Development of an online reputation monitor

GJC Venter

21735514

Dissertation submitted in fulfilment of the requirements for the degree *Magister* in Computer and Electronic Engineering at the Potchefstroom Campus of the North-West University

Supervisor: Prof WC Venter

November 2014

Declaration

I, Gerhardus Jacobus Christaan Venter, hereby declare that the dissertation entitled “Development of an online reputation monitor” is my own original work and has not already been submitted to any other university or institution for examination.

_____

G.J.C. Venter

Acknowledgements

I would like to thank:

- My father and study-leader Prof W.C. Venter for his support, assistance and guidance throughout this research.
- My mother Dr. A. Venter for proofreading some of my articles, supporting me throughout this research and bringing me a cup of coffee whenever I needed it most.
- Ornico House Ltd for their financial support as well as providing technical assistance.

My thanks to anyone else that contributed to the project that I have not mentioned above.

All glory unto Him.

Abstract

The opinion of customers about companies are very important as this can influence a company's profit. Companies often get customer feedback via surveys or other official methods in order to improve their services. However, some customers feel threatened when their opinions are publicly asked and thus prefer to voice their opinion on the internet where they take comfort in anonymity. This form of customer feedback is difficult to monitor as the information can be found anywhere on the internet and new information is generated at an astonishing rate.

Currently there are companies such as Brandseye and Brand.Com that provide online reputation management services. These services have various shortcomings such as cost and is incapable of accessing historical data. Companies are also not allowed to purchase these software and can only use the software on a subscription basis.

The design proposed in this document will be able to scan any number of user defined websites and save all the information found on the websites in a series of index files, which can be queried for occurrences of user defined keywords at any time. Additionally, the software will also be able to scan Twitter and Facebook for any number of user defined keywords and save any occurrences of the keywords to a database. After scanning the internet, the results will be passed through a similarity filter, which will filter out insignificant results as well as any duplicates that might be present. Once passed through the filter the remaining results will be analysed by a sentiment analysis tool which will determine whether the sentence in which the keyword occurs is positive or negative. The analysed results will determine the overall reputation of the keyword that was used.

The proposed design has several advantages over current systems:

- By using the modular design several tasks can execute at the same time without influencing each other. For example; information can be extracted from the internet while existing results are being analysed.
- By providing the keywords and websites that the system will use the user will have full control over the online reputation management process.
- By saving all the information contained in a website the user will be able to take historical information into account to determine how the keywords reputation changes over time. Saving the information will also allow the user to search for any keyword without rescanning the internet.

The proposed system was tested and successfully used to determine the online reputation of many user defined keywords.

Disstertation keywords: Online Reputation Monitor, Web crawler, Facebook, Twitter, dtSearch, Sentiment Analysis.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API: Application programming interface

ORM: Online reputation management

SDK: Software development kit

Chapter 1
Introduction

This chapter will serve as an introduction for this dissertation. The chapter will start by providing a quick scenario that will demonstrate how consumer opinions influence a company followed by a general overview regarding the need and use of ORM system and the problems this research will aim to address. To finish the chapter the overall objectives of this research will be stated followed by a quick outline of the methodology this research will follow and a general outline of the following chapters.

1.1. Scenario

The Xbox One, the third generation of home entertainment and video game console, was unveiled by Microsoft™ on 21 May 2013. While technologically superior to its previous generations, the system caused controversy amongst critics and consumers due to strict digital rights management policies, such as requiring the user to connect the console to the internet every 24 hours and blocking the use of pre-owned games.

Due to these restrictions, the perception of the Xbox One by the online community was largely negative. Many unhappy customers used the internet to voice their concerns on blogs and social networking sites such as Facebook and Twitter with many of them planning to purchase one of the Xbox One's competitors instead. Microsoft has listened to the feedback and changed many of the Xbox One's policies since its original announcement but a lot of consumers still have a negative perception surrounding the console which influences the console's sales up to the present day.

1.2. The problem

Word-of-mouth communication is considered to be a valuable marketing resource and is often underestimated. This includes all forms of information exchange among customers regarding characteristics, usage and experiences with particular products, brands or companies [1]. According to Reichheld [2], the tremendous cost of marketing and other promotions make it hard for a company to grow profitable. Reichheld believes that the only path to a profitable growth rate lies in the company's ability to get loyal customers to become the company's marketing department by sharing positive information or experiences involving the company. His research showed that there is a strong correlation between a company's growth rate and the number of customers who are likely to recommend using the company's services to somebody else.

Most companies know this and employ techniques such as focus groups and surveys to generate various statistics, as detailed in [3]. However, these methods are not always effective; consumers often feel under pressure when their opinions are publicly asked and therefore adjust their answers to avoid any potential confrontation. Instead consumers often voice their opinions on the internet by making use of blogs and/or social networking sites where they take comfort in anonymity. Most companies are aware of this but are unable to generate statistics from these sources as they are too numerous and new information is generated too rapidly. Therefore companies require computerized techniques that will allow them to monitor their online reputation.

1.3. Project objectives

Determining online reputation is not a new field and has been done for years by organizations such as BrandsEye [4] and Brand.Com [5]. However, the services these companies offer have several limitations: in order to make use of the ORM services a company has to pay an ORM company a monthly free which often costs thousands of rands. At time of writing the "small" package at BrandsEye costs $500 per month (R5 500 at an exchange rate of R11 = $1) and the "medium" package $800 per month (R8 800 at an exchange rate of R11 = $1) [6]. In order to lessen this cost many companies would rather opt to purchase ORM software to perform the reputation monitoring themselves, but none of the ORM companies have such an option.

Another problem with existing ORM systems is many of them do not scan all available information sources, such as web pages and social networking posts, or scan information sources that are not relevant to a specific brand, for example looking for "CNA" on websites that only contain articles about fast-food restaurants. This would either cause a significant portion of customer opinions to be missed or the collection of too many results which would need to be filtered out.

Lastly, existing ORM services cannot access historical data as information for a brand is only collected from the present day onwards. If the user wants to add a new brand they would have to wait while the ORM system starts scanning for the new brand before a general overview can be calculated. This also eliminates any quick-search functionality which would be a nice feature.

Many of these limitations are implemented in order to make the ORM system as user friendly as possible while still providing the client with the necessary service. However, this makes it very hard for the user or company to customize the ORM service to their specific needs, which may lead to subpar results. As such, the goal of this research is to develop an ORM system that will present the user with as much control as possible while solving the problems mentioned above.

To be successful, the ORM system must be capable of:

- scanning a list of user-specified web pages at a sufficient rate to ensure that all the results are kept up to date;
- scanning social networking sites such as Twitter and Facebook for public opinions;
- storing all the gathered information in a database or other storage system to allow the ORM system access to historical data;

- analysing the gathered information for user-specified keywords in real time and report information regarding any keyword occurrences such as the location and sentiment of the mention,
- display the analysed mention to any interested party.

To achieve these goals the implementation of the new ORM system will primarily make use of existing components - there is no use in redesigning the wheel. Available components will be evaluated by measuring their performance according to criteria that are considered important for this research. After the evaluation the components that best match the criteria will be selected.

1.4. Research methodology

In order to complete this project the following methodology will be used

1.4.1. Study existing ORM software

Existing ORM systems will be studied to obtain information about the services these systems provide, how users interact with these systems and the components that make up an ORM system. The ORM system that will be investigated are:

- BransEye.
- Brand.Com

1.4.2. Study existing tools

The components used in these ORM systems will be studied in more detail. These components include:

- Web crawler
- Social network scanner
- String similarity filter
- Sentiment analysis tool

1.4.3. Design the ORM system

A new system will be designed once enough information regarding the internal operation of an ORM system as well as the components that make up an ORM system has been retrieved. The design will prioritize customizability that will allow the user to control as much of the processes as possible

1.4.4. Experiments

The new ORM system will be tested to determine the effectiveness and accuracy of the system. These tests include:

- using different components and implementations;
- altering the number of active web crawling instances;
- using different computer hardware and internet connections.

The results from the experiments will be used to optimize the software as well as determine the strengths and limitations of the system.

1.4.5. Conclusion and recommendations

Conclusions on the effectiveness of the system will be drawn after all tests were executed and future research areas that could improve the system will be highlighted.

1.5. Dissertation Outline

1. **Introduction**. This is the current chapter which gives some information regarding the factors that inspired the research, the problem which the research will aim to solve and the methodology that will be used to solve the problem.
2. **Background and literature study**. In this chapter the use of ORM systems will be clarified and existing ORM systems will be investigated. The components that make up an ORM system,

which include web and social network crawlers, similarity filters and sentiment analysis tools will also be discussed.

3. **Concept Design**. The new ORM system will be conceptualized and all available components will be investigated and compared whereby the best component will be identified and selected for the new ORM system. After all the components have been selected the concept for the new ORM system will be revised whereby any changes that are required by the components will be implemented into the design.
4. **Implementation**. The selected components will be implemented and optimized. Any other significant features of the software such as the use of multithreading and text filters will be discussed as well.
5. **Tests and results**. All tests that were carried out on the system as well as their respective results and their influence on the software will be discussed.
6. **Conclusion and recommendations**. In the last chapter a brief overview of the research will be given along with the objectives that were achieved and a summary of the results. Any potential future work and future research ideas will also be discussed for anyone that wishes to continue with the project.
7. **Appendixes**: Appendixes are additional chapters that will give extra information on specific concepts that are either mentioned or used during this dissertation.

Chapter 2

Background and Literature Study

This chapter will provide the reader with detailed information regarding the importance of customer opinions on company sales and why companies require ORM services. Next the components that make up an ORM system will be further discussed by first providing in-depth information about each component followed by its method of operation and interaction with other ORM components. Lastly existing ORM systems will be evaluated whereby the advantages and disadvantages of several existing ORM solutions will be listed and discussed.

2.1. The need for ORM

The goal of any business venture is to produce a profit. The most common way this can be achieved is either by selling goods or delivering a service. For many businesses a profit is so important that profit margins are the starting point for any budget planning. However, guessing the number of sales is one of the most difficult tasks any business has to make as it is difficult to predict or estimate the number of potential customers with reasonable accuracy. Therefore businesses try to influence the general public into becoming potential customers by means of advertising [7].

Advertising is a mass communication tool that communicates the same message to each person in public. According to Ayanwale [7] advertising is used to establish a basic awareness of a product or service in the potential customer by providing selected information about the product or service. Advertising can be used to great affect: Ayanwale's research [7] shows that advertising has a major influence on customer preferences and a separate study by Stephen Hoch [8] claims that while consumers state they do not believe everything advertisements claim it does help them to make decisions. Ayanwale also noted that brand preferences do exist and that customers will stay with a tried-and-tested brand even if there are better alternatives on the market.

From the research done by Ayanwale [7] and Hoch [8] it can be seen that brand preferences influences a company's sales. Therefore companies must investigate how the public perceives its brand. In order to do this a company must first determine the public's opinion regarding the company itself and its associated brands. Retrieving this information is not quite as simple as customers are scared of any potential backlash by providing their opinions publicly and will therefore use the internet to express their opinions where they take comfort in anonymity. This presents a problem as references regarding the company or brand can potentially be found anywhere on the internet, but scanning the entire internet is impossible as it is simply too large and grows too fast. However, much of the information on the internet is repeated as different websites write articles or posts about the same information. Therefore relevant information can be retrieved by scanning a specific number of websites relevant to the information that is required.

Scanning web pages manually is a very exhausting task and a number of factors such as wrong interpretation of opinions, typing errors, sickness and fatigue can impact performance. Therefore the ideal solution would be to develop a software application in order to lessen the amount of human interaction. Such software is called Online Reputation Monitoring (ORM) software.

2.2. How does ORM work?

Online reputation management (ORM) consists of monitoring various media such as web and social networking sites, detecting relevant content and analysing what people say about an entity [9]. In order to accomplish this an ORM service must scan the internet for specific keywords, download and analyse the results before displaying them. This is shown in *Figure 1*.

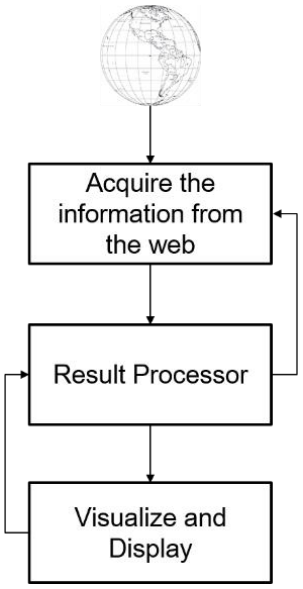

Figure 1: Overall process of an ORM system

The three processes demonstrated in *Figure 1* should be capable of operating independently from each other; a user must be able to process any web crawler results even if the web crawler is currently busy acquiring new information from the internet. Likewise a user should be able to view results for a specific day even if another user is busy processing results for a different day.

While *Figure 1* can give the reader a general overview of the functionality of an ORM system, each component can be further expanded upon, as shown in *Figure 2.*

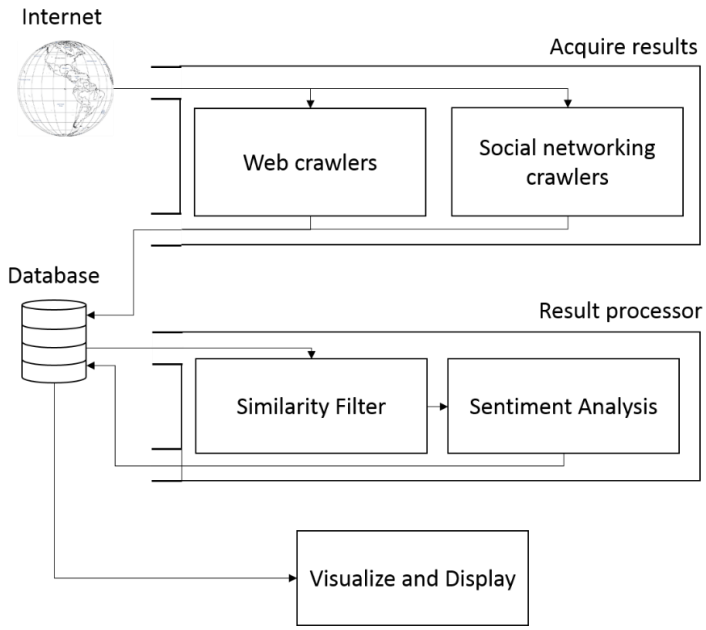

Figure 2: Detailed process of an ORM system

In order to acquire information from the internet the user must make use of a *web crawler*. Web crawlers are programs that explore the World Wide Web, retrieve information according to specific criteria and store any results in a database or some other storage for future use [10]. Unfortunately, for reasons that will be discussed in Section 2.3.2, web crawlers cannot scan social networking sites and therefore a tool that can extract information from such sites will also be needed.

Once the information from the web crawlers have been retrieved the results must be processed. Not all results will contain significant meaning, such as tags within a web page that will only highlight important aspects of that page. Therefore a similarity filter must be included to filter out results that are deemed too similar to the keyword that was used to detect the results. If a result passes through the similarity filter it must be passed to the final part of the result processor which will calculate the result's sentiment. Once the processing is complete the result must be saved back to the database where it can be retrieved and shown at will.

2.3. ORM components

As can be seen from *Figure 2* an ORM system consists of several processes that work together to produce a result. Therefore, in order to design an ORM system the components that make up such a system must be investigated. As shown in *Figure 2* the components are:

- a web crawler;
- a social network scanner;
- a method of analysing the results from the web and social network crawler;
- sentiment analysis tools.

The way web crawler results are generated depends on the web crawler that will be used for this ORM system. This will be discussed in the web crawler component section (Section 3.2.1). The rest of the components will be investigated in the following sections.

2.3.1. Web crawlers

Web crawlers are programs that explore the World Wide Web. A key motivation for designing web crawlers are to retrieve web pages and store them or any relevant data for future use [10]. This process is known as *web crawling* and is most notably employed by search engines to locate new resources on the web.

The type of data that can be extracted from web pages depends on the implementation of the web crawler. Some web crawlers are configured to extract only specified phrases [11] while others extract and index each word in a web page for future use [12].

Figure 3 shows the architecture of a basic web crawler [10]. Before a web crawler is started a user must specify a series of seed URLs which is stored in the *frontier*, a list of URLs that must be investigated. When the web crawler starts it will load the first URL in the frontier, download the associated web page, scan the page and store any relevant information in a database or local storage. Once the crawler has finished scanning the page it will load the next URL from the frontier and repeat the process until all the web pages in the frontier have been scanned. This is known as the *crawling loop.*

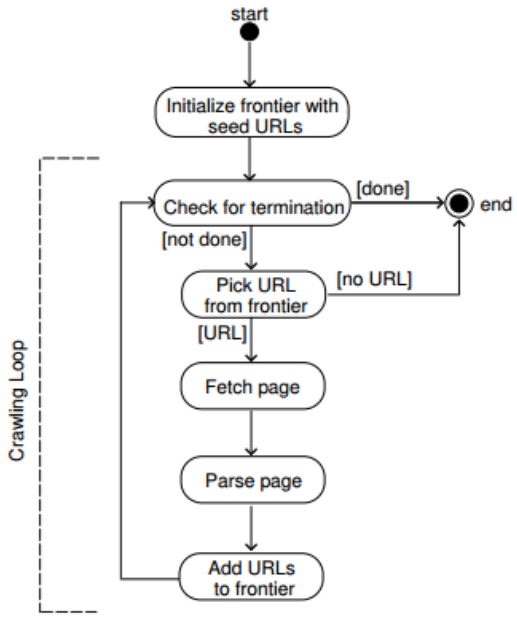

Figure 3: Basic Crawler Architecture

Crawlers are often configured to scan a website up to a certain depth. This is the extent to which a web crawler will scan pages within a website. Many web sites contain multiple pages, which in turn contains additional subpages. This is illustrated in *Figure 4*.

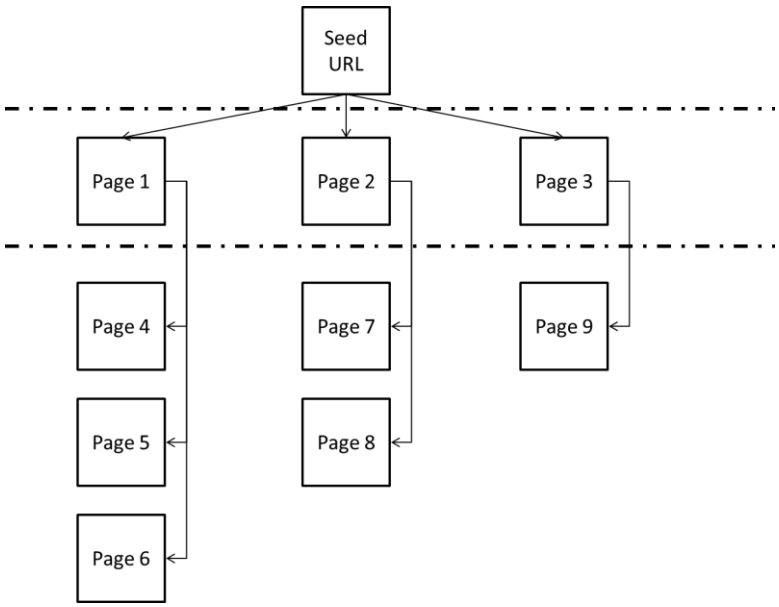

Figure 4: Crawl Depth Illustration

The crawl depth for each web site is specified when the user adds a seed URL to the frontier. IF the user wish to scan only the original page the crawl depth must be set to 0. If the user sets the crawl depth to 1 the web crawler will scan the seed URL and add the URLs for *Page 1, Page 2* and *Page 3* to the top of the frontier. Once the web crawler has finished scanning the seed URL it will proceed to download and scan *Page 1*, followed by *Page 2* and finally *Page 3*. If the user sets the crawl depth to 2, the crawler will add the URLs for *Page 4*, *Page 5* and *Page 6* to the top of the frontier when scanning *Page 1*. Once *Page 1* has been scanned, the web crawler will scan *Page 4, Page 5* and *Page 6* before proceeding to *Page 2*. This process will be repeated for the links within *Page 2* and *Page 3*. It should be noted that as the crawl depth increased the number of web pages in the frontier will increase as well. Therefore the time it will take for the web crawler to scan all the pages in the frontier increases dramatically with the crawl depth.

Web crawler execution ends when all the links in the frontier have been scanned. At this stage a timer is often initialized which will redeploy the crawler at a specific time or after a specified interval.

When not optimized, web crawlers are designed to extract information from websites as fast as possible. While this would be perfect for the applications that use the results from the web crawlers this will add a massive load on the web servers that house the websites that are being scanned. For a single web crawler this would not be a problem but when multiple web crawlers are active simultaneously the web server can run out of resources available memory and processing power. This will lead to websites becoming unresponsive or stop working altogether, even if they are being scanned or not.

Therefore web servers need to manage their resources. Preventing the website from being scanned by a web crawler will be counterproductive as the information contained by the website will not be scanned by search engines, which will cause the website to become obscure to the online community, but allowing web crawlers to scan the web site at full capacity will impact stability. To solve this problem, websites usually implement a "polite policy": a file named "robots.txt" is added to the website which aids web crawlers by providing specific information such as:

- a list of web files that are archived or contain insufficient information;
- a list of web directories that are not to be scanned;
- different kinds of permissions to specific web crawlers;
- a crawl delay between web pages.

This will ensure the web crawler will only scan the correct information such as the website's articles and content while excluding files such as style sheets which are only needed to maintain the visuals of the webpage. This "polite policy" will also speed up web crawler execution; by reading the "robots.txt" file web crawlers will scan only the files that contain the information while skipping unnecessary ones.

