not require any changes in the core search engine; however, bottlenecks like bandwidth and the web server must be avoided. This could be done by building the system into a high bandwidth cluster [13]. Scaling this system to a larger database would require some modifications. The cache was designed to hold the entire database indexed in main memory. For the Swedish ad market, this is roughly about 2-400.000 ads. If you bought more main memory you could scale this system to a couple of million ads. If you also modified the cache so that it keeps some part of the index stored on disc, this problem would be fixed. But as the index grows infinite another problem arises; the response time. If the database has several billions of ads, the word index would be very big; some words could have a million hits. The search engine is currently written for a single CPU, and to compare such amount of data takes a lot of processing. To fix this problem, a load balancing structure could be implemented, which divides search queries into parts, where each part is processed by a different CPU in a small cluster. By using this parallelism the response time could be reduced significantly.

In my opinion, this specialized type of search engine is the next generation of search engines. By joining together different types of specialized search engines into one major distributed meta-search engine, you could outperform today's leading search engines by far. Making this efficient would probably require some form of standardized protocols and quality control of the specialized engines, which could be a major problem.

6 References

[1] Sergey Brin and Lawrence Page. “The Anatomy of a Large-Scale Hypertextual Web Search Engine”. 1998. <http://www-db.stanford.edu/pub/papers/google.pdf>

[2] Jiawei Han and Kevin Chen-Chuan Chang. “Data Mining for Web Intelligence”. IEEE Computer, November, 2002, Vol. 35, Issue 11, Pages 64-70.

[3] Knut Magne Risvik and Rolf Michelsen. “Search engines and Web dynamics”. Computer Networks, Volume 39, Issue 3, 21 June 2002, Pages 289-302.

[4] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. “Building a Distributed Full-Text Index for the Web”. 2001. <http://dbpubs.stanford.edu:8090/pub/2000-29>

[5] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. “Focused crawling: a new approach to topic-specific Web resource discovery”. 1999. <http://www.cs.berkeley.edu/~soumen/doc/www1999f/pdf/www1999f.pdf>

[6] Beverly Yang and Hector Garcia-Molina. “Comparing Hybrid Peer-to-Peer Systems”. 2000. <http://dbpubs.stanford.edu:8090/pub/2000-35>

[7] Jian Liu. “Guide to Meta-Search Engines”. 1999. <http://www.indiana.edu/~librcsd/search/meta.html>

[8] Luiz André Barroso, Jeffrey Dean, Urs Hölzle. “Web Search for a planet: The Google Cluster Architecture”. 2003. <http://www.computer.org/micro/mi2003/m2022.pdf>

[9] Jon Bentley and Robert Sedgewick. “Fast Algorithms for Sorting and Searching Strings”. 1997. <http://www.cs.princeton.edu/~rs/strings/paper.pdf>

[10] M. Goodrich and R. Tamassia, Wiley. “Data Structures and Algorithms in Java”. 1998.

[11] Jason Hunter and William Crawford. “Java Servlet Programming”. First Edition. 1998.

[12] David. D. Lewis. “Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval”. 1998. <http://www.ai.mit.edu/people/jimmylin/papers/Lewis98.pdf>

[13] Taher Haveliwala, Aristides Gionis. “Scalable Techniques for Clustering the Web”. 2000. <http://theory.lcs.mit.edu/~indyk/webdb.ps>