This will lessen the time web crawlers spend analysing a website allowing web crawlers to move onto the next website in its frontier at a faster rate.

2.3.2. Social network crawler

In order to provide a more accurate analysis of a company or brand social networking sites must be scanned as well. Unlike regular web sites web crawlers cannot scan social networking sites. Social networking sites only present data once the user has registered and even if logged in the site will only show data regarding the people the current user is connected to. Therefore the problem is two-fold: normal web crawlers do not possess the necessary authentication capabilities to access the content of social networking sites and those that do will only have access to limited information.

To allow applications access to some of their data various social networking sites have designed an interface that allows registered applications to access the site's public data. This interface is known as an Application Programming Interface (API) and is often the only way of interfacing with a social network's database systems.

All social networking sites have different API's and therefore different methods of extracting information. Due to time constraints it is not be possible to design a universal method of interfacing with all social networking sites. The research in this thesis will focus on interfacing with the two most popular social networking sites, namely Facebook and Twitter.

2.3.2.1. Twitter

Twitter is an online social networking service with microblogging capabilities, created by Jack Dorsey in March 2006. Since then the site has grown rapidly and had 500 million registered users in 2012 who posted 340 million messages per day [13].

Twitter allows people to send *tweets*, which is a short message with a maximum length of 140 characters by using a computer or almost any mobile device. All users can access the site and read tweets but only registered users can create and send new tweets. Additionally registered users can opt to "follow" other registered users which will automatically provide the user with any messages the users they are following make. All tweets are public by default but a sender may choose to send tweets that are only visible by his or her followers. Though a tweet may be about any topic, a study in August 2009 by Pear Analytics has shown that more than 70% of tweets are either considered as "pointless nonsense" or conversational, which would be of use for an ORM service [14].

Twitter has created the Twitter Application Programming Interface (API) which allows external applications to access its databases [15]. The Twitter API allows developers to access a certain number of tweets for various purposes, including but not limited to statistic generation, targeted marketing and online reputation management such as this project.

Some of the information the API will provide include:

- tweet sender;
- tweet content;
- tweet date;
- unique tweet URL.
- tweet language.

In order to use the Twitter API, the user must first create a Twitter account and register an application. Once the application is registered the user will receive:

- A consumer key;
- A secret consumer key;
- An access token;
- A secret access token.

These four keys are used by Twitter to uniquely identify the application and determine which resources the application can access.

If the user wishes to request data from the Twitter API the user must provide the API with the four keys. Once the application has been authenticated the API will provide the user with an authentication token, which the user will use with any subsequent API calls. After verification the API can be used for a variety of tasks as detailed within the official Twitter documentation [16] (please note that this link uses a secure connection. In order to view the page and its content the user will have to register a Twitter account and enable application development).

2.3.2.2. Facebook

Facebook is an online social networking service that was founded on 4 Feb 2004 by Mark Zuckerberg at Harvard University. Since then the site has grown rapidly; at the end of January 2014 1.23 billion users were active on the website every month while more than 945 million users were connecting via mobile devices. In 2013 Facebook's revenue was $7.85bn, a 55% increase for the previous year [17].

When first registering the user is asked to create a personal profile, where he or she can specify information such as name, surname, residential address and current interests. Once the profile has been created the user can connect to other people which the site refers to as *friends* who may view each other's profiles and give personal comments.

Like Twitter, Facebook gives the user the option to post messages which may be public or private depending on the user's preference. Messages usually include text, images and occasionally hyperlinks that redirects to other websites. Unlike Twitter, Facebook messages are private by default.

In order to allow developers to access Facebook data with their applications, Facebook has developed the *Facebook Software Development Kit (SDK)*. While the SDK supports several features such as profile management, it will also allow applications to access several public posts which will be of use for ORM systems.

To use the Facebook API the user must first create a Facebook account by registering on the website. Once registered, the user must enable the developer settings on the account and register a new application in order to receive 2 Facebook keys:

- Application ID;
- Secret application ID.

These 2 keys are used by Facebook to uniquely identify the user's application and determine whether the user has access to the application. If the user wishes to request data from the Facebook API, the user must first authenticate the Facebook application by using the 2 Facebook keys. Once authenticated the user can use the API to perform a large variety of tasks as detailed within the official Facebook Developer documentation.

2.3.3. String similarity algorithm

Once the web or social networking crawlers have finished their crawling processes, all the results for a specific keyword will be returned. However, a certain portion of the web crawler results will have insignificant meaning, such as tags within web pages. Below is an example of such a message:

Keyword: MTN **Retrieved result: MTN Business**

The sentence above only refers to MTN as a business and has no semantic meaning, making it useless for an ORM system. To filter out these results an algorithm must be developed that will compare the result to the keyword that was used to detect the results and determine whether the result and the keyword differ enough. This will be done by using a mathematical comparison which will determine the number of operations that are needed to transform one string into another.

2.3.4. Sentiment analysis

If the result passes through the similarity filter it must be presented to the user. However, the user often wishes to know whether the content of the message is positive or negative, otherwise known as the *sentiment*.

Sentiment analysis on a computer is not an easy task as computers can't guess the meaning of a word by using its context as humans can. In fact the main problem with sentiment analysis is to determine how the sentiments are expressed within the sentences. According to Nasukawa and Yi [18], sentiment analysis involves identification of:

- Sentiment expressions.
- Polarity and the strength of the expressions.
- The relationship of the expressions to the subject.

Though there are several methods to calculate the sentiment of some text, the two main methods either use a lexical or machine learning approach [19].

2.3.4.1. Lexical approach

A system that is based on the lexical approach uses a dictionary or lexicon of pre-tagged words. Each word that is present in the text is compared against the dictionary to find its polarity. This indicates whether the word has a positive or negative meaning as well as the strength of the sentiment. After the sentiment of each word has been determined, the sentiment of the given text is calculated by summing the sentiment of each word. If the total sentiment is positive the text has a positive meaning, whereas if the total sentiment is negative the text will have a negative meaning.

According to Annett [19], the accuracy of such a system varies between 64% and 82%, depending on orientation of statistic metrics and the dictionary that was used.

2.3.4.2. Machine learning approach

A system that uses the machine learning approach uses two components; a series of feature vectors and a collection of tagged corpora. A tagged corpus is a collection of documents that the system uses to initially train itself [20].

Feature vectors are usually a variety of *uni-grams*, single words from within a document, or *n-grams*, two or more words from a document that are in sequential order. Other features that are often proposed include the number of positive words, number of negative words, and the length of a document.

Both the feature vectors and collection of tagged corpora are used to train a classifier, which can be applied to an untagged document to determine its sentiment.

According to Annett [19], the accuracy of such a system varies between 63% and 82%, but the results are dependant of the features that were selected.

2.3.4.3. Optimal approach

Both the lexical and machine learning approaches have several advantages and disadvantages.

Where the lexical system is significantly easier to develop and does not have to be trained a powerful linguistics resource from which decisions can be made is required and such a resource can be hard to find. It is also difficult to take sentence context into account when making decisions, for example sarcasm [21]. A lexical system might categorize a sentence such as "The prices at MTN are ridiculously low" to be negative due to the negative modifier "ridiculously" which would be wrong as the sentiment actually has a positive sentiment.

With the machine learning approach the dictionary is not required. Current systems that use this technique display a high level of accuracy, but high accuracies can only be achieved by using a representative collection of labelled training texts and a careful selection of features. In addition, classifiers that work in one domain, such as blogs, might give sub-par results in other domains [21].

Research done by Blinov [21] indicates that though machine learning techniques are capable of providing results with a high accuracy, systems based on the lexical approach can achieve matching accuracies using only small dictionaries. He concluded that though the advantages of the machine

learning approach are numerous, the lexical approach should not be discarded and instead be used in conjunction with the machine learning approach.

2.4. Existing solutions

Online reputation management systems are nothing new and have been in use for several years. In addition to doing online reputation calculations, ORM systems often contain additional functionality which are used to offer extra services to attract potential customers. In order to determine what kind of additional functionality are included the following existing ORM services were studied:

- Brandseye
- Brand.Com

Other ORM systems such as Radian 6, SaidWot and Alterian SM2 were initially included in the investigation but due to a lack of feedback they could not be investigated.

2.4.1. Brandseye

Brandseye originated in 2004 when consumers wanted to know what people were saying about their brands. At first the Brandseye team manually searched the internet for relevant information using search engines but this process was too cumbersome and using various free services did not yield any better results. As such the team began developing their own system capable of monitoring brands online and received their first paying client in 2006 [4].

Brandseye features a web based application with support for the latest versions of Mozilla Firefox and Google Chrome. Microsoft Internet Explorer is not supported, and produced faulty results during testing.

In order to use the system the user has to enter several ‘brands’ and ‘phrases’. Brands are pre-determined categories under which phrases are stored whereas phrases are the keywords the system will search for on the web.

The system supports various types of searching such direct searches, inclusive searches and exclusive searches. With a direct search the system searches for a series of keywords and will present only results where all the keywords feature. With an inclusive search two keywords are specified and the system will only present results where both of the keywords feature though they do not have to follow each

other as in the search string. Exclusive searches are the opposite of an inclusive search. The system will present only results where a primary keyword features and the secondary keywords do not.

Once a phrase has been detected by the system the number of occurrences in both the phrase and the brand is updated, with the brand showing the total number of hits all the phrases assigned to it has gotten. The number of hits each phrase received can also be individually shown by the system.

Once the system has received any number of results various reports such as 'amount of posts linked to Twitter and Facebook', 'country of author origin', 'post language' can be generated. The system also allows for custom reports to be generated by providing the user a wizard where they can specify the report criteria.

The advantages of Brandseye include rapid website scanning and a user friendly interface that is very easy to use. In addition the web site presents the user with as much control as possible without exposing too much technical functionality.

However, the system does not make use of historical data and will only report results for a keyword from the date it was added. Additionally the system does not automatically calculate the sentiment which must be manually added by the user and is quite expensive to use, costing a monthly fee of $500 per month for the small package and (R 5 500 at an exchange rate of R11 = $1) $ 800 per month (R 8 800 at an exchange rate of R11 = $1) for the medium package [6].

2.4.2. Brand.Com

Like Brandseye, Brand.Com features a web based application with support for the latest versions of Mozilla Firefox, Google Chrome and Microsoft Internet Explorer.

After the user has been successfully registered and entered some keywords for the system to detect, the user is taken to his Dashboard, from where he or she can control the system. The Dashboard provides the user with a general overview of his account and the keywords the system must detect. On the Dashboard the user can see the number of positive and negative results and the number of times the keyword was searched per month. Selecting either of these options will generate a graph showing the user the percentage of results that was generated per month for the last 12 months. These results are generated by using a search engine. At the time of investigation Brand.Com has been successfully incorporated with Google, Yahoo and Bing. Brand.Com also offers a live feed, where information from Twitter, Facebook and additional websites that contain the user-provided keywords are automatically listed.

An interesting feature is the ability to add system-provided keywords that are related to the user-provided ones. However, this functionality could not be explored further due to the limitations of the free account.

Based on the abilities of the free account, Brand.Com has some obvious advantages. Brand.Com scans the internet at a significant pace, as new results are usually added at a rate of 1 per second. The related keywords feature is also a very helpful tool as it will help users to refine their search results.

However, during the time of testing no results were marked as positive or negative though is not clear whether the user has to mark results him or herself or whether this functionality is disabled due to the free account. In addition the user has to select the data server location and only a few options are provided, all of them limited to the USA which will result in the user primarily using US websites in order to detect keyword mentions. It is not clear how this influences results. At the time of writing using Brand.Com costs upwards of $3 500 (R38 500 at an exchange rate of R11 = $1).

Chapter 3
Design

This chapter will detail the methodology that was followed to design the new ORM system. The chapter will start by providing the user the goals that the ORM system must accomplish and any restrictions that must be taken into account. Next a new ORM architecture will be proposed along with motivation on why certain design choices were made. For each section of the new ORM architecture various existing tools will be researched and evaluated in order to determine whether any pre-programmed tools can be incorporated into the design of the ORM system or whether a new tool must be designed. The chapter will end with a re-evaluation of the proposed ORM architecture to determine whether any of the existing tools will require a change in the architecture.

3.1. The Design

As stated in Section 1.3, the main goal of an ORM system is to:

- scan web pages and different social networking sites at a sufficient rate to ensure all results are kept up to date;
- analyse the results to give the user information such as the location, page, paragraph and sentiment of the results;
- use the analysed results to generate reports.

Before a concept design can be done, the requirements and boundaries of the research must be established. At the start of the project the following requirements were established:

- The ORM software must be compatible with Microsoft Windows platforms with Windows 7 as the oldest acceptable operating system. Other operating systems are optional.
- The software must be written within the .Net Framework with .Net Framework 3.5 as the lowest acceptable version.
- The system must focus on English results.

Due to these requirements it was decided to use Microsoft Visual Studio 2010 as development environment and Microsoft SQL Server as database management system.

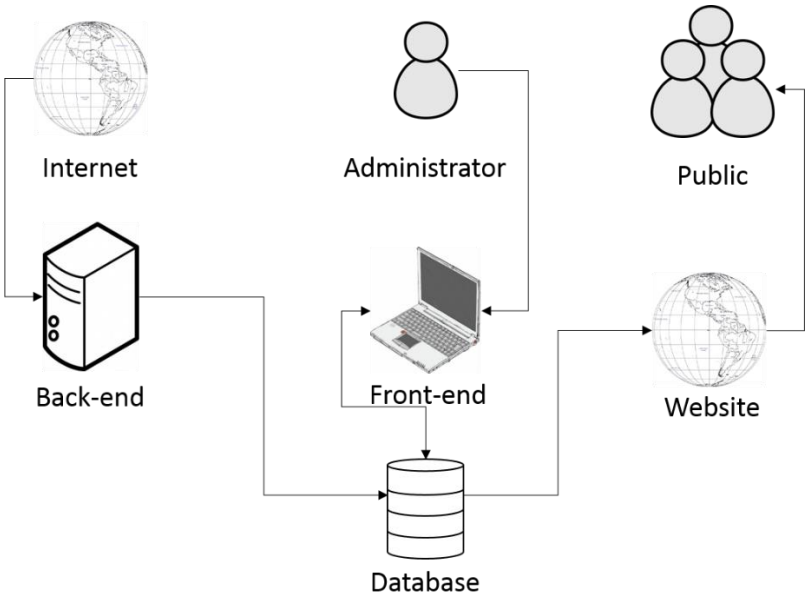

Figure 5: Proposes System Architecture

From a design viewpoint it will be a good idea to divide the ORM system into three sections, as shown in *Figure 5.* The sections will consist of:

- the Back-End;
- the Front-End;
- the Website.

3.1.1. The Back-End

To start the ORM process information must be retrieved via the internet. This will be done by web crawlers and social network scanners which will be maintained by the Back-End. The information will be passed through several filters which will remove any duplicate and non-English results. The results that pass through the filter will be saved to a database or any designated storage device. Once the Back-End has finished crawling the web and social networking sites it will proceed to wait for a predetermined amount of time before restarting the process.

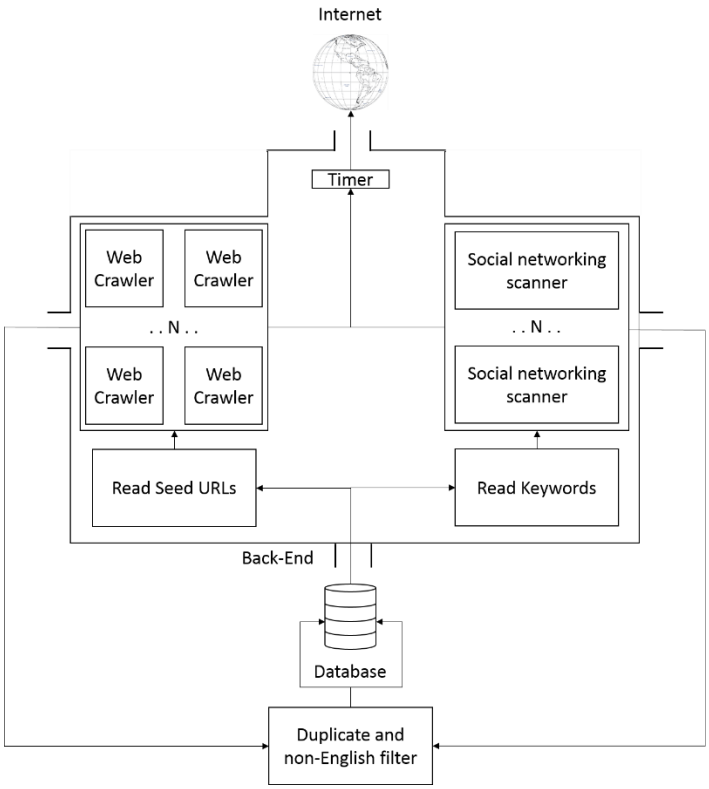

Figure 6: Proposed Back-End Architecture

Figure 6 shows the proposed Back-End architecture. When the crawling process starts the Back-End will read the list of websites and keywords from the database and divide the information among the web and social network crawlers. The websites and keywords must be inserted via the Front-End before the web crawling processes starts. After dividing the information the web crawlers will proceed to scan the web sites while the social network crawlers will scan the Twitter and Facebook public streams for any mentions of the keywords. Once the web crawlers have finished scanning the sites in their frontiers they will save the information to the database and enter a waiting period, repeating the web crawl after a pre-determined amount of time has passed. The social network scanners will not shut down and will continue scanning until manually stopped.

It can be seen that the Back-End does no calculations other than those that are necessary to maintain the web and social network crawlers as well as their filters. A primary characteristic of the Back-End will be its ability to function with as little human interaction as possible. The only interaction that is required will be to monitor that the Back-End has not crashed or to turn off the system in times of maintenance.

The number of active web crawlers in the Back-End can be changed by a system administrator. This will allow the system administrator to optimally allocate the available internet bandwidth across all the applications that require an internet connection while simultaneously reducing the time it will take the Back-End to scan all the allocated web sites. This will also prevent the internet connection from being overloaded; if the system administrator activates too many web crawlers some of the web crawlers may not receive enough information to continue operating which will result in a time out. This will be further discussed in Section 4.1.1.

3.1.2. The Front-End

When the Back-End has finished acquiring data from the internet the system administrator can select a date range for results to be generated. The stored records for the specified date range will be loaded, processed by passing it through a similarity filter and finally evaluated by a sentiment analysis tool.

The similarity filter will remove results that are too close to the keywords, such as tags within a web page. This will lessen the number of results that will have to be processed in the following steps. The sentiment analysis tool will use the results that have passed through the similarity filter and determine whether the results are positive or negative. The sentiments will be used to determine the overall opinion of the company or brand that is being investigated. Once the sentiment of the results have been calculated it will be saved back in the database.

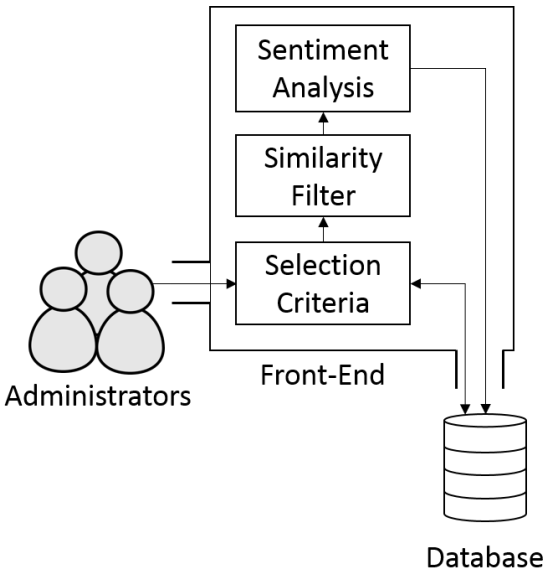

Figure 7: Proposed Front-End Architecture

Figure 7 shows the proposed architecture of the Front-End system. Unlike the Back-End, the Front-End has no web crawling capabilities and will be used only for information extraction and refining. The Front-End will also be designed to allow multiple instances of the software to be run in parallel, which will allow multiple system administrators to access the system at the same which will increase the number of results that can be processed at a simultaneously.

Unlike the Back-End, it will not be possible to fully automate the result generating processes of the Front-End. Computers cannot (yet) detect the meaning of words based on context, which may lead to incorrect results. This is shown in the example below:

Keyword: Kalahari

Returned results:

- "I bought this lovely watch from Kalahari for only R299.99"
- "Boy, the Kalahari is hot this time of year."
- "Kalahari has a major special on games, you should go check it out."
- "Rain in the Kalahari, who would have known?"

While a human would be fully capable of determining which results refer to the online market place and the Kalahari Desert, a computer cannot do so. Instead the computer will use all the results that contain the keyword “Kalahari” to determine the online reputation for the keyword, regardless of their semantic meaning which would greatly influence the results, especially if the people are more favourable towards the online marketplace than the desert. As such a system administrator must filter through the results provided by the web crawler and must manually determine which results are correct. Once the system administrator has marked a result as relevant the system will calculate the result’s sentiment and category by using the sentiment analysis tool before adding the result to a list that contains all the results that the user has marked as relevant. Once the user has finished marking the relevant results he or she may proceed to review the results. Once processed, the results will be saved back to the database from where it will be used to generate various statistics.

3.1.3. The Website

The website will be the final part of the ORM system. A website will be developed that will use the analysed results stored to visualize and display the data to any interested party. This process is shown in *Figure 8.*

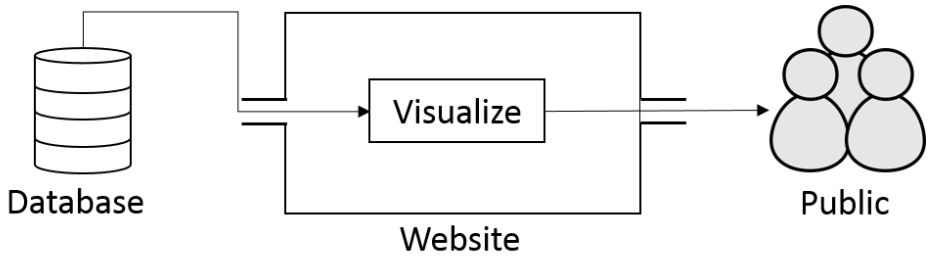

Figure 8: Proposed Website Architecture

As shown in *Figure 8* the website will not contain any processing functionality and will only be used to display the results on the web.

3.2. Component selection

As stated in Section 1.3 the ORM system will make use of existing components. Each of these components will be discussed in the following sections:

3.2.1. Web crawlers

Ideally the ORM system would make use of a powerful web crawler such as *GoogleBot* or *BingBot*, respectively developed by Google and Microsoft but unfortunately these crawlers are not available for public use. An investigation discovered many free and commercially available web crawlers available on the internet such as:

- A commercial web crawler from dtSearch named the dtSearch engine [12].
- A web crawler development kit named the HTML Agility Pack [22].
- Several open source web crawlers from GitHub that is written within the .Net Framework.

3.2.1.1. dtSearch Engine

Background and features

The dtSearch Engine is developed and maintained by the dtSearch Corporation. The company began developing text retrieval systems in 1988 and started marketing its software in Virginia in 1991. Since then the dtSearch engine has expanded from a desktop application to include web applications [23].

The dtSearch engine has numerous features that include:

- support for Microsoft Windows and Linux platforms;
- support for C++, java, and the .Net Framework;
- natural language searches which allows the users to search requests in ‘plain English’;
- support for entire phrases, Boolean operators, wildcard searches and words that are within proximity of each other;
- support for phonetic searches and words with variations on their end such as *applies* and *applied;*
- saving of web pages where results are located.

Method of operation

In order to use the dtSearch Engine within a programming environment the user must first specify which websites to crawl as well as their respective crawl depths. Additional information that the dtSearch Engine can use but are not compulsory include:

- file filters that allow the web crawler to automatically skip specific files,
- the maximum time to spend on a website and,
- the maximum file size to download.

When the web crawler is started it will acquire the first website in its frontier and proceed to scan it. The dtSearch Engine will not look for any user specified keywords on the web; instead it will proceed to generate a list containing every word on the page as well as the locations of the word and store the results in an *index file*, a collection of documents that contains every word the dtSearch Engine has detected as well as the positions where the words were found. This technique allows the dtSearch engine to search large volumes of text very quickly [24]. Unfortunately, these index files cannot be saved to the database and must be saved to a local storage drive.

Should the crawl depth of the web crawler be 0 the web crawler will proceed to scan the next page in the frontier. If not, the web crawler will extract all the hyperlinks in the current page and all them to the top of the frontier, repeating this process until it has reached its specified depth as shown in Section 2.3.1. This process is demonstrated in *Figure 9*.

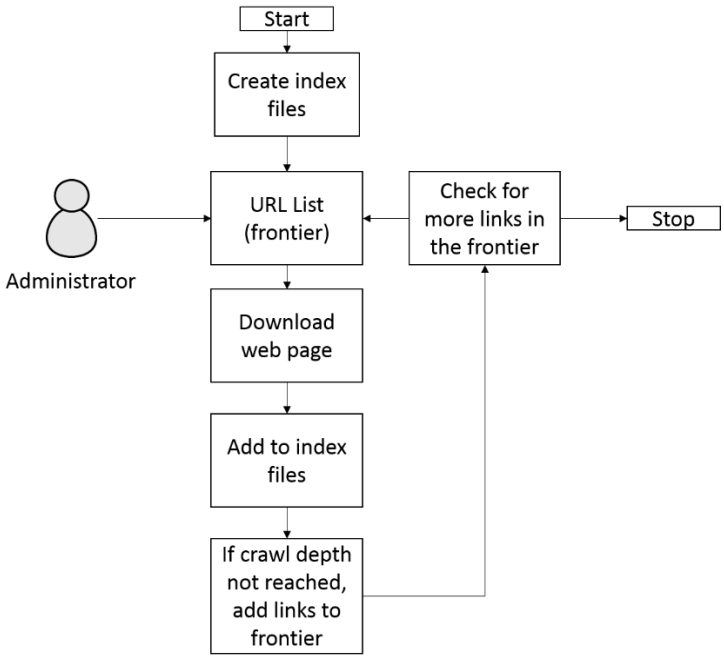

Figure 9: dtSearch Engine search method

The index files are created by an algorithm unique to the dtSearch Engine. As such only the dtSearch Engine is capable of reading data from the index files. In order to use the data the dtSearch Engine contains several result generating capabilities; once the user provides the system with a keyword the dtSearch Engine is capable of quickly scanning through all the index files and extract the paragraphs that contain the specified keyword. By using the index files the result generator is also capable of regenerating the web site on which the results were found.

For more detail about the index files, see Appendix A

Advantages and disadvantages

Using the dtSearch engine will provide the system with a web crawler that is actively being maintained and currently used worldwide by several companies including *ContractIQ* [25], *Densan Consoltants* [26] and *American Technology Services Inc* [27]. As such no custom web crawler functionality will have to be written which will save a lot of development time.

Unfortunately, the dtSearch Engine is commercially used and is therefore not free. A trial version can used to test its functionality but should the ORM software ever be used commercially the dtSearch Engine will have to be purchased, which would cost $2 500 (R 27 500 at an exchange rate of R11 = $1). The user will also not have any control over the internal functionality of the web crawler and will instead have to use the provided settings to customize the dtSearch Engine for the software application. Lastly, due to its use of index files the dtSearch Engine cannot save its results directly to a custom database.

3.2.1.2. HTML Agility Pack

Background and features

The HTML Agility Pack (HAP) is a library that allows the user to parse web pages that have not undergone any HTML restructuring, for example web pages that has missing sections or tags. Using this functionality the HAP allows the user to build document models that can be used to fix the HTML as well as provide an easy way to extract information from web pages, which may be used to build web crawlers. [22].

HAP features:

- support for the Microsoft Windows platforms;

- support for the .NET Framework;
- gives the user complete control over his or her web crawling application.

Method of operation

Like the dtSearch Engine, the user must specify a list of URLs the HAP must scan but instead of using index files the HAP makes use of document models, which is a model that displays document information in a tree-like system. An example of a document model is shown in *Figure 10.*

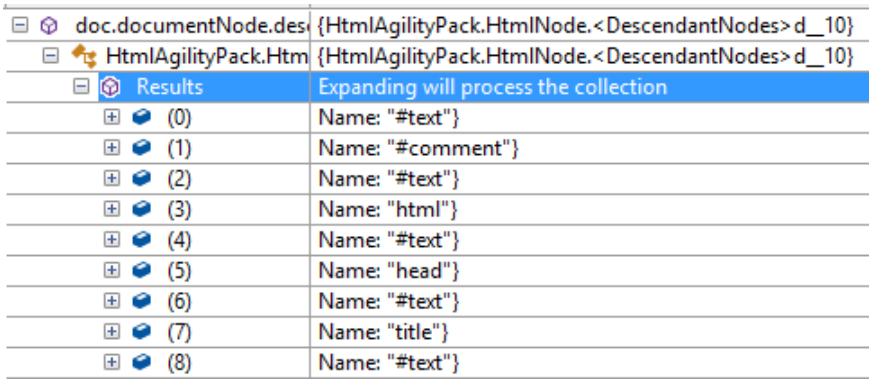

Figure 10: Document Model

By using the model the user can scan for every sentence in the web document and check whether a specific keyword is present and if so extract various information such as the redirected URL, the title of the page and any hyperlinks within the document. What happens from here depends on the user: the user can either proceed to store the text for later use or leave the web page to continue scanning for more results, it depends on what the function of the web crawler must be.

Like the dtSearch Engine, the HAP will continue adding links to the frontier until the crawl depth has been reached and stop the crawling process once all the links in the frontier have been scanned. This entire process is detailed in *Figure 11.*

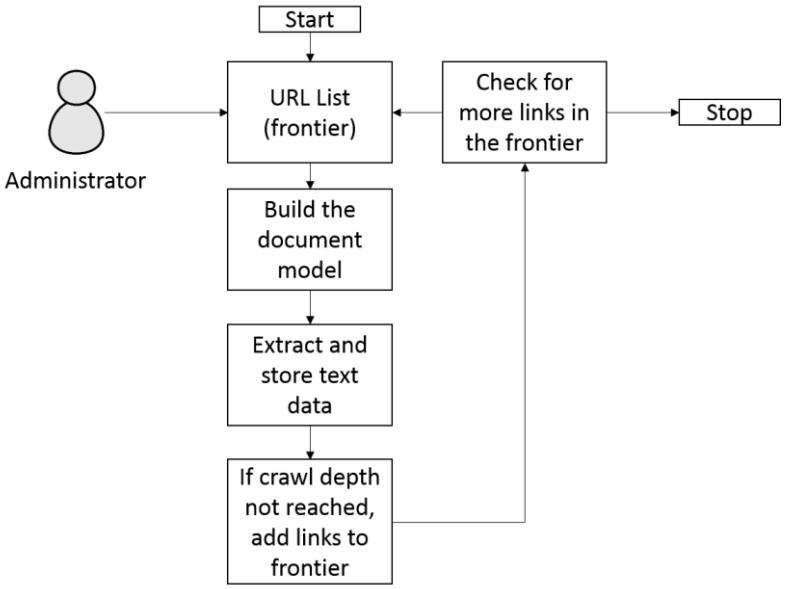

Figure 11: HTML Agility Pack search method

Advantages and disadvantages

The HAP has two main advantages: it's free and the user will have full control over the web crawling application. Unfortunately the HAP only contains functionality to support the *development* of web crawlers. The HAP does not contain a web crawler by itself and therefore the user would have to develop and test a web crawler from scratch, something which could take up a lot of time. Additionally, the HAP is open-source and developed by a community of developers. While this may lead to rapid development and addition of several new features it also means the community may stop supporting the HAP at any time which would increase development difficulties, especially if there are any errors within the HAP itself. As of writing the HAP was last updated on 10 Jul 2012, which suggests the community may have moved on to other projects.

3.2.1.3. Open-source web crawlers

Along with the dtSearch Engine and the HTML Agility Pack several open source web crawlers from an open source repository named GitHub were investigated. Though an initial search for "*web crawler"* yielded hundreds of matches the results were restricted to those that satisfied the project requirements, namely web crawlers that operate on Microsoft Windows platforms which use the .Net Framework. Filtering out the irrelevant results gave 44 results, of which the three with the best ratings were selected. At the time of investigation these results were:

- Sjdirect/abot (https://github.com/sjdirect/abot)
- Bolthar/tenteikura (https://github.com/bolthar/tenteikura)
- Dfdemar/Weaver (https://github.com/dfdemar/Weaver)

3.2.1.3.1. Abot web crawler:

The Abot system is an open source web crawler kit developed in C#, which has been designed with speed and flexibility in mind. It uses the HTML Agility Pack (Section 3.2.1.2) as its basis and improves on it by automating many of the HTML Agility Pack's processes such as building document models, multithreading, http requests, scheduling and link parsing. However, the user is still required to design his or her own events for actually processing the page data and generating results.

Method of operation

As the Abot system uses the HAP as its basis, interfacing with a web crawler developed by using the Abot system differs only marginally from a HAP application. Firstly the user has to specify the URLs in the frontier and configure the web crawler application. The web crawler will proceed to scan the first URL in the frontier and present the data to the user, which can be analysed and stored for later use before the system will move onto the next page. Like the dtSearch Engine and the HAP, the Abot system will continue adding links to the frontier until the crawl depth has been reached and stop the crawling process once all the links in the frontier have been scanned. This entire process is detailed in *Figure 12*.

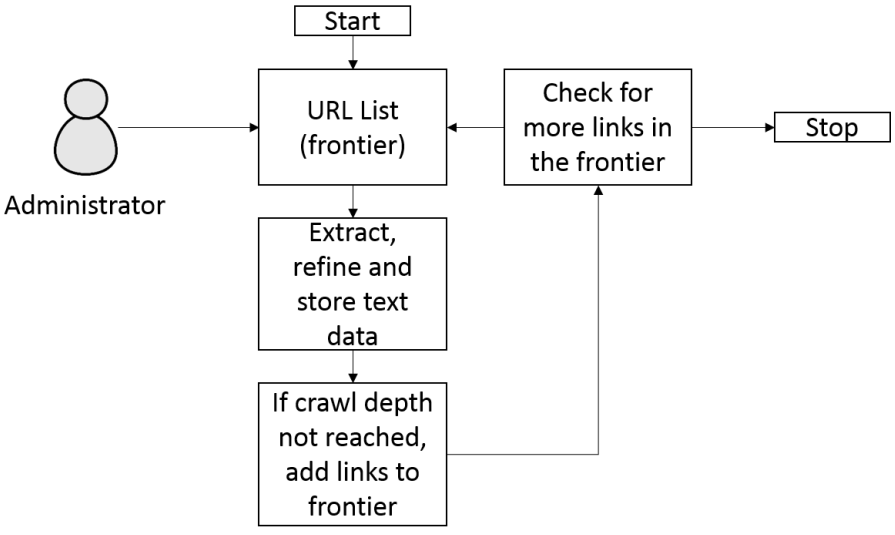

Figure 12: Abot search method

Advantages and disadvantages:

Many of the advantages and disadvantages from the HAP carry over to the Abot system. Like its basis the Abot system is free and the user will have full control over the web crawler application. The Abot system compares favourably to the HTML Agility Pack as the user will not have to develop the low-level processes such as multithreading and link parsing which will lower the development time required to create a full web crawler. The Abot system is also actively maintained by the community, with the latest update released on 1 Feb 2014.

However, like the HAP the Abot system only contains functionality to aid in the development of web crawlers. While many low-level processes have been pre-developed, many others like result generation and web page recalling must be developed manually. This will significantly increase development time.

3.2.1.3.2. Tenteikura

This web crawler is merely an implementation of the HAP. As the HAP and an implementation of the HAP have already been examined in Sections 3.2.1.2 and 3.2.1.3.1 the Tenteikura web crawler will be discarded from the investigation.

3.2.1.3.3. Weaver

As explained in Section 2.3.1, web crawlers implement a polite policy in order to prevent the web servers they are scanning from malfunctioning. This is where the Weaver web crawler differs from regular web crawlers; this crawler is designed to extract as much information from its target website by ignoring any polite policies.

Method of operation

The Weaver crawler is a custom web crawler implementation that does not rely on any external components. Once the user specified a list of seed URLs the web crawler will start downloading the first URL in the frontier before immediately creating a second instance that will run in parallel with the original one. This instance will download the next URL before repeating the processes for the following URLs. Once a thread has finished scanning a URL it will list all the URLs within the web page and

scan them by creating a thread for each URL, provided the crawl depth and thread limit have not been reached. The web crawler will stop once all threads have been scanned. This is shown in Figure 13.

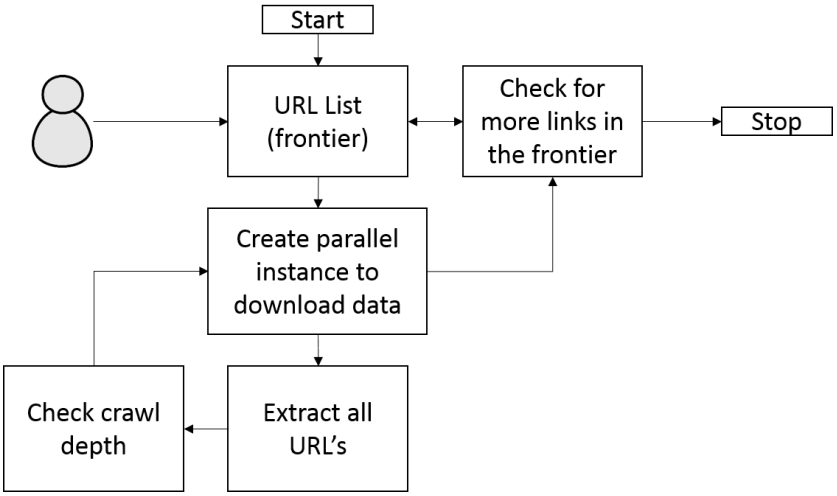

Figure 13: Weaver search method

Advantages and disadvantages

The lack of any external components makes the Weaver web crawler easier to deploy than the other web crawlers as no external libraries have to be maintained. Its focus on speed is also a major advantage.

The Weaver web crawler's largest disadvantage lies in its extensive use of parallel instances. More system resources are used as the number of instances increase which can lead to a system malfunction if too many instances are created. Another disadvantage is the absence of any result generating capabilities. As such all result generating capabilities would have to be developed which can take significant amounts of time.

3.2.1.4. Speed comparisons

In order to compare the speed of the web crawlers, each web crawler was timed to see how long it took the web crawler to scan through the same 100 websites with a crawl depth of 0. Each test was carried out 3 times in order to determine an average. Due to the different methods each web crawler uses to generate results, the tests will only determine the time it took to crawl the websites. *Table 1* shows the web crawler speed comparisons.

Table 1: Web crawler speed comparisons

Web Crawler	Min (s)	Avg (s)	Max (s)
dtSearch Engine	422.09	463.10	503.11
HTML Agility Pack	486.87	545.22	647.24
Abot	708.10	730.49	777.92
Weaver	-	-	-

All tests were carried out on the following system:

- Intel Core i7-2600QM 2.4GHz;
- 8GB Memory;
- 1Mb/s internet connection speed.

During testing it was noticed the Weaver web crawler was highly unstable and could not complete a single crawl test. Due to this instability the viability of the Weaver web crawler's architecture was called into question and was dropped from the investigation.

3.2.1.5. Essential features

When picking an existing component care must be taken to select a component that provides as much of the required functionality as possible. If a web crawler does not have some of the required functionality it must be added and tested which can take a significant amount of time. This can also increase the risk of the system becoming unstable, especially if the added functionality does not correctly interface with the used component.

For the proposed ORM service, the essential features are:

- An active web crawler: does the component provide a web crawler that is ready to use or must the user develop some additional functionality in order to use the system?
- Threading: does the web crawler have internal logic that support threading or must threading capabilities be added?
- Storage: does the web crawler have a method of saving the results or must the method be developed?
- Result generator: does the crawler have a method of generating results from the web crawl results?

The dtSearch Engine features a fully functioning web crawler that is capable of threading and can be immediately used upon installation. As the results that are gathered by the dtSearch Engine are stored in index files by using a unique algorithm the dtSearch Engine features a result generator capable of interpreting the index files. Conversely, as the HAP is a web crawler development kit it contains algorithms that support web crawling but does not contain a functioning web crawler that can be used upon installation, nor does it contain any threading capabilities, storage algorithm or result generator. Lastly, the Abot system improves upon the HAP by adding several features such as threading capabilities but also does not feature a fully functioning web crawler, storage algorithm or result generator.

By using the information in the previous paragraph a feature matrix can be completed that will compare the functionality of each web crawler. The feature matrix is shown in *Table 2*.

Table 2: Essential features comparison

	Functional web crawler	**Threading capable**	**Storage algorithm available**	**Result generator available?**	**Final Count**
dtSearch Engine	Yes	Yes	Yes	Yes	4/4
HAP	No	No	No	No	0/4
Abot System	No	Yes	No	No	1/4

From *Table 2* it can be seen that the dtSearch Engine contains all the essential systems.

3.2.1.6. Additional feature comparison

Additional features are features that are not essential to the ORM service but might be used later on.

The dtSearch Engine contains the following additional features:

- The user is able to set the crawl depth for each individual web site, e.g. setting the crawl depth for Website A to 1 and the crawl depth for Website B to be 2.
- A score will automatically be calculated by the dtSearch Engine when results are generated. The score will indicate the likelihood of the usefulness of the result.
- When a result is generated it will automatically include the URL, the page and the paragraph where the result was located.

- The user will be able to regenerate the web page where results were located. The dtSearch Engine accomplishes this by using the indexed data.
- Can index documents such as Word or PDF pages.
- Contains logging capabilities allowing the user to see exactly which web sites were detected and scanned.
- Supports a polite crawling policy.

The HTML Agility Pack provides the user with the following additional features:

- Allows the user to build a document model which will provide various information such as titles, headers, footers and content and be extracted.

The Abot system has the following additional features:

- Contains logging capabilities allowing the user to see exactly which web sites were detected and scanned. This is mostly used whenever a web crawler developed by using the Abot system malfunctions.
- Can limit the number of web pages to crawl and the number of times a HTML page may be redirected. This is to prevent pages from looping indefinitely and will lessen the number web pages to scan.
- Supports a polite crawling policy.

From the above description it can be seen the dtSearch Engine has 7 additional features that may be of use, the HTML Agility Pack 1 and the Abot web crawler 3.

3.2.1.7. Customizability and cost comparison

The dtSearch Engine is closed-source. Its source code can therefore not be modified and has to be bought. The user must use the settings that are provided by the dtSearch Engine in order to configure it. Unlike the dtSearch Engine the HTML Agility Pack and Abot system are open-source and therefore can be customized to a great extent. There are no cost involved with acquiring and using the open-source web crawlers.

3.2.1.8. Web crawler conclusion

To summarize, the dtSearch Engine is a very fast but expensive commercial web crawler that is actively being maintained and developed. The HTML Agility Pack allows the user to develop a web crawler that will fit all requirements of an ORM service but it is in danger of being dropped by the community. The Abot system is a refined version of the HTML Agility Pack that is actively being maintained but still requires the user to develop his or her own web crawler.

A weighted average will be used to determine which web crawler will be best suited for this project, which is as follows:

- Speed: 30%. As stated in Section 1.3, one of the main goals of the ORM service is that it must be fast enough to keep the information as updated as possible. As such the web crawler must be capable of both scanning the web and generating results at a rapid pace;
- Essential Features: 30%. Due to the scope of this project the implementation of the web crawler must not take too long. Implementation of the web crawler includes: designing, development, testing and integration into the ORM service;
- Additional Features: 15%. Additional features may increase customer satisfaction;
- Customizability: 15%. How easily the web crawler can be adapted to fit the ORM specification.
- Cost: 10%. The entire software package must be cost affordable for any potential client.

These factors are not the only ones that can be used to compare the web crawlers but are deemed the most important for this project. The results are shown in *Table 3*.

Table 3: Web crawler decision matrix

Category	dtSearch Engine	HTML Agility Pack	Abot
Speed (30%)	30	25	19
Essential Features (30%)	30	0	8
Existing Features (15%)	15	2	7
Customizability (15%)	7	15	12
Cost (10%)	0	10	10
Total (100%)	**82**	**52**	**56**

In *Table 3* the weighted averages were applied to each web crawler. It can be seen that despite its cost the dtSearch Engine will be the best web crawler to use with the ORM service. While the HTML Agility

Pack and the Abot web crawler may be almost as fast and free, it will take a lot of time to develop an entire web crawler with all the required features.

3.2.2. Social Network Crawlers

As explained in Section 2.3.2, API's must be used to extract information from social networking sites. The following sections will detail how the ORM system will interface with the social networking sites;

3.2.2.1. Twitter API

The Twitter API is the name given to a collection of separate components developed by Twitter that each has a separate function. Two of the components within the API allow the user to extract information from the Twitter databases, namely the REST API [15] and the Streaming API [16]. Both components will provide the user with data from the Twitter database by different means: the REST API will allow the user to extract information on a query basis whereas the Streaming API will open a continuous feed of data.

Querying the Twitter database with the REST API will grant the user access to historical data but is slow and will cause significant internet overhead. Additionally the REST API limits the user to 180 queries per 10 minute time frame. In contrast, the Streaming API gives the user low latency access to Twitter's global stream of data; this will present the user with results as they become available but the user will not have access to historical data. This is shown in *Figure 14*

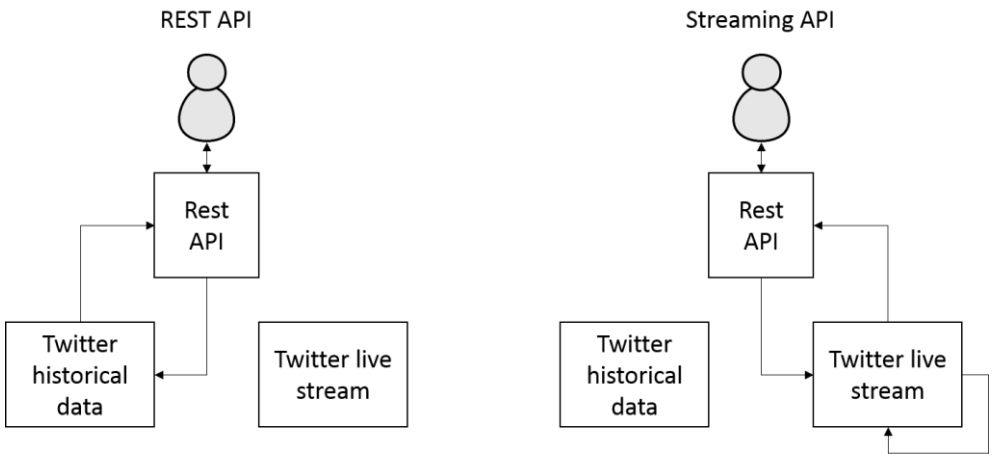

Figure 14: Twitter API Comparison

Due to its restrictions and limitations the REST API will not be suitable and therefore the ORM system will make use of the Streaming API. This will leave the user unable to access any historical data, but upon further investigation it was noticed that the user can only extract the last 100 records from the Twitter database by using the REST API. Therefore it would be better to collecting results by using the Streaming API and save all results to a local database where they can be accesses at any given time.

To aid with Twitter integration programmers have developed several free components that can access the REST API and Streaming API. For this project two such components were identified, namely *TweetInvi* and *Linq2Twitter*. Both of these components were investigated to see if they could be of use and if none of these components are found to be viable a custom Twitter integration method will have to be developed.

3.2.2.1.1. TweetInvi and StreamInvi

TweetInvi [28] is a .Net software development kit (SDK) developed in C# by the TweetInvi Team. The SDK consists of two components: TweetInvi (a component of the TweetInvi package) and StreamInvi.

TweetInvi provides the user with an easy way to interface with the Twitter REST API whereas StreamInvi provides the user with a way to interface with the Twitter Streaming API. As it was decided that the ORM system will use the Twitter Streaming API the system will only make use of the StreamInvi component.

By using StreamInvi the user can access two types of streams; a sample stream which contains unfiltered Twitter data, or a filtered stream which only presents the user with data that matches a set of conditions.

Method of operation

To use StreamInvi the user must first pass a validation check by providing the four Twitter keys, as discussed Section 2.3.2.1. After the application has been verified the user must specify whether to use the sample stream or filtered stream, provide the keywords which will serve as filters and start the process. The stream will continue scanning Twitter's public data and report a result once a keyword has been detected. This result will be saved to the computer's internal memory where it will be processed and saved to a database from where it can be accessed later on. The stream will continue scanning Twitter until the user stops the process or the computer is shut down. This process is demonstrated in *Figure 15.*

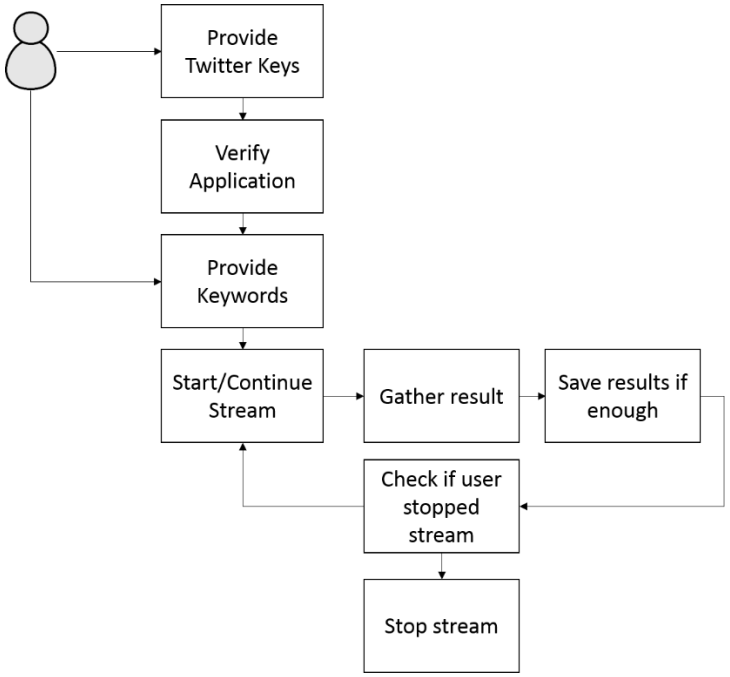

Figure 15: StreamInvi scanning architecture

Advantages and disadvantages

The StreamInvi component is very easy to implement within an application. The component also allows the user to easily add more keywords to its filter and gives the user complete control over the results, allowing the user to either save the results or implement additional logic before saving. Additionally, StreamInvi also includes several debugging capabilities, which allows the user to identify and handle any potential errors that may occur during software execution.

However, information provided by the StreamInvi component will only identify the user that has made the tweet, the time of the tweet and the actual message. Additional filters will have to be developed to detect the language of the tweet in order to filter out incompatible results. TweetInvi and StreamInvi are both open-source components; while the components are actively being maintained the community may drop the project in the future if they lose interest, which could lead to maintenance problems.

3.2.2.1.2. Linq2Twitter

LINQ is a .NET Framework component developed by Microsoft that adds native data querying capabilities. It is commonly used to extract data from arrays, XML or enumerable classes by using a query structure similar to SQL.

By using LINQ's capabilities a community group has developed an open-source library named *Linq2Twitter* [29]. The library uses LINQ syntax to handle queries from the Twitter API and present the user with the results. By using the Linq2Twitter library the user can develop Twitter applications capable of sending and receiving posts from their timeline as well as receive a stream of continuous Twitter posts.

Method of operation

In order to start using the Linq2Twitter library the user must initialize the library and verify the application by using the four Twitter keys. After the application has been verified the user must provide the keywords that will specify which sentences must be returned by the Twitter filtered stream. Once these parameters have been set the stream can be started.

The stream will scan Twitter and provide the user with any results that contains one or more of the specified keywords. All results will be saved in the computer's internal memory where it will be processed and saved to a database where it can be accessed from later on. The stream will continue scanning Twitter posts until the user stops the process or the computer is shut down.

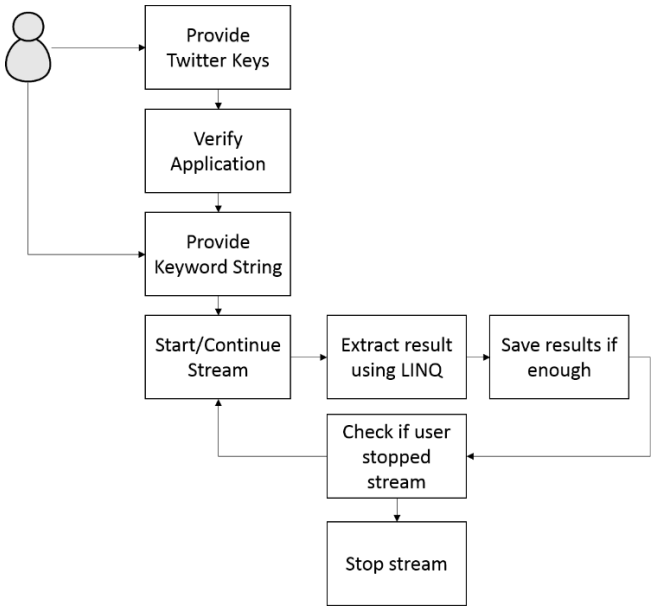

Figure 16: Linq2Twitter search method

Advantages and disadvantages:

Like StreamInvi, Linq2Twitter contains a lot of functionality such as Twitter authentication that would otherwise have to be programmed manually. Once a result has been provided the user will be able to

implement any custom functionality that will process the result. Additionally, Linq2Twitter has detected more results than StreamInvi (see Table 4) and the results contains additional information such as the language of the message, the number of replies to the message and the number of the message has been shared.

However, Linq2Twitter requires the use of LINQ syntax which is complex to use. Linq2Twitter is also an open-source component, and while it is currently being maintained the community may drop the project in the future.

3.2.2.1.3. Effectiveness comparison

Evaluating Twitter components are more difficult than evaluating web crawlers as the number of generated tweets vary by the second. To compensate both applications were run in parallel in order to determine which component would gather the most tweets within the allocated time. Both tests were executed twice.

Table 4: Twitter component comparisons

Test	StreamInvi	Linq2Twitter	Difference
Test 1	1747	1815	3.89%
Test 2	1939	2036	5.00%

All tests were carried out on the following system:

- Intel Core i7-2600QM 2.4GHz;
- 8GB Memory;
- 1Mb/s internet connection speed.

3.2.2.1.4. Existing features

StreamInvi has the following unique features:

- StreamInvi allows the user to easily extract the owner of a Twitter message as well as the date on which it was posted, the actual message and the keyword that was detected.

Linq2Twitter has the following unique features:

- Linq2Twitter provides the user a lot of information regarding Twitter messages, including but not limited to the Twitter ID, location and alias of the user that created the message, as well as how many times the message was spread by others.
- Linq2Twitter identifies the language of the message.
- It is indicated within the results whether the message is an original or a reply to another message.

From the above description it can be seen that Linq2Twitter has 3 additional features that may be of use while StreamInvi has only has 1.

3.2.2.1.5. Complexity

Implementing the StreamInvi component within a programming environment is very simple as there are no special techniques required in order to interface with the component. Conversely, using Linq2Twitter requires the use of LINQ syntax which is harder to implement and maintain.

3.2.2.1.6. Customizability

While both components are open source, it can be seen that Linq2Twitter can be more easily adapted to any situation than StreamInvi due to the large amount of information that results from Linq2Twitter contain. Should more information be required from the StreamInvi components the user would have to make the changes within the component's source code and recompile the entire component, whereas the user can simply extract the correct information from the Linq2twitter results without modifying the component itself.

3.2.2.1.7. Twitter component conclusion

To summarize, StreamInvi is very easy to implement and use but provides only limited information. Linq2Twitter provides more information and detects more results than StreamInvi, but is more difficult to implement.

To determine which component will be best suited for this project, a weighted average will be used. The weighted average is as follows:

- Number of results: 30%. As stated in Section 1.3, one of the main goals of the ORM service is that it must be fast enough to keep the information as up to date as possible. As such the Twitter component must be capable of retrieving results as they are being generated without losing any.
- Existing Features: 30%. Additional features may increase customer satisfaction;
- Complexity: 20%. How easy it is to implement and modify the software
- Customizability: 20%. How easily the component can be adapted to fit the ORM specification.

These factors are not the only ones that can be used to compare the components but are deemed the most important for this research. The results are shown in *Table 5*.

Table 5: Twitter weighted averages

Category	StreamInvi	Linq2Twitter
Amount of results (30%)	27	30
Existing Features (30%)	10	30
Complexity (20%)	20	10
Customizability (20%)	10	20
Total (100%)	**67**	**90**

In *Table 5* the weighted averages were applied to each Twitter scanner. It can be seen that while StreamInvi is quite fast and easy to implement, the limited information provided by the results make the component less viable than Linq2Twitter. If StreamInvi were to be used, additional filters would have to be implemented in order to filter out non-English results. As such, Linq2Twitter was selected as the Twitter scanning component for this research.

3.2.2.2. Facebook SDK

The Facebook SDK for .Net is the only component available to interface with the Facebook system from within the .Net environment. Another set of tools known as *Parse* is also available but must be purchased before it can be used. Additionally a test application has demonstrated that the Facebook SDK for .Net supports all the features required for the ORM service and will therefore be used.

3.2.3. String similarity formula

Once the web or social networking crawlers initialized, all results of a specific keyword will be returned. However, a certain portion of the web crawler results will have insignificant meaning such as tags within web pages.

Manually filtering out such results will cost the user valuable time. Therefore similarity filters are often included alongside web crawlers, which are filters that indicate the number of operations that are needed to transform one string into another. There are varying kinds of string similarity filters; some filters such as the Levenstein-Munkres algorithm [30] [31] analyse the strings and determine the number of changes that are needed to literally translate a string into another, while others such as proposed by Mihalcea [32] and Li [33] extract and compare sentiments from the given strings.

As the user would simply wish to filter out sentences that are deemed too similar to its keyword a technique that will look at the physical sentence will be needed. As such the string similarity techniques proposed Mihalcea [32] and Li [33] will be not be ideal for this solution and a technique based on transforming one string into another will be used. Two such algorithms were identified, namely the Damerau- Levenshtein algorihm [34] and the Levenshtein-Munkres algorithm [30] [31].

The Damerau-Levenshtein algorithm was first proposed in 1964 by Fred. J. Damerau in order to cope with errors in a coordinate indexing and retrieval system [35]. Damerau distinguished between four different edit operations, namely insertion, deletion, substation of a single character and transposition. According to his research these four edit operations correspond to more than 80% of all human misspellings. However, his research only took misspelling that requires at most 1 edit operation into account. The corresponding edit distance was taken into account by V.I. Levenstein, which developed the Levenshtein algorithm that can be used to measure the similarity between two strings by calculating the least number of edit operations that are necessary to modify one string into becoming another [30] [36]. When combined, the Damerau-Levenshtein algorithm can be used to calculate an edit distance that allows multiple edit operations. This algorithm is mostly used to improve spell checkers, although

the Damerau-Levenshtein algorithm has also been used in biology to measure the variations between DNA stands.

Like the Damerau-Levenshtein algorithm, the Levenshtein-Munkres algorithm also uses the Levenshtein algorithm, but instead combines it with an algorithm developed by H. Kuhn and J. Munkres. The Kuhn-Munkres algorithm is an algorithm capable of solving linear assignment problem (LAP) instances. The algorithm was first published by H. Kuhn in 1955 and later improved upon by J. Munkres in 1957 [31].

The Levenshtein-Munkres algorithm calculates its similarity score by following three steps:

- Breaking up the result and keyword into a list of their individual terms. These terms are called tokens.
- Computing the similarity between the tokens within the same list. This is done by using the Levenshtein algorithm. The algorithm is used to compare the similarity between the tokens by providing a set of rules that calculates the cost of changing a token to another by using a series of one step operations. Each one-step operation has an associated cost; substitution costs 2 units whereas cost of insertions and deletions is one unit.
- Computing the similarity between the two token lists. This is done by the Kuhn-Munkres algorithm.

When executed the algorithm will give the user a percentage that will indicate the similarity between the sentences.

While both algorithms measures the distance between different sentences the Levenshtein algorithm will be most ideal for this project as it calculates the similarity between the sentences in terms of a percentage. The Damerau- Levenshtein algorithm only provides the user with the number of edit operations that are needed to change one sentence into another, which will make it difficult to determine whether the sentences differ enough from its keyword. This is shown in the following result:

Sentence: It's Audi

Keyword: Audi

Damerau-Levenshtein distance: 4

Levensthein-Munkres similarity: 50%

As can be seen from the example above the Levenshtein-Munkres algorithm is easier to interpret than the Damerau-Levenshtein algorithm. Therefore the Levenshtein-Munkres will be used within this project.

3.2.4. Sentiment analysis tool

To finish the ORM process the sentiment of a record must be calculated. After the record has been processed by the similarity filter its sentiment must be calculated by a sentiment analysis tool. There are various sentiment analysis tools available such as the *AlchemyAPI* [37] and the *Salience Engine* provided by *Lexalytics.*

AlchemyAPI was founded in 2005 with the aim to provide advanced language analysis tools for its customers. In order to aid developers AlchemyAPI provides the user with an SDK that can be used within most programming environments. In order to the use the SDK the user first needs to register for an account on the AlchemyAPI system, which will provide the user with 2 keys. In order to determine the sentiment of a sentence the user must use the 2 keys to verify the application and provide the AlchemyAPI with the sentence. AlchemyAPI will proceed to analyze the sentence and present the user with the result.

AlchemyAPI is a commercial service. This project used the free version from AlchemyAPI which limits the user to 1000 requests per day. There are other options that allow the user to generate results for requests but these are not free. After the free version the cheapest version allows the user 5 000 requests per day for $250 per month. For more pricing options the user must contact the AlchemyAPI support.

The Salience Engine from Lexalytics provides the user with a multi-lingual text analysis engine that can be integrated into systems for business intelligence and social media monitoring. The Salience Engine supports various programming frameworks including the .Net Framework and includes many features, such as sentiment analysis, named entity extraction and summarization.

From features alone the Salience Engine would be better suited for this project, but in order to acquire the Salience Engine the user would have to contact Lexalytics. Additionally, there are no pricing opinion available on the website whereas the AlchemyAPI SDK can easily be acquired after registration.

Based on availability AlchemyAPI would be best suited for this project, but a thought must be given regarding its accuracy; if the SDK cannot accurately calculate the sentiments of the given texts it will be of no use for the project.

In order to determine the accuracy of the system it was used to calculate the sentiments of 36 random texts which were retrieved from Twitter. When the sentiment was manually determined by reading the message and determining whether its content is positive or negative, it was found that the AlchemyAPI has an accuracy of 77.78%. While this may not seem high enough, a study by Biz360 [38], a social media monitoring measurement and engagement platform, has shown that humans only agree to sentiments 79% of the time and even if a system would have an accuracy of 100% humans would disagree with the calculated sentiment 21% of the time. As the AlchemyAPI achieves an accuracy that matches human interpretation and can easily be acquired, it can be used for this research.

3.3. Revision of concept design

Some of the components chosen in Section 3.2 cannot be implemented as envisioned in the original concept design due to operating differently than expected:

- The dtSearch Engine cannot save data directly to a database. Instead the dtSearch Engine creates index files on the local storage device which can be used later on. To accommodate this change the Back-End needs to be changed so it can save data to both a local storage device and to the database and the Front-End must be changed in order to read and filter the data.
- The Alchemy API does not always identify the sentiment correctly, most notably where the message is written in a sarcastic tone. Therefore the user needs to have an option where he or she may review the sentiment or assign one manually.

In order to accommodate these changes the original design must be altered.

3.3.1. The Back-End

The only change made to the Back-End was to allow the dtSearch Engine to access a storage device where the engine may store the crawling indexes. The final Back-End design is shown in *Figure 17*.

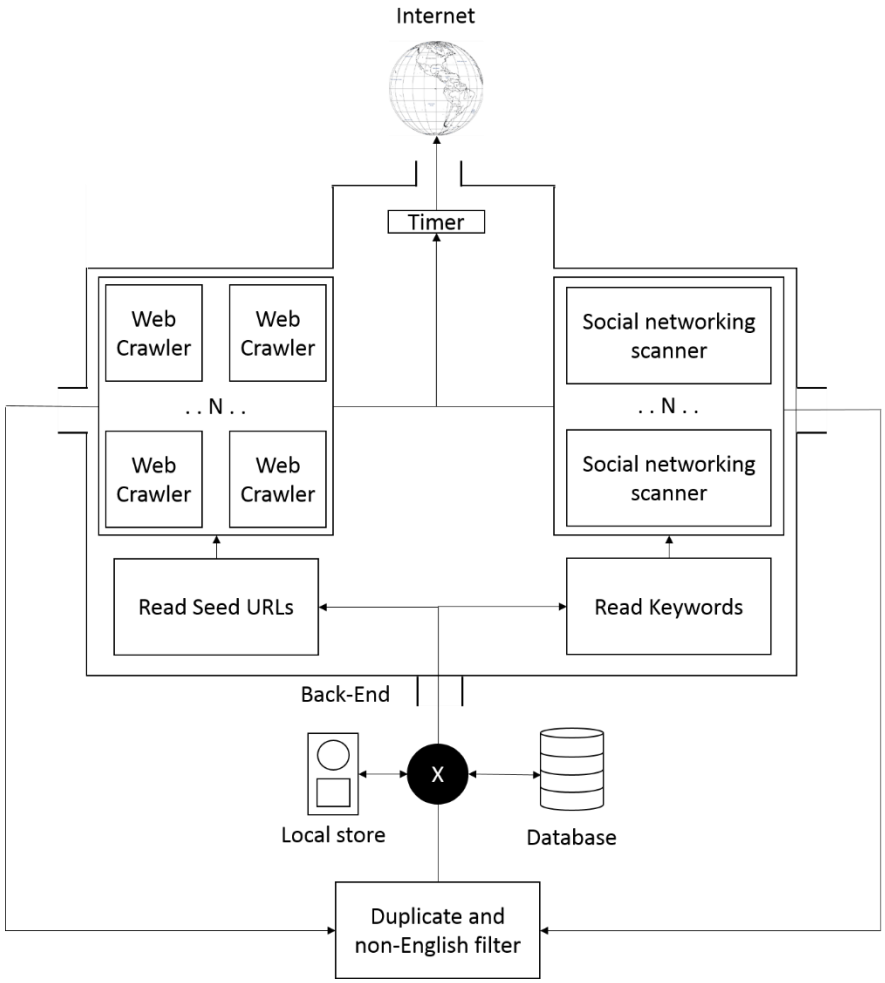

Figure 17: Final Back-End Architecture

From *Figure 17* it can be seen that the web crawlers, Facebook SDK and the Twitter API all acquire their information from the internet but whereas the Facebook SDK and the Twitter API save their data to the database, the web crawlers save their information to a separate storage device. The Front-End will be able to locate both the database and the storage device to extract the necessary data.

3.3.2. The Front-End

The Front-End design required two changes; it required functionality in order to read the web crawler indexes from the storage device, and an option for the user to either add an existing sentiment to a processed result or to edit an existing one. The final Front-End architecture is show in *Figure 18.*

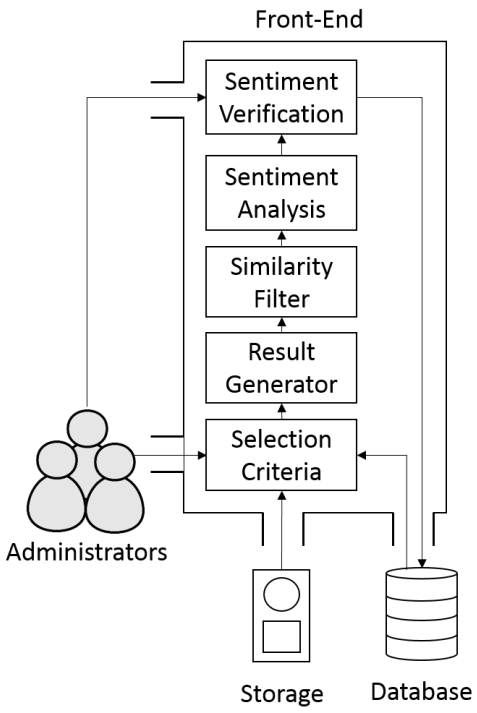

Figure 18: Final Front-End Architecture

From Figure 18 it can be seen the Front-End will acquire data from the storage device as well as the database. Included in the design is the ability to verify any sentiments that are generated before they are saved to the database.

3.3.3. The Website

No design changes were necessary for the website.

Chapter 4

Implementation

This chapter will detail the process how each of the tools that were chosen in Chapter 3 were implemented and configured within the ORM system as well as a motivation for each of the configurations. At the end of a chapter the overall flow of data will be illustrated.

4.1. The Back-End

The following components will be used in the Back-End

- The dtSearch Engine as web crawler;
- The Linq2Twitter library as Twitter API interface;
- The Facebook SDK for .NET as Facebook API interface;

4.1.1. Web crawler

As explained in Section 3.2.1.1, using the dtSearch Engine within a programming environment requires the system administrator to specify which websites must be scanned, the crawl depth of each website as well as the location of the storage device to which the dtSearch Engine can save its index files. The websites, crawl depth and storage location is specified by a system administrator using the Front-End, which will save the web crawl information to a database where it can be accessed by the Back-End. This will tell the dtSearch Engine where it should find its data and save the results, not how the dtSearch Engine should be implemented within a software application. This allows the user to implement the dtSearch Engine according to his own specifications.

For this research, a core requirement is to scan web pages as fast as possible in order to ensure the ORM system's information is always up to date, as stated in Section 1.3. During the concept design it was noticed that while the dtSearch Engine may quickly scan a few dozen web pages, scanning the thousands of web pages required for an ORM service will take significantly longer. This happens as the dtSearch Engine only scans a single page at a time while the other pages are kept in a queue. Such an implementation is called a Single Instance Web Crawler.

In order to allow more than one web site to be crawled at a time multiple instances of the dtSearch Engine must be initialized and the web pages divided amongst them. This will allow multiple web pages to be scanned simultaneously and each web crawler instance will have a shorter queue. However, using multiple web crawler instances will require more system resources than a single instance. This is called a Multiple Instance Web crawler.

The differences between a Single Instance Web Crawler and a Multiple Instance Web Crawler are shown in *Figure 19.*

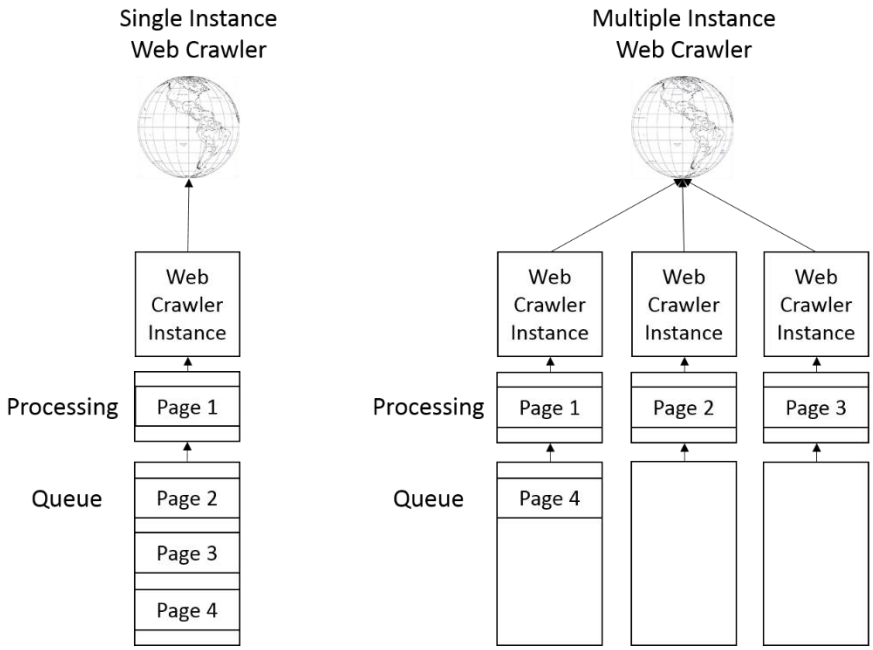

Figure 19: Single instance web crawler and multiple instance web crawler operation

In *Figure 19*, four web pages were assigned to both the Single Instance Web Crawler and the Multiple Instance Web Crawler. As seen the Single Instance Web Crawler can only process one page, namely *Page 1*, while *Page 2*, *Page 3* and *Page 4* are kept in the queue to be scanned. On the other side the Multiple Instance Web Crawler simultaneously processes *Page 1*, *Page 2* and *Page 3* while only *Page 4* waits in the queue. This means that a Multiple Instance Web Crawler which uses 3 instances is 200% faster than the Single Instance Web Crawler but uses more resources; for 3 instances running in parallel both the CPU and RAM will have higher loads and the internet connection will have more traffic.

This raises two questions:

- How to initialize more than 1 web crawler, and
- What is the optimal number of web crawlers to use?

There are several techniques capable of initializing more than 1 web crawler. The user may opt to execute several web crawler applications in parallel but this technique is tedious to implement and will cause unnecessary overheard as the operating system will have to keep track of each application and not just the web crawler implementation.

A better technique would be to make use of *threading*, which is a programming technique that creates sequences of instructions that are executed in parallel, which are called *threads*. This technique allows software to execute multiple instructions on the same processor, or split instructions across various processing units (CPU's). However, like all techniques threading has several disadvantages; initializing too many threads may cause the system to use too much of its resources and cause the software and

possibly the entire system to become unstable. Therefore the amount of threads must be managed and threads must be programmed to release all their resources after the completion of their task.

Using threading will allow the user to initialize multiple instances of the dtSearch Engine while minimizing unnecessary overhead. This brings us to the second question: what is the optimal number of threads? Initializing too many threads may increase the web crawling speed but will cause the system to become unstable while not initializing enough threads will unnecessary lower the number of web pages than can be scanned per second.

In order to determine the optimal number of web crawlers the impact of threading upon the system must be examined. To do this several tests were executed where the system resources and number of websites to be crawled were kept the same but the number of web crawlers were varied. In each test the execution time as well as the number of system resources used was measured. The results are shown in *Table 6.*

Table 6: Web crawler threading comparison, 1Mb/s

Threads	1	2	4	8	16	32
Time (s)	2381	1259,83	722,37	573,41	430,03	382,01
CPU (%)	1,5	2,5	4,1	5,0	5,1	6,5
Memory (%)	1,4	2,5	4,1	7,08	14,5	25,1
Network Average (KB/s)	23,67	57,02	98,16	127,18	133,28	125,03
Thread time minimum (s)	2381	1116,83	626,56	469,03	325,67	NA
Thread time maximum (s)	2381	1402,83	900,73	822,12	601,09	NA
Comment	Stable	Stable	Stable	Stable	Stable	Unstable

All tests were carried out on the following system:

- Intel Core i7-2600QM 2.4GHz;
- 8GB Memory;
- 1Mb/s internet connection speed.

From *Table 6* it can be seen that as the number of threads increased the amount of system resources used by the program increased as well but the web crawler execution time decreased.

From 8 threads onwards it can be seen that while the memory usage increased as predicted, the CPU usage differs very little. This occurs because the internet bandwidth has been reached. As such, 8, 16 and 32 threads all extracted the same amount of data from the internet and therefore the CPU did not have to do any additional processing aside from maintaining the web crawlers.

In this test the ORM software became unstable at 32 threads. When investigated it was noticed that the limited internet bandwidth prevented all the crawlers from receiving data fast enough, which resulted in several of the crawlers timing out while their frontiers still contained URLs. To verify that the timeouts did indeed occur because of the insufficient internet bandwidth another test was carried out using a 16Mb/s internet connection.

Table 7: Web crawler threading comparison, 16Mb/s

Threads	1	4	16	32	64
Time (s)	2121,41	538,25	124,29	64,56	48,81
CPU (%)	2,5	5,1	6,1	7,5	Unknown
Memory (%)	32,4	35,1	45,5	56,1	Est. 61%
Network Average (KB/s)	34,51	151,65	628,95	1008,84	Unknown
Comment	Stable	Stable	Stable	Stable	Stable

All tests were carried out on the following system:

- Intel Core i7-2600QM 2.4GHz;
- 8GB Memory;
- 16Mb/s internet connection speed.

From *Table 7* it can be seen that the internet bandwidth does indeed have an influence on the stability of the system. With a 16Mb/s internet connection the system was stable for 64 threads whereas the system became unstable at 16 threads when a 1Mb/s connection was used. The CPU network usage for the 64 thread test could not be calculated as the software did not execute long enough in order to get an average reading. Further tests showed that the system could support up to 128 crawlers before becoming unstable.

The best way to determine the optimal number of threads that can be used for this system is to derive a formula that can be used to calculate the number of threads. From *Table 6* and *Table 7* it can be seen that the main factor limiting web crawler performance is the internet bandwidth. As such, the formula will use the internet bandwidth to determine the optimal number of web crawlers.

Using the values from *Table 6* and *Table 7*, it can be seen that each web crawler requires an estimated 35kB/s in order to function correctly. Therefore, an internet connection of 1Mb/s (125KB/s) can theoretically supposed 4 web crawlers. However, *Table 6* showed a 1Mb/s can support up to 16 simultaneous web crawlers. Upon investigation it was noticed that while the internet traffic from a web

crawler may peak at 125KB/s, there are times when the web traffic is lower than indicated by the average. This is demonstrated for 2 web crawlers in *Figure 20*.

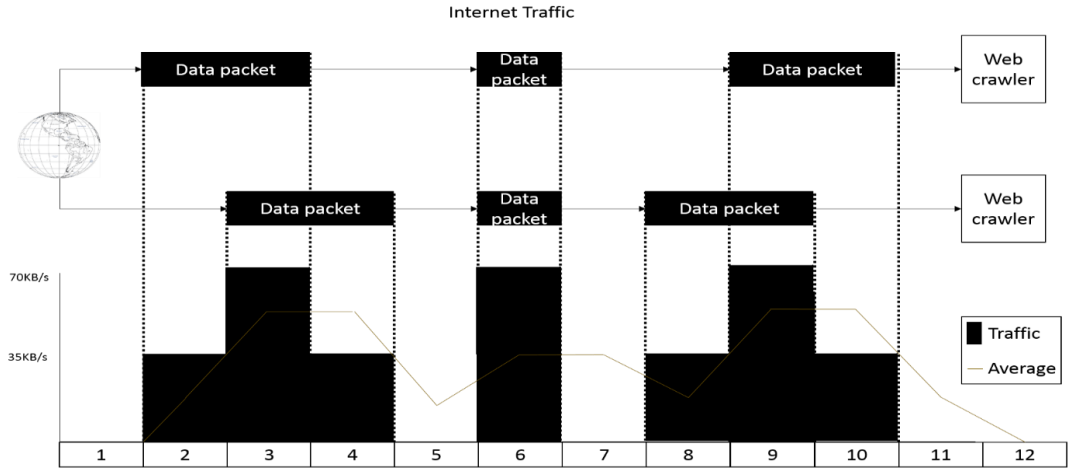

Figure 20: Web crawler internet traffic

Figure 20 shows a possible scenario for network traffic that may occur when two web crawlers are active. Graph properties:

- X-Axis: Time units
- Y-Axis: Used bandwidth
- Black blocks: Immediate download speed
- Golden line: Download average

Theoretically, two web crawlers will use up to 70KB/s. As shown in *Figure 20* this does happen from time to time, as seen at time unit 3 and time unit 9, but there will also be times when the web crawlers will process the information that has been downloaded and will not download new information. At such a time the moving average will give a false reading that will indicate that data is still being downloaded even though the network line will be inactive; this will happen as the moving average uses historical data in its calculations.

In order to properly optimize the line bandwidth the slots where the internet traffic is low must be utilized to provide more web crawlers with data as shown in *Figure 21*.

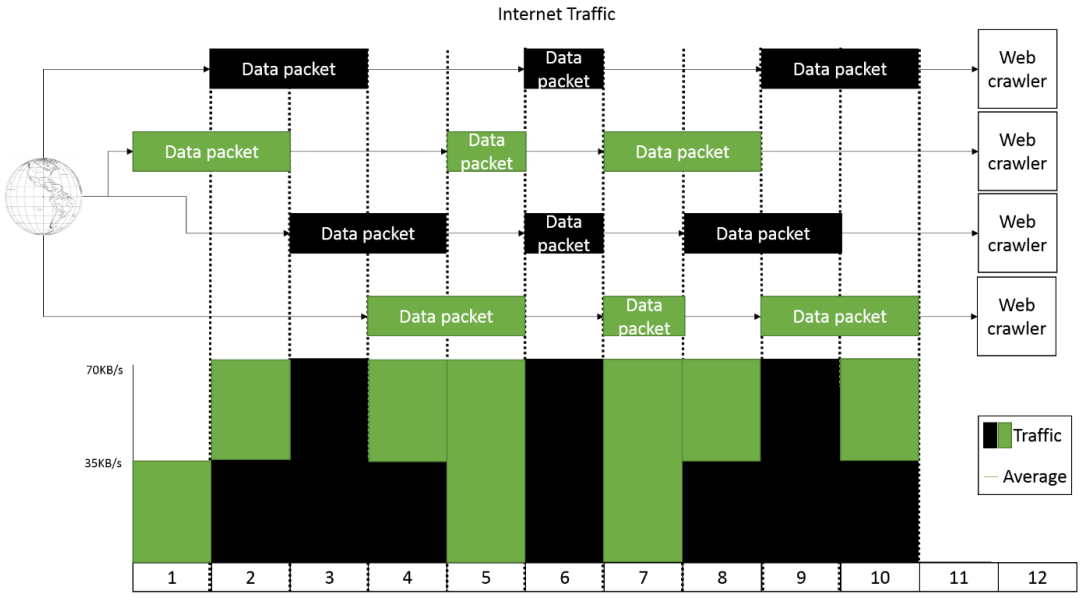

Figure 21: More efficient internet traffic

Graph properties:

- X-Axis: Time units
- Y-Axis: Used bandwidth
- Black blocks: Immediate download speed
- Green blocks: Additional web crawlers

In *Figure 21* the number of web crawlers were doubled. This resulted in more efficient use of the internet bandwidth; when specific crawlers indicated by the black squares are busy processing the information they have received the web crawlers indicated by the green squares will proceed to download their data.

Figure 20 and *Figure 21* suggests that the number of web crawlers calculated by dividing the line speed by 35KB/s can be squared in order to make maximum use of the internet connection. This would confirm why 16 web crawlers were able to operate on a line that otherwise would have supported only 4.

Therefore, using the information from *Table 6*, *Table 7*, *Figure 20* and *Figure 21*, the general method for determining the number of web crawlers can be summarized as:

- Determining the line speed in KB/s.
- Assuming each web crawler requires 35KB/s in order to operate, calculate the number of web crawlers (X) by dividing the line speed by 35KB/s and rounding the answer to the closest number.
- As a 70KB/s line could support 4 web crawlers instead of 2 and a 128KB/s line 16 instead of 4, the number of web crawlers must be squared.

Using this information, *Equation 1* can be used to determine the optimal number of web crawlers per implementation:

$$Number\ of\ Crawlers = \left(\frac{LS * 0.125}{35}\right)^2$$

Where *LS* = Internet connection speed in kb/s

Equation 1: Number of web crawlers for internet connection speed

Using *Equation 1* shows that a 1Mb/s internet connection will be able to support 16 web crawlers, as proven by *Table 6* and that a 16MB/s internet connection can support 118 web crawlers, which was proven by a separate test running 128 threads.

It should be noted that *Equation 1* is merely used as a close estimation; the actual amount of data that a web crawler uses depends on the websites that has been assigned to crawl, for example some websites can only be downloaded at 30KB/s while others may download at 40KB/s. This is demonstrated with the test using the 16Mb/s internet connection; according to *Equation 1* the web crawler will run 118 threads but the test proved 128 threads were possible.

Therefore, when selecting the active number of web crawlers the system administrator is encouraged to use this formula as a guideline, not as a definitive answer. The number of active threads can be set at any time the web crawlers are not active by using an interface within the Back-End.

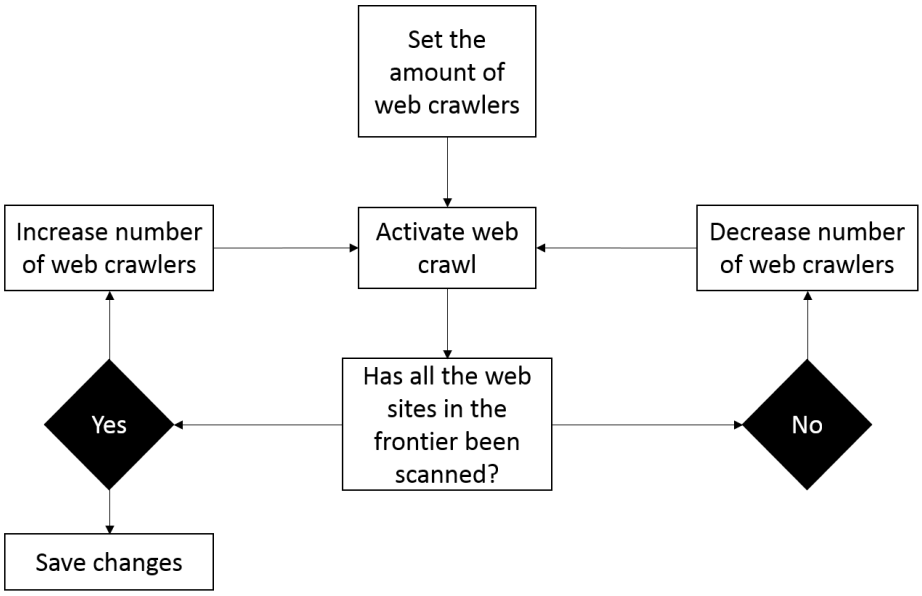

Figure 22: Verify amount of web crawlers

The method suggested in *Figure 22* is a possible way of verify whether the internet connection can number of active web crawlers. Using *Equation 1* the system administrator can set the initial amount of web crawlers and initiate the web crawling process. To verify whether the internet connection can support the number of active web crawlers it would not be necessary to initiate a full crawl of all the websites as the entire verification process will take too long. Instead the amount of websites should be chosen so that each web crawler has 5 websites in its frontier. A different number of websites per crawler can be used but during development it was found that 5 websites per web crawler was ideal.

Once the web crawling process has finished the system administrator must verify whether all the websites has been scanned. A quick way of accomplishing this is by determining whether the last website that has been scanned is the last website allocated to the web crawler. If not all of the websites were scanned the internet connection does not have sufficient bandwidth for all the web crawlers and as such the number of web crawlers must be decreased. If the web crawl was successful the system administrator can either opt to save the results or increase the amount of active web crawlers and restart the process.

4.1.2. Twitter API

Unlike the web crawlers the Linq2Twitter component does not access historical data but connects to live stream of Twitter data and waits for data that matches specific criteria. The streaming process can be started once the required Twitter keys had been provided and the application has been verified. Once started, the application will wait until it receives a record containing one of the required keywords. When a record is received it will be presented to the user for processing while the library will continue waiting for more results.

Like before, there are various methods to handle the records:

- Should the record be processed or directly saved to the database?
- Should each record be saved to the database as it is retrieved or should records be saved in a group?
- Will a multithreaded application have any benefits?

Whether the record can be processed or not will depend on whether the processing will have any influence on the number of records that are retrieved from Twitter. To determine this two tests were carried out; in the first test the Twitter records were directly saved to the database with no processing at all, while in the second a record viability filter was implemented. The record viability filter performs the following checks:

- Is the language of the Twitter record English?
- If yes, has the record been detected before?
- If no, does the message content contain the required keyword?
- If yes, store to memory.

The test was carried out 4 times; twice without the filter and twice with the filter. Each test was carried out for 10 minutes. The results are shown in *Table 8.*

Table 8: Twitter record process; no filter vs filter

Keyword	**No Filter Test 1**	**No Filter Test 2**	**Filter Test 1**	**Filter Test 2**
Correct keywords detected	1261	1438	611	572
Keyword not detected	0	0	12	24
Non-English language	0	0	585	615
Duplicate entries	0	0	107	142
Total Amount of Tweets	1261	1438	1315	1353

All tests were carried out on the following system:

- Intel Core i7-2600QM 2.4GHz;
- 8GB Memory;
- 1Mb/s internet connection speed.

From *Table 8* it can be seen that including a record viability filter will not influence the number of results and therefore it will be included in the Back-End.

Once a result has been processed it must be saved to a database to be used by the Front-End. This brings us to the second question, namely should each record be saved to the database as it is retrieved or should results be saved in a batch?

In order to decide which technique would be better the processes involved with each saving technique must be analysed. This is shown in *Figure 23*.

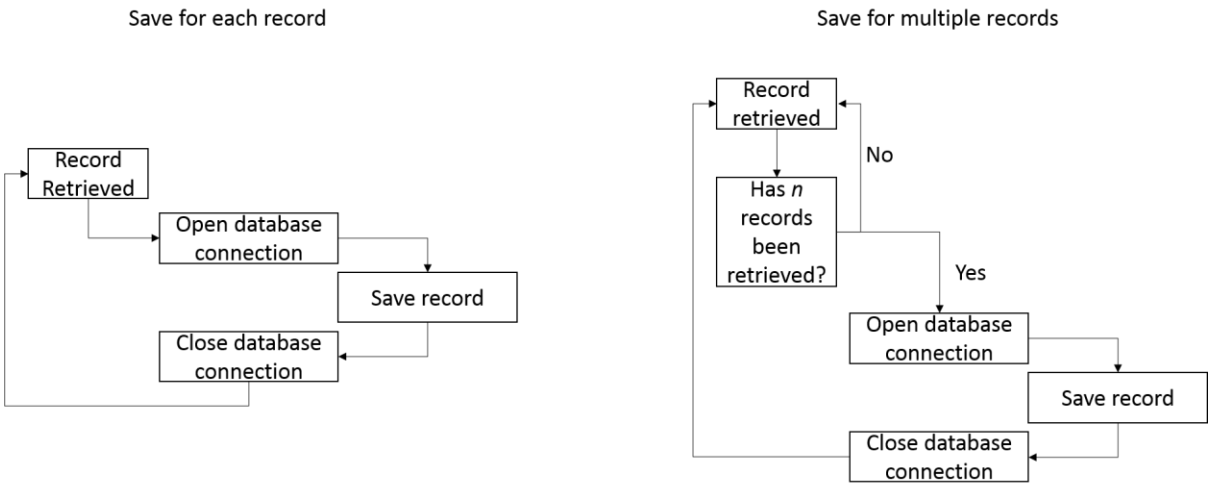

Figure 23: Database save methods comparison

For both methods it can be seen that a connection to a database must be created in order to access the database. Should the user wish to save a record as it is presented a database connection will have to be created every time, which will lead to a significant amount of overheard required just to initialize the database connection. This will slow the program, especially if the database is located on a remote server. It will be possible to create the connection once and use it whenever a record has to be updated, but this will still lead to a significant number of database statements that will pass the necessary information to the database. It might also cause the system to become unstable as the database connection might close if it remains unused. This might happen as the user does not know when the Twitter API will get a result.

It will therefore be better to store a certain number of records in the system's memory and save the records to the database once enough results have been accumulated. This will result in one connection to the database and one statement to pass all the information, thereby lessening the amount of overhead.

During development it was decided to save the data to the database once 250 records have been accumulated. While this number may be altered at will it was discovered that using 250 records will not slow the system and it will ensure that information will be rapidly saved to the database. It will also lessen the number of records that might be lost in case of a system failure such as a power outage of loss of internet connection.

A final thought must be given to whether multithreading might be of use. Because results are only presented as they are found, adding additional threads will not yield any additional rewards as the records might not be present at the available time. As more keywords are added to the system there might come a time when the library will start skipping records. At such a time multithreading techniques may be added but at this time the Twitter API will be restricted to a single thread.

4.1.3. Facebook API

As stated in Section 2.3.2.2, using the Facebook API will require the user to provide 2 authentication keys, after which the user will be able to perform queries using the API.

The Facebook API provides the user with two methods of retrieving public posted data; the Graph API, which is the primary method application to access the Facebook servers, and a Public Feed API which presents the user with a stream of user and page status updates.

The Graph API lets the user retrieve information from the Facebook servers via queries, similar to the Twitter REST API (Section 3.2.2.1). After the user has authenticated the application, the user may use the Graph API to make various requests including but not limited to searching public posts, interacting with the current user's Facebook profile and handling various errors. On the other hand the Public Feed API provides the user with a stream of public status updates as they are posted on Facebook.

As before it will be tempting to make use of the Public Feed API as only a single connection will be used over which continuous updates are passed. Unfortunately, the Public Feed API is not capable of providing the user with continuous posts. Instead the Public Feed API will only present the user with limited information regarding user and page status *updates*. To retrieve further information the user will have to use the Graph API instead. This is fully explained in the official Facebook API documentation [39] (please note that this link uses a secure connection. In order to view the page and its content the user will have to register a Facebook account).

Due to this limitation the Public Feed API will not be of use to the ORM service and therefore the software will make use of the Graph API.

To use the Graph API within an application the user must first provide the 2 authentication keys to authenticate the application. If authentication is successful the user will be presented with an Access Token, which must be used with every subsequent request. In order to retrieve information regarding the latest public posts the user must use the Graph API to perform a *search* request. This is accomplished by accessing the Search functionality from the Graph API by using the following URL:

https://graph.facebook.com/search?q=QUERY&type=OBJECT_TYPE

In order to specify the keywords that must be detected the user must replace "QUERY" with the specific keywords in a comma separated string. The "OBJECT_TYPE" specifies the information for each result that must be returned. A sample query is shown below:

https://graph.facebook.com/search?q=Microsoft,MTN&type=id,from,caption,message

This query will present the user with the unique Facebook ID, information regarding the user that created the message, the title of the message and the message itself from the latest posts in which *Microsoft* or *MTN* feature as keywords.

Each query will return the latest 25 results. This is a restriction built into the Graph-API over which the user has no control. This may lead to several questions:

- Will 25 records for each query be enough?
- If not, is there any way to ensure a maximum amount of results will be retrieved while staying within Facebook's restrictions?

To answer the first question the tempo at which new Facebook public posts are generated must be determined. This was accomplished by developing a test application that requested data for 5 keywords from Facebook each minute. Once the data has been retrieved it was passed through a filter which removed all the records that has previously been found. This test was carried out in 5 minute segments for two separate groups of keywords.

Table 9: New/Filtered Facebook results per minute

Keywords	Query 1	Query 2	Query 3	Query 4	Query 5
Batch 1	88/37	20/105	21/104	18/107	31/94
Batch 2	89/36	32/93	42/83	44/81	32/93

From *Table 9* it can be seen that for the initial query the number of new results is much higher than the number of filtered results as there are no historical data for the new data to be compared against. Any duplicates that occur are posts that have been repeated by multiple users on Facebook itself.

It can also be noted that from the second query onwards the number of new posts are significantly lower than the number of filtered results. Though the number of new posts depend on which keywords are used, the number of new posts for both Batch 1 and Batch 2 dropped more than 50% from the initial query. This shows that new data are being generated very slowly and that 25 results per query will not cause the user to lose any data.

4.1.4. Complete Back-End implementation

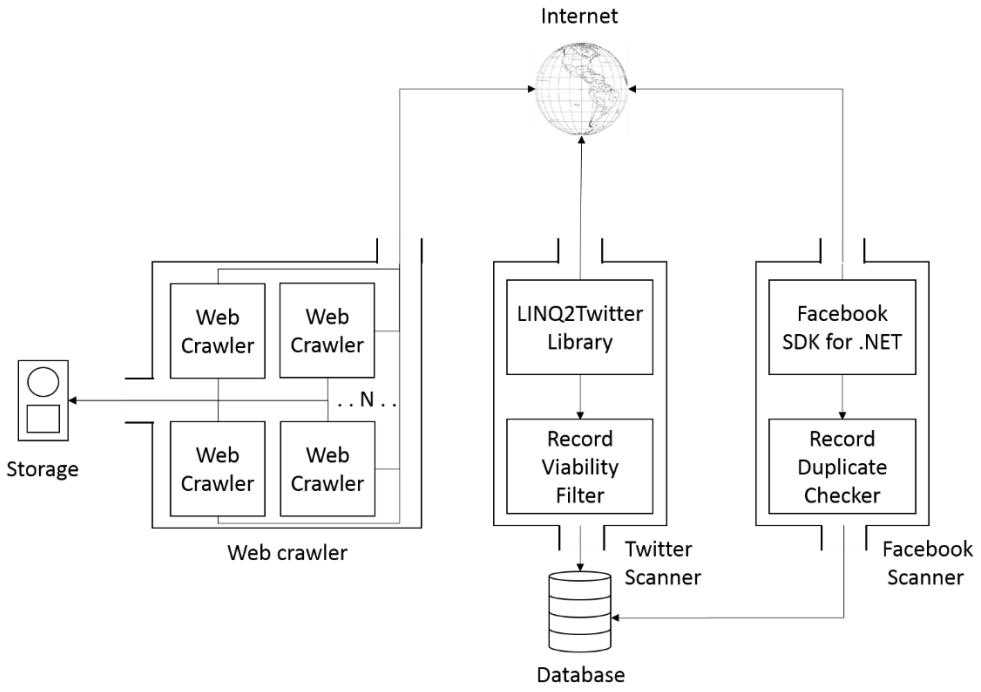

Figure 24: Complete Back-End implementation

Figure 24 shows the complete implementation of the Back-End as discussed in Sections 4.1.1-4.1.3. Each system is designed to function as a separate unit; this will ensure that even if one system malfunctions it will not influence the others. All results will be saved to a designated store drive as well as a database where the Front-End and Website can access the data.

To ensure the system will run at optimal capacity, the following system requirements are recommended:

- Quad-Core 2.4GHz CPU;
- 8GB Memory;
- 1TB HDD;
- 10Mb/s or faster Internet Connection.

Using the system described above will ensure that 256-512 web crawlers can be used simultaneously with the Twitter and Facebook Scanners.

4.2. The Front-End

The following components will be used in the Front-End

- The dtSearch Engine as result generator;
- The Levenstein-Munkres algorithm as similarity checker;
- The AlchemyAPI tool as sentiment calculator.

4.2.1. dtSearch Engine

As the index files are created by an algorithm unique to the dtSearch Engine, the software must use the result generating capabilities presented by the dtSearch Engine. The result generating capabilities of the dtSearch Engine allows the user to:

- search for specific or multiple keywords within an index file;
- search for instances where keywords are mentioned but others are not (Boolean searches);
- specify the number of results that has to be returned;
- search using synonyms of the specified keywords;
- regenerate the original web page.

To use the result generating capabilities the user must first specify which index files to use, the keywords that have to be detected and which features to use. Once specified the dtSearch Engine will scan through all the index files to detect the specified keywords and report back any results.

Unlike the components in the Back-End there are not many different implementations available for the dtSearch Engine result generator. When provided with the results from a web crawl using 788 websites with a crawl depth of 0 and 26 keywords the result generator took 3 seconds to generate 200 results for each keyword. This is deemed fast enough and therefore no multithreading capabilities will be included.

4.2.2. Similarity Filter

As discussed in Section 3.2.3 a similarity filter will be included as a portion of web and social networking crawler results will have insignificant meaning. These results include tags within a web page or tags under a picture from Facebook or Twitter results.

However, some of the Twitter and Facebook results that are passed through the Back-End filters will still not be viable for the ORM service, such as duplicate results that were detected in separate crawling sessions, or results that are too long and cannot be saved to the database without being truncated. Additionally, as Twitter messages are limited to 140 characters it was decided to remove results that contain more than three question marks. Such results will have no significant meaning as it will involve people asking each other questions with no answer in the text itself. Therefore the Front-End will contain two additional filters that will remove these results.

Implementing the similarity filter may not be difficult but tuning the threshold of the filter may take some time. If the filter threshold is too low not enough results will be passed through but if the filter threshold is too high too many useless results will be included within the results the user receives.

To determine the optimal threshold a test was executed. In the test the web and social network crawlers were executed for twenty minutes which provided 9 298 results. Of the 9 298 results 2 150 were duplicates and 5 results were too long and thus removed. The remaining 7 143 results were passed through a series of similarity filters, each with a different similarity threshold. The filtered out results were manually scanned and determined whether they were relevant to their keyword or not. The results are shown in *Table 10*.

Table 10: Similarity filter tuning table

Similarity threshold (%)	Amount of results passed through	Example of filtered out result
0	0	N/A
10	6384	I so love my apple devices
20	6996	I want an Apple TV
30	7115	Checkers after school
40	7131	An Apple a day
50	7140	N/A
60	7140	Checkers, ??
70	7142	Checkers ??
80	7143	N/A
90	7143	N/A
100	7143	N/A

Table 10 shows the similarity thresholds that was used as well as the amount of results that were passed through the filter and an example of a result that was filtered out. It can be seen that results that are more than 30% similar to their keyword contain very little significant meaning, but results that are below the 20% threshold are indeed viable for an ORM service, for example “I so love my apple

devices" contains a positive sentiment towards Apple products. Therefore results between 20% and 30% need to be investigated. For each category all the results were manually checked and the percentage results that contained significant meaning were calculated. The results are shown in

Table 11.

Table 11: 20% to 30% similarity filter investigation

Similarity threshold (%)	% Relevant results
20	93.4%
21	85.7%
22	64.7%
23	45.1%
24	40.0%
25	58.3%
26	42.8%
27	50.0%
28	75.0%
29	66.6%

Table 11 shows the similarity thresholds that was investigated as well as the percentage of results that was found to be relevant. It can be seen that the results near the 20% threshold offers more viable results than the 30% threshold but there are always a combination of viable and non-viable results. It will therefore be impossible to set the threshold to a specific value where all viable results will be retrieved and all non-viable results filtered out. As such the user must make a decision: if the threshold is set to 30% the user will obtain the largest number of viable results but the number of non-viable results will take some time to filter out. On the other hand if the user sets the threshold to 20% a large number of non-viable results will be filtered about but some viable results will be lost as well.

For this project the threshold will be set to 20% which will filter out most of the non-viable results while only losing a few viable ones, as proven by

Table 11. It should be noted that the values obtained in

Table 11 are specific to this test and may change depending on the current mood of the public or the keywords used.

A final thought can be given to multithreading, namely whether it is necessary. In the current test the similarity for all 9 298 keywords were calculated in 1.77 seconds. This is fast enough for everyday use and therefore multithreading will not offer any benefits.

All tests were carried out on the following system:

- Intel Core i7-2600QM 2.4GHz;
- 8GB Memory;
- 1Mb/s internet connection speed.

4.2.3. Sentiment analysis tool

Once the results have passed through the similarity filter their sentiments have to be calculated. As discussed in Section 3.2.4 the Alchemy API will be used for this research.

The only way to interface with AlchemyAPI is to use the library file that is provided when the user registers on the network. The library will allow the user to send a sentiment request to the AlchemyAPI servers that will process the given text and present the user with a result.

Unfortunately the AlchemyAPI does not allow batch processing; the user cannot pass multiple messages through a single sentiment request to be calculated together. This technique presents a problem for the ORM system. ORM systems require a large number of messages to be processed in the shortest amount of time in order to keep its information as up to date as possible.

To compensate for this the AlchemyAPI allows the user to send multiple requests at the same time, the number of which differs depending on the user's AlchemyAPI account. As this project uses a free account it can only pass up to 5 concurrent requests.

This begs the question: what is the optimal number of threads to use? In order to determine this several tests were executed where a number of messages were retrieved from the database and passed to the AlchemyAPI using a varying number of concurrent threads. Due to the restrictions of the free account this test was limited to using 100 messages. The results are shown in *Table 12*.

Table 12: AlchemyAPI multithreading

Amount of threads	Time (seconds)
1	61.42
2	32.25
3	30.86
4	31.57
5	31.65

The results in *Table 12* are not as expected. It can be seen that if 2 threads are used the calculation speed is indeed 100% faster, but from there on increasing the number of threads does not influence the calculation speed in any significant way whereas theoretically increasing the number of concurrent threads should bring the speed down even further. It is theorised that AlchemyAPI can only process a

limited number of results at a time and therefore implemented a queuing policy in order to process all the sent messages as fast as possible. AlchemyAPI was contacted in order to verify whether this is correct but no response has been retrieved at the time of writing.

4.2.4. Complete Front-End implementation

Figure 25 shows the final Front-End Architecture. It can be seen that a Front-End instance does not interface with anything except the local storage drive where the dtSearch Engine index files are located and the database where the Facebook and Twitter files can be extracted. As such several Front-End instances can be opened at the same time, allowing several system administrators use the system at the same time.

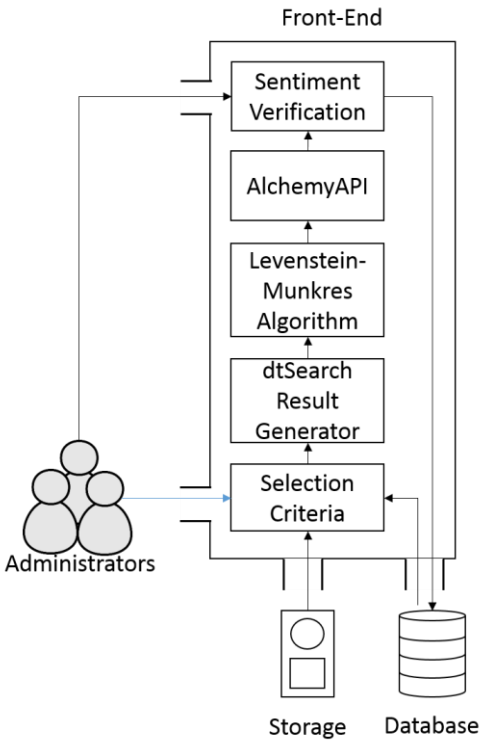

Figure 25: Final Front-End Architecture

4.3. The Website

The website will not make use of any third-party tools as it will only be used to display results that have been generated by the Front-End. For this reason the implementation of the website will not be discussed here.

4.4. Final method of operation

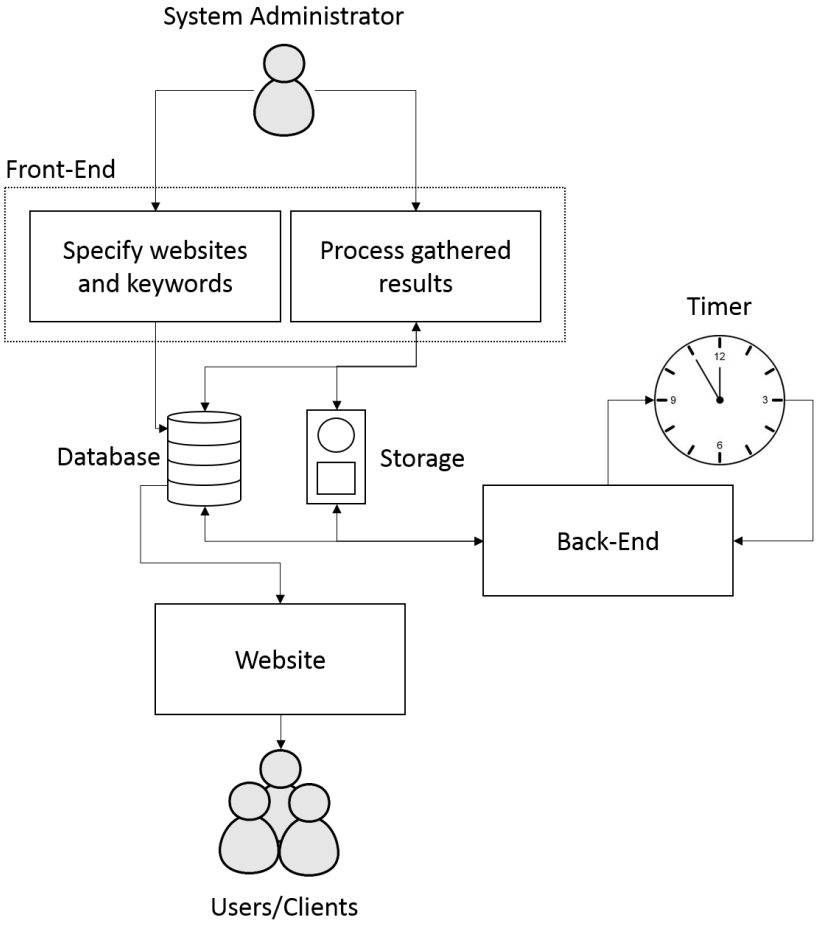

Figure 26: Final ORM operational flow

The operational flow of the ORM system can be seen in *Figure 26*. To start the ORM process the system administrator will specify a series of keywords as well as a series of websites and their crawl depths by using an interface in the Front-End. This data will be saved to the database, and will be used by the Back-End in order to acquire information from the web, as detailed in Section 4.1. The results from the

Back-End will be saved back to the database as well as a local storage drive such as a hard disk. After saving the results the web crawlers will scale down their processes and wait for a pre-determined amount of time to pass before restarting the crawling process. The Twitter and Facebook scanners are unaffected by the web crawlers and will remain active when the web crawlers scale down their processes. The results from each web crawl session will be saved in a different location in order to preserve historical information.

The system administrator can process the Back-End results by using a different Front-End interface, as detailed in Section 4.2. The results that can be processed depend on the Back-End status; if the Back-End web crawlers are idle all web crawler results can be processed but if the web crawlers are currently busy crawling the web the user can only process the results for a previous session. Twitter and Facebook results can be processed as they are saved to the database.

Results for the website will become available as the system administrator updates the database with the processed results.

Chapter 5

Results

The ORM system was evaluated by using it in a real world scenario similar as it would be used by a client. The scenario contained a list of keywords that must be detected as well as a list of websites and their respective crawl depths that must be scanned by the web crawlers. This chapter will detail the results that was retrieved by testing the ORM system in this manner.

5.1. Process

The scenario used in the evaluation consisted of 200 websites with a crawl depth of 1 and the following keywords:

- Spar
- Shoprite
- Wimpy
- Spur
- CNA
- Musica
- Woolworths
- Edgars
- Audi
- Jeep

These keywords were chosen as they span different sectors of the industry, namely general shopping, food, entertainment, clothing outlets and motoring. The websites were chosen by randomly selecting 200 websites from a list containing 788 websites in different categories.

As the ORM system would run on a server within a real world scenario, the ORM system was installed on a system with higher specifications than the one used during development. The "server" system has the following specifications

- Intel Core i7-3770K 3.0GHz;
- 16GB Memory;
- 1Mb/s internet connection speed.

5.2. The Back-End

To test the Back-End within a real world scenario, all three of the ORM components were run in parallel.

5.2.1. Web crawler

As a 1Mb/s internet connection was available, the Back-End was set to use 16 active web crawlers with the 200 websites divided amongst them. A graph indicating the execution time for each web crawler thread is given in *Figure 27*.

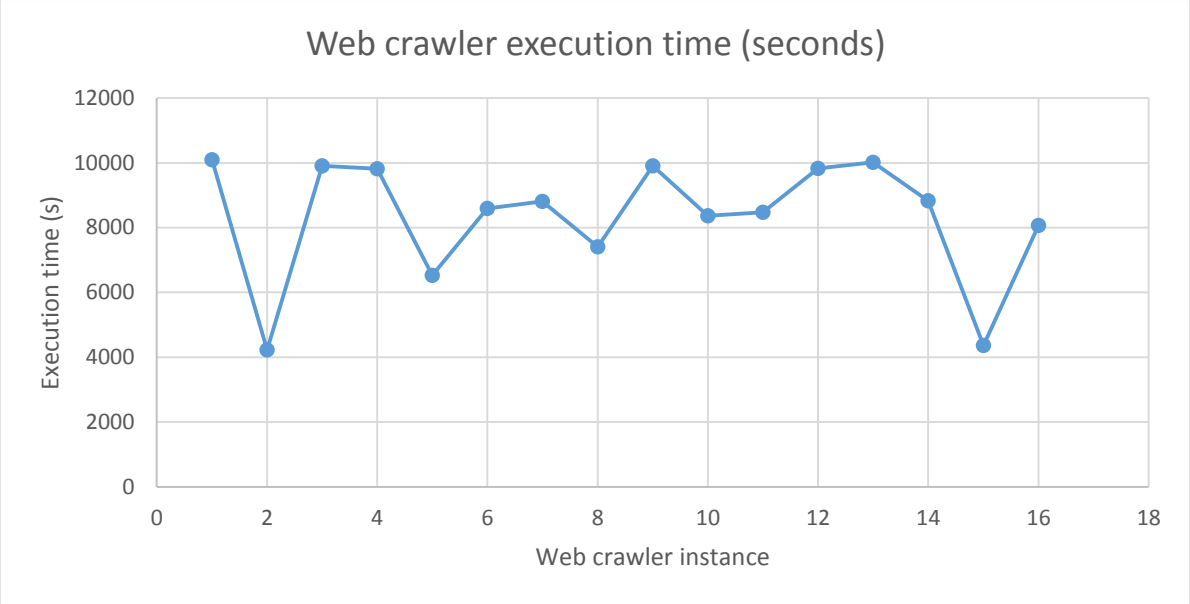

Figure 27: Web crawler execution time

The average execution time for all 16 web crawlers was 2 hours 18 minutes and 16 seconds. The shortest execution time for a web crawler was 1 hour 10 minutes and 28 seconds and the longest 2 hours 48 minutes and 13 seconds.

From *Figure 27* it can be seen that web crawlers 2 and 15 executed significantly faster than any of the others. When investigated it was noticed that the websites assigned to web crawlers 2 and 15 contained fewer links to other websites, and therefore less pages had to be scanned. This is shown in *Figure 28*.

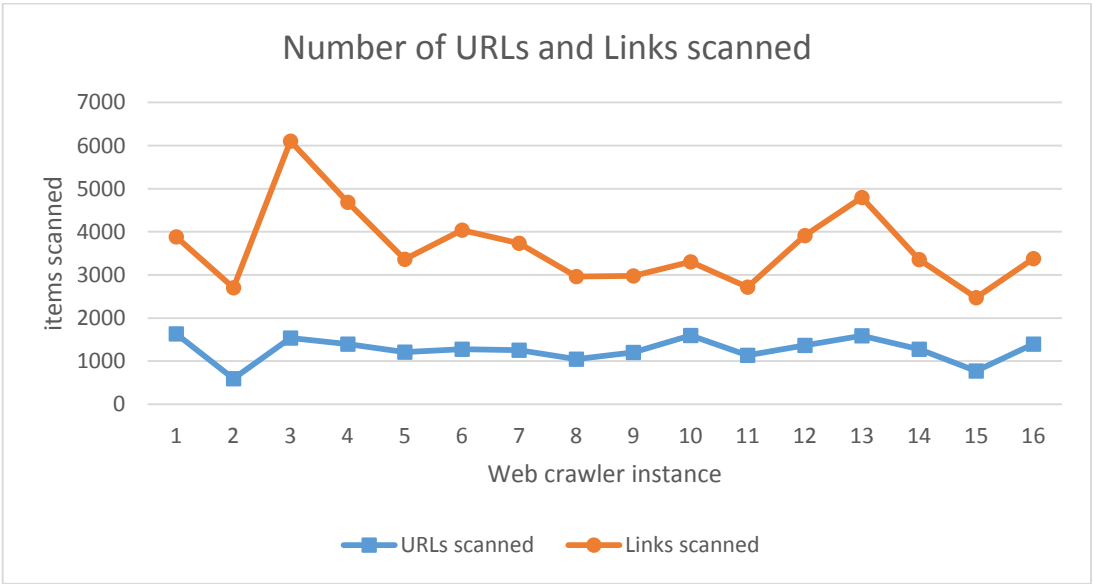

Figure 28: Web crawler URLs and Links scanned

It should be noted that for all the web crawlers the number of links that were scanned is significantly more than the number of URLs that were scanned. This happens because the dtSearch Engine implements a duplicate policy; once a link has been detected the dtSearch Engine will first proceed to verify that the page has not been scanned previously. If the page has not been scanned before the dtSearch Engine will load the URL and scan the associated page. If the link has been scanned before the dtSearch Engine will log that the link has been detected before proceeding to check the next one. Therefore the number of URLs scanned represent the number of web sites that the dtSearch Engine has downloaded and indexed, whereas the number of links scanned represent the number of links the dtSearch Engine has detected in total.

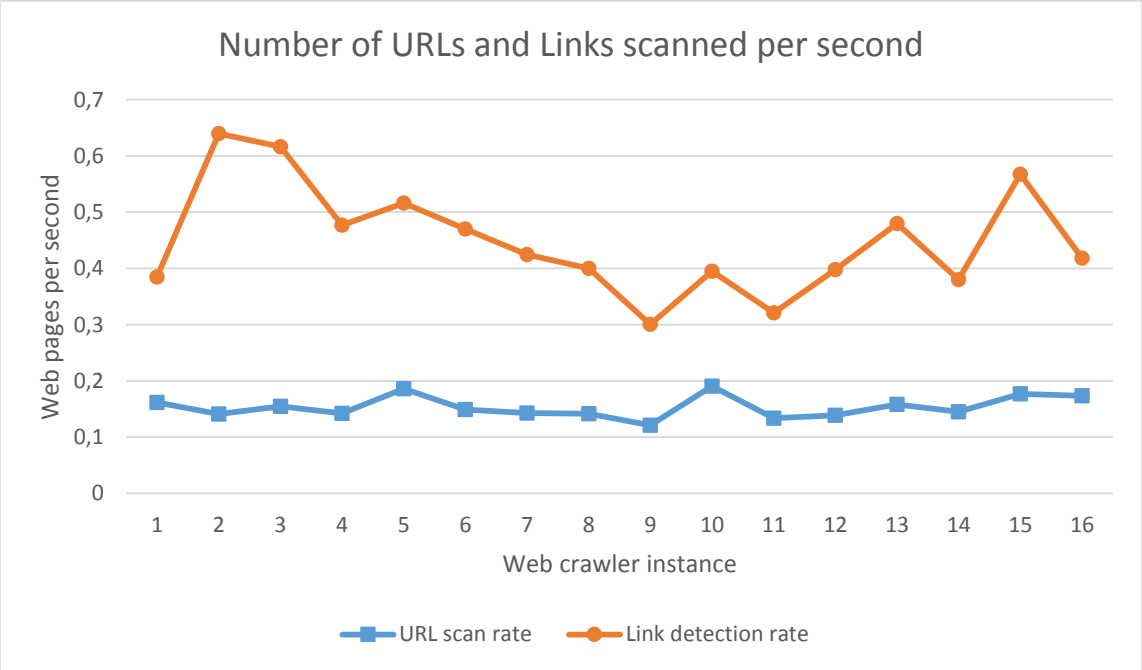

Figure 29: Web crawler URLs and Links scanned per second

As stated in Section 1.3 the web crawler must be able to scan the internet as fast as possible. Therefore the crawling rate of the web crawlers measured in web pages per second are very important. This is shown in *Figure 29*.

Figure 29 shows that no single web crawler instance was able to scan the web at a rate faster than 1 website per second. However, as each web crawler instance is downloading a separate website at a time

all the crawling rates can be added together in order to get an indication for how fast the web crawlers are scanning the internet. Once all the web pages were scanned the cumulative web crawling rate was calculated to be 2.45 webpages per second and the cumulative link detection rate to be 7.19 links per second. This will increase significantly should a faster internet connection be used.

5.2.2. Twitter Scanner

The Twitter Scanner was loaded with the 10 keywords as specified in Section 5.1 and executed for 3 hours and 40 minutes.

During this test 20 694 results were detected. From the total number of results 16 370 were non-English results and 1 311 were duplicates of previous results. As such 17 861 (85.44%) results were filtered out which left 3 013 valid results. The information is graphically shown in *Figure 30*.

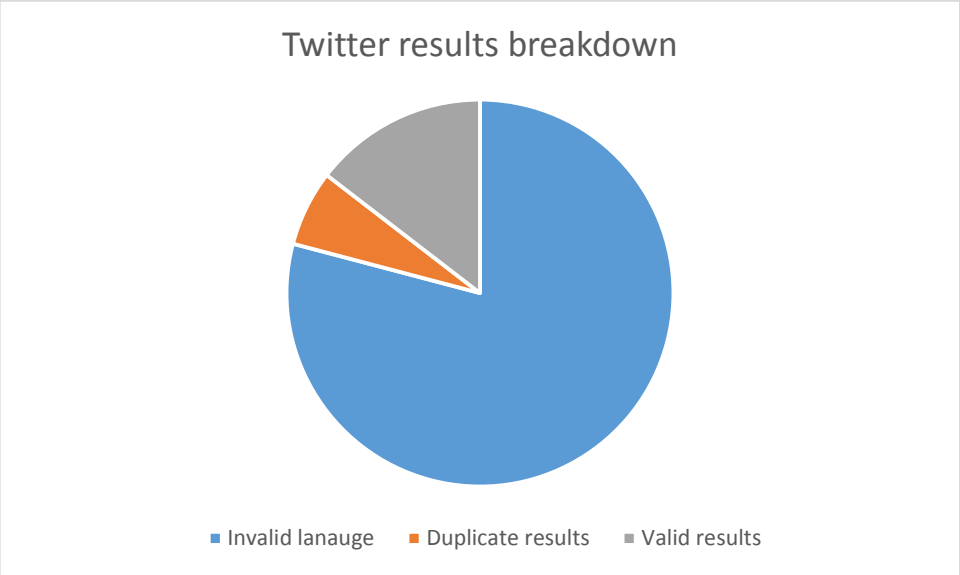

Figure 30: Twitter results breakdown

Analysing the results shows that valid tweets containing of the ten keywords were detected at a rate 0.22 valid results per second. While this may seem low it should be kept in mind this is the rate at which results that are of any use are presented by the Twitter public stream. To verify that no data was lost two separate programs were connected to the Twitter public stream and both received the same amount of results. If all results are taken into account the result detection rate is 1.57 results per second. Unlike

the web crawlers a faster internet connection will not have any additional benefit; the only way to detect more results would be to add more keywords to the Twitter filter.

5.2.3. Facebook Scanner

The Facebook Scanner was provided with the 10 keywords as specified in Section 5.1 and executed for 3 hours and 40 minutes.

During this test 27 991 results were detected. From the total number of results 23 416 were duplicates of previous results and 4 107 were results that did not contain significant meaning or was written in a different language. As such 27 523 (98.32%) results were filtered out which left 468 valid results. The information is graphically shown in *Figure 31*.

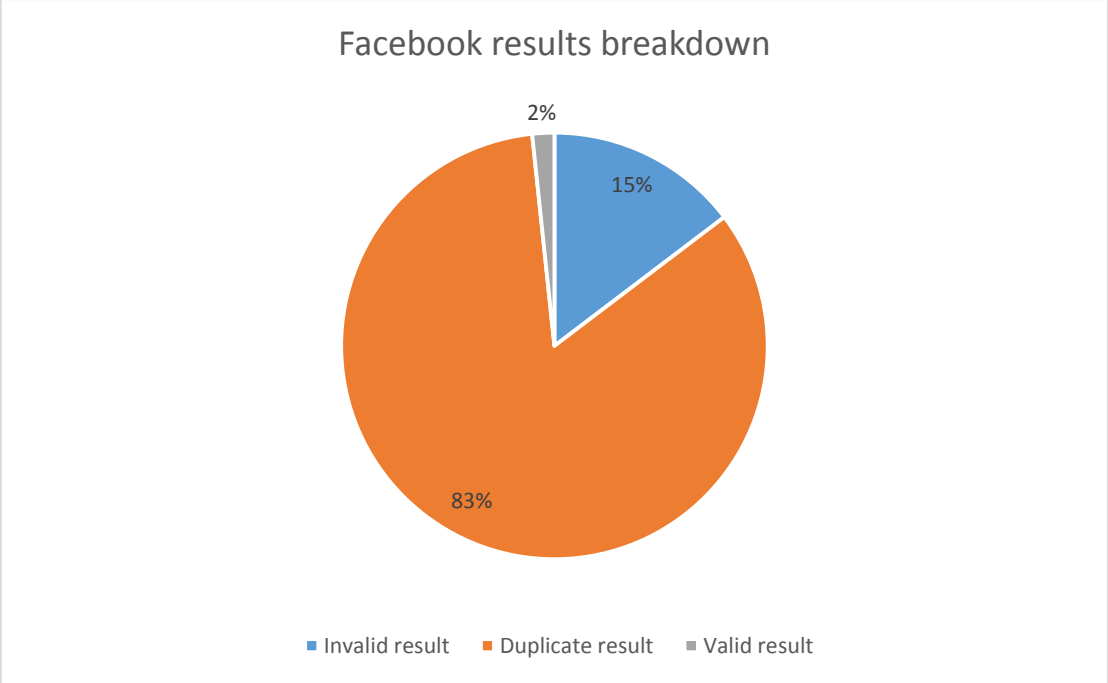

Figure 31: Facebook results breakdown

Analysing the results show that valid Facebook posts that contained one of the ten keywords were detected at a rate of 0.035 valid results per second. While this may seem low it should be kept in mind this is the rate at which results that are of any use were detected. Unlike Twitter it was not needed to connect two separate programs to the Facebook public data as presence of duplicate entries presented

by the same user proves that no data has been lost. If all results are taken into account the result detection rate is 2.12 results per second.

It should be noted that although more results are retrieved from Facebook than Twitter, Twitter has more valid results. The reason for this lies in the scanning methods of Facebook and Twitter. To extract data form Twitter a continuous stream is opened that automatically presents new results whereas Facebook does not support this functionality and has to be periodically scanned. By scanning Facebook a fixed number of results are always returned. However, results are generated very slowly on Facebook, as demonstrated in Section 4.1.3 and therefore a large number of duplicate results will be retrieved.

5.3. The Front-End

The data generated by the Back-End could be accessed by the Front-End once all its tests were complete.

5.3.1. Web crawler result generator

When provided with the index files generated by the web crawler it took the result generator 2.53 seconds to generate results for the 10 keywords with a similarity threshold of 20%, as discussed in Section 4.2.2. The number of results for each keyword is shown in *Table 13*.

Table 13: Amount of web crawler results

Keyword	Amount of results
Audi	79
CNA	2
Edgars	22
Jeep	47
Musica	4
Shoprite	42
Spar	45
Spur	53
Wimpy	10
Woolworths	104
Total	408

A screenshot of the web crawler results screen is shown in *Figure 32*. From the screenshot it can be seen that all web crawler results contain the following information:

- Title, location, page and paragraph of the web page where the result was located,
- The data along with the keyword that was detected,
- An internal score as calculated by dtSearch that indicates how strongly the result matches the keyword along with a relevancy score which will filter out results which are deemed too similar to its keyword,
- The number of times the specific result was found as well as the number of times the keyword features within the text (Hit Count).
- The calculated sentiment of the given text.

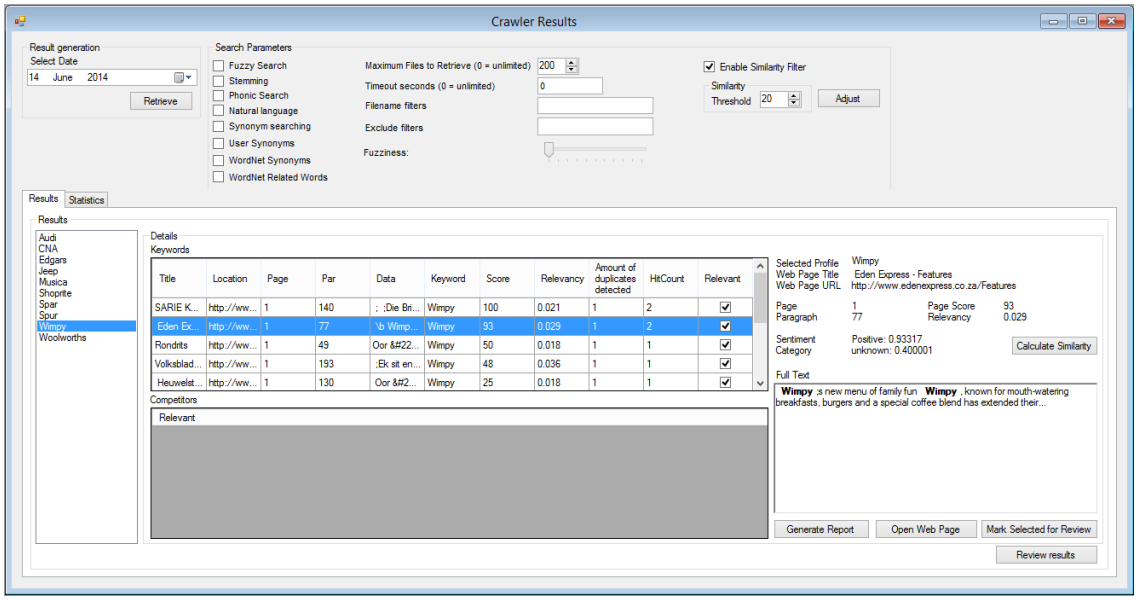

Figure 32: Screenshot of web crawler results

When entering a keyword into the system, the software will allow the system administrator to specify a competitor a keyword. This will be saved to the database as additional keywords and will be included in the scanning processes by the Back-End, but when analysing the information the system will show the information for the current keyword as well the information for its competitors. This will allow the system administrator to compare the web crawling results for the keyword against those of its competitor. After the results have been analysed by passing the results through the similarity filter and

sentiment analysis tool, the analysed results will allow any potential client to see how their company compares against its competitors.

It may be noticed that the “Competitors” grid at the bottom of the screen is currently empty. For this scenario no competitors were entered and as such no information can be displayed. If the user were to give the current profile a competitor, for example “Woolworths”, its results will be shown in the grid.

This demonstrates the effectiveness of scanning the internet and generating results at a later date. By using the index files all the data is already present and the user can request results for new keywords at any time without rescanning the web.

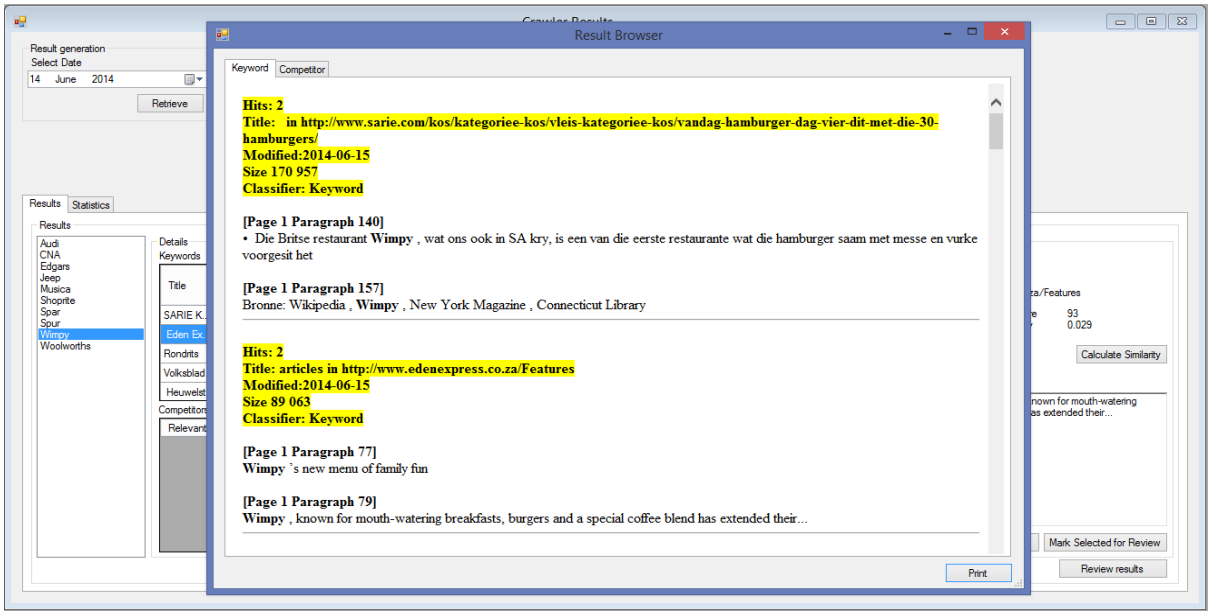

Figure 33: Results report of web crawler

The user has the ability to generate a report of all the results that were detected. The report will give a summarized version of all keyword mentions as well as the website, page and paragraph where the keyword was found. The report will exclude information such as calculated sentiments as sentiments are only calculated once a user has marked a result as relevant.

The user will also have the ability to re-generate the web page of any detected result in order to provide some information regarding the context in which the keyword was found. Examples of the results report and regenerated web page are shown in *Figure 33* and *Figure 34.*

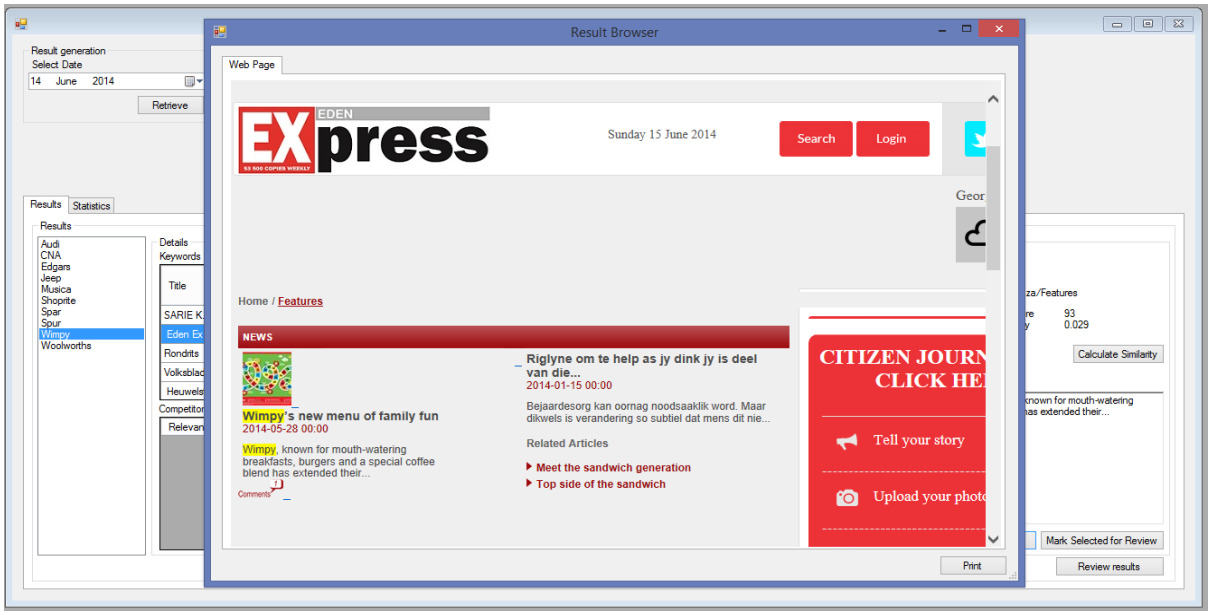

Figure 34: Regenerated web page of web crawler result

As explained in Section 3.1.2 the web crawler cannot detect the context in which a keyword features and therefore not all of the results that are provided will be relevant. As such the user must filter through the results provided by the web crawler and must manually determine which results are relevant, as demonstrated by the results column in *Figure 32*. Once the user has marked a result as relevant the system will calculate the result's sentiment and category by using AlchemyAPI before adding the result to a list that contains all the results that the user has marked as relevant. Once the user has finished marking the relevant results he or she may proceed to review the results. This is shown in *Figure 35*.

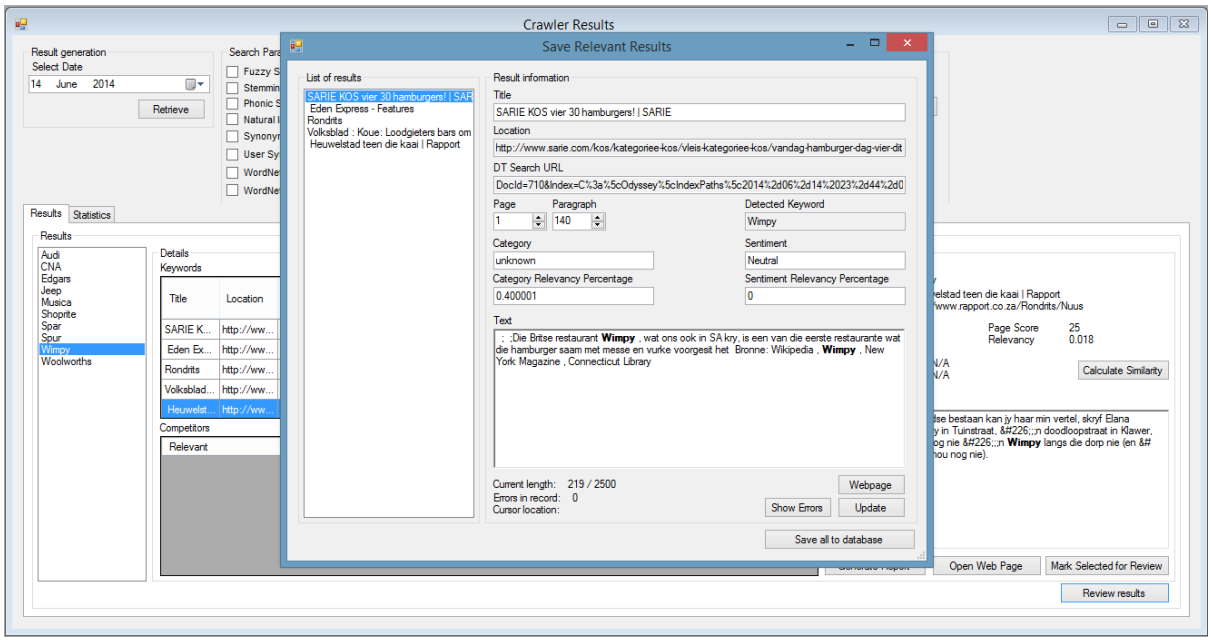

Figure 35: Web crawler results review

When reviewing the results the user have the ability to alter the title, category, sentiment and text of the result in order to remove any unnecessary characters which may have been added when the result was acquired from the web. After reviewing the results the user will be able to save the results to the database from where it will be used to generate statistics.

5.3.2. Twitter scanner results

When provided with the results from the Twitter scanner it took the Front-End 0.53 seconds to process the 3 013 results that were retrieved by the Back-End. Like the web crawler results a similarity threshold of 20% was used, which filtered out 284 results. A screenshot of the Twitter results processor for "Edgars" is shown in *Figure 36.*

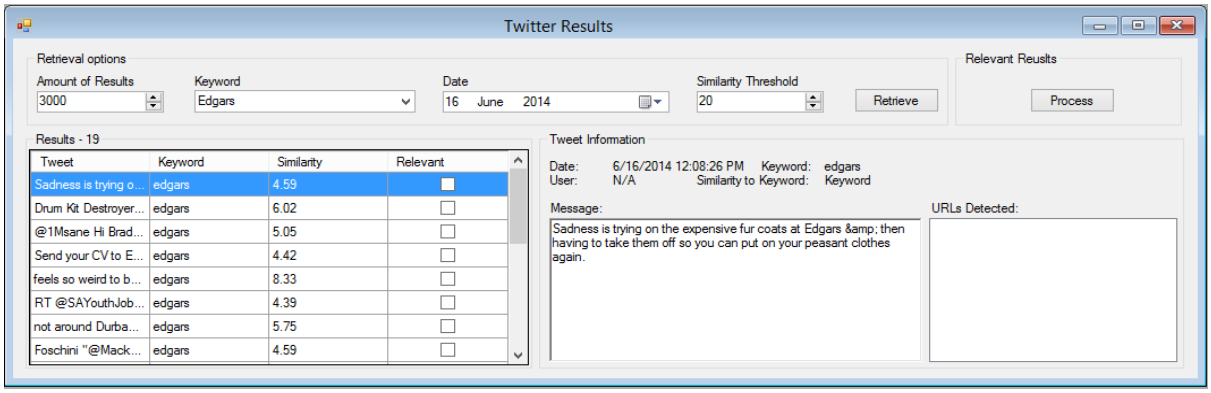

Figure 36: Twitter results processor

From the screen shown in *Figure 36* the user can select which results are relevant and mark them to be saved. Once the saving procedures starts the sentiment of each marked result will be calculated by using AlchemyAPI and it will be saved to the database to be used when generating statistics.

5.3.3. Facebook scanner results

When provided with the results from the Facebook scanner it took the Front-End 0.83 seconds to process the 468 results that were retrieved by the Back-End. This may seem strange as 3 013 Twitter results took only 0.53 seconds to scan, but it should be noted that a single Twitter message can only be 150 characters in length, while a Facebook message contain up to 7 700 characters. Like the web crawler and Twitter scanner results a similarity threshold of 20% was used, which filtered out 123 results. A screenshot of the Facebook results processor is shown in *Figure 37.*

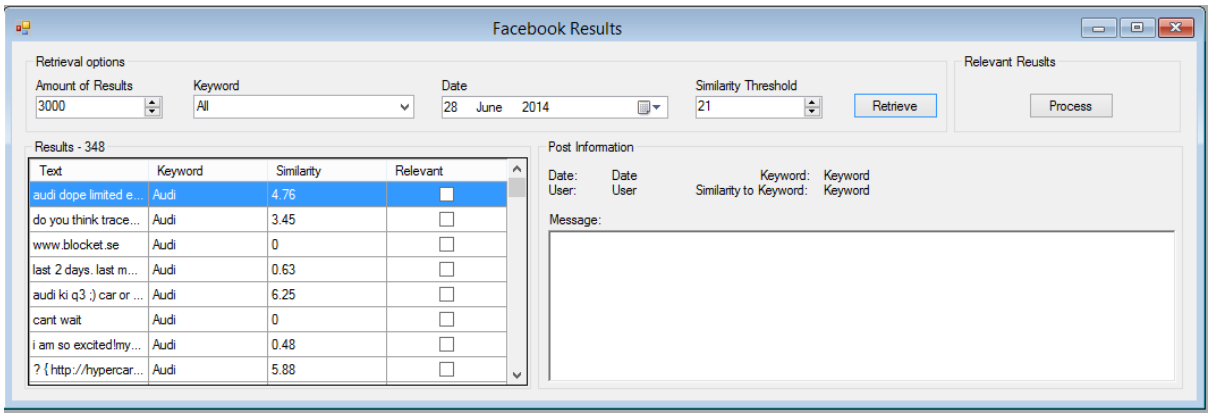

Figure 37: Facebook results processor

During a deeper analysis, it was noticed that the searching techniques used by the Facebook Graph Search compensates for synonyms and spelling mistakes and as such the user may not always receive the correct keyword. For example: searching for “Edgars” yielded results for both the clothing outlet as well as various people named “Edgar”. As such additional filters must be implemented in order to ensure that the keyword the user searchers for does indeed feature within the message returned from Facebook. After implementing such a filter 161 results were returned from the original web and social network crawlers.

From the screen shown in *Figure 37* the user can select which results are relevant and mark them to be saved. Once the saving procedures starts the sentiment of each marked result will be calculated by using AlchemyAPI and it will be saved to the database to be used when generating statistics.

5.3.4. Online reputation calculation

Once the user has calculated the sentiment of the web crawler, Twitter and Facebook results the profile overview can finally be calculated. This will be done by generating the averages of the calculated

sentiments for each of the categories (web crawler, Twitter Scanner and Facebook). If the average sentiment for the keyword is larger than 0 then result will have a positive perception, and vice versa.

It will take too long to determine the online reputation of all the brands that have been listed at the start of the chapter as the free account of AlchemyAPI only allows the user to process 1000 sentiments during a day. Therefore the online reputation of only one company will be determined in this section. It was decided to determine the online reputation of Edgars as the web and social networking crawlers have produced enough results to determine its online reputation but the number of results will not surpass the limitations of the AlchemyAPI free account.

Figure 38 shows a screenshot of the final result for the scenario used in this chapter. It can be seen that Edgars currently has a positive perception on both the internet, Twitter and Facebook. When manually verifying the online sentiments it was found that 82.85% of the sentiments were correctly identified, which matches the figures presented in Section 3.2.4.

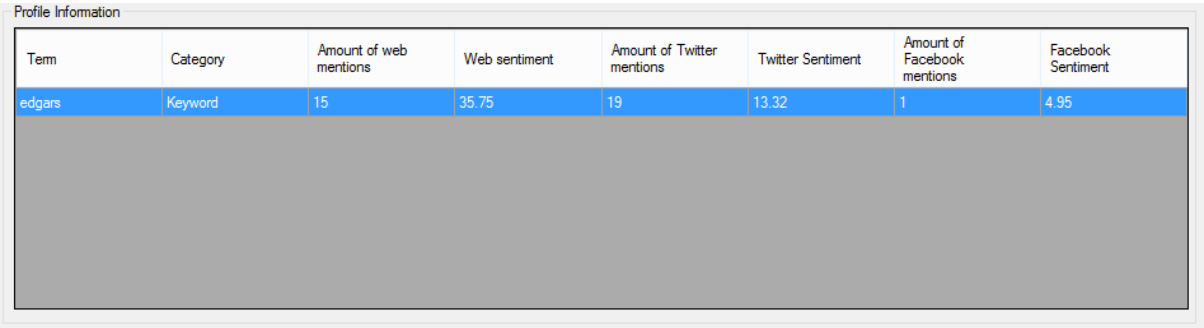

Figure 38: Profile overview

5.4. The Website

The Website will be able to show the processed results to any interested party without allowing them access to the database. In order to be useful the Website must allow the user to:

- View the processed results of any keyword, as well as indicate the origin of the keyword.
- Indicate whether the specified brand has a positive or negative online reputation.
- Allow the user to compare the keyword against similar brands.

All of these results can easily be retrieved from the database by using the processed results. Screenshots of the webpages are shown in *Figure 39* - *Figure 42*.

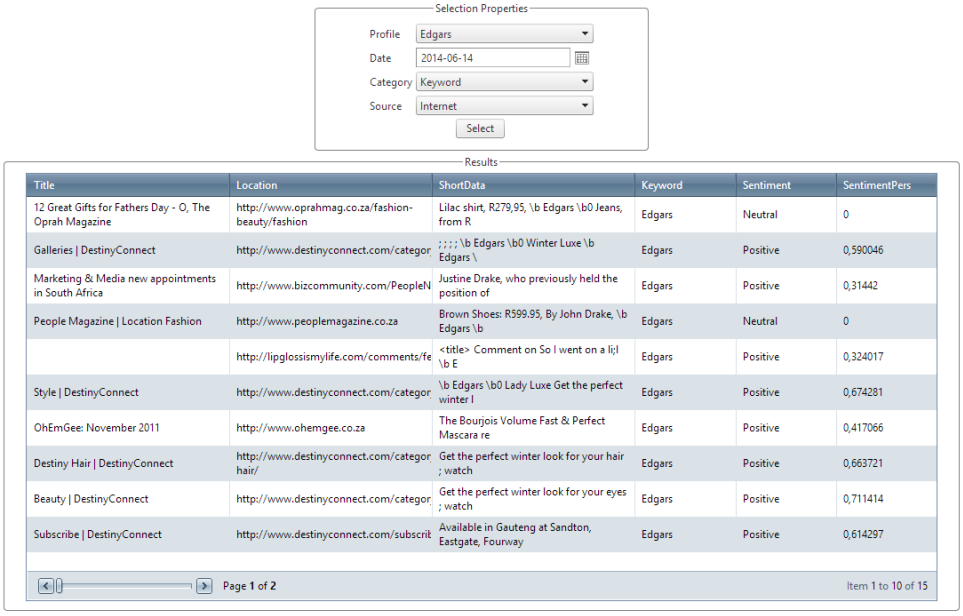

Figure 39: Website results – Internet

Figure 39 shows the results from the web crawlers after they have been processed. In the screenshot the user can clearly see the title, location, a summarized version of the result, the keyword as well as the sentiment of each result.

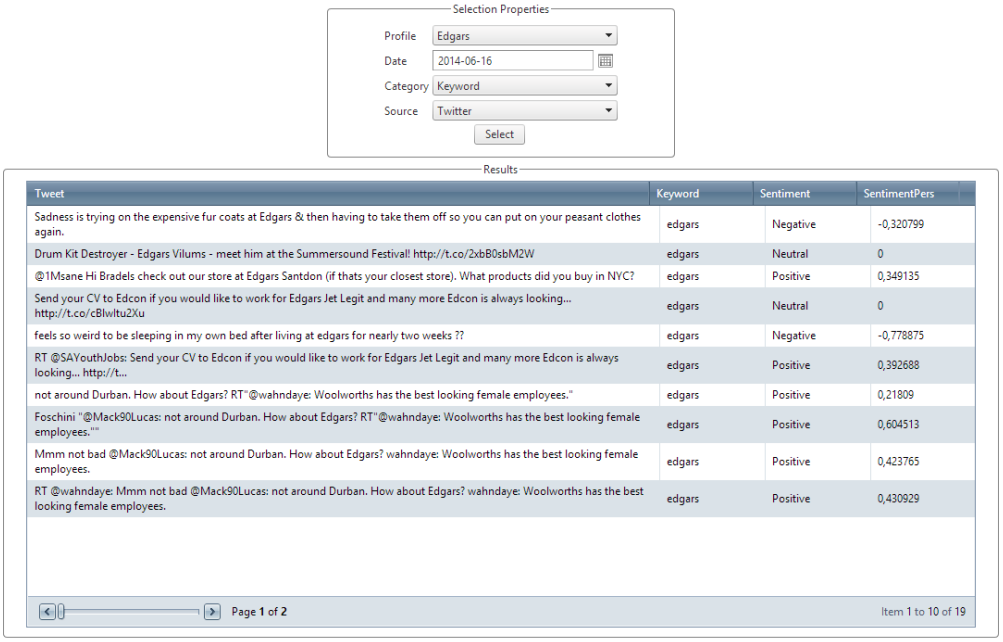

Figure 40: Website results – Twitter

Figure 40 shows the results that have been retrieved from Twitter after they have been processed. In the screenshot the user can clearly see the full Twitter message as well as the keyword that was detected and the calculated sentiment.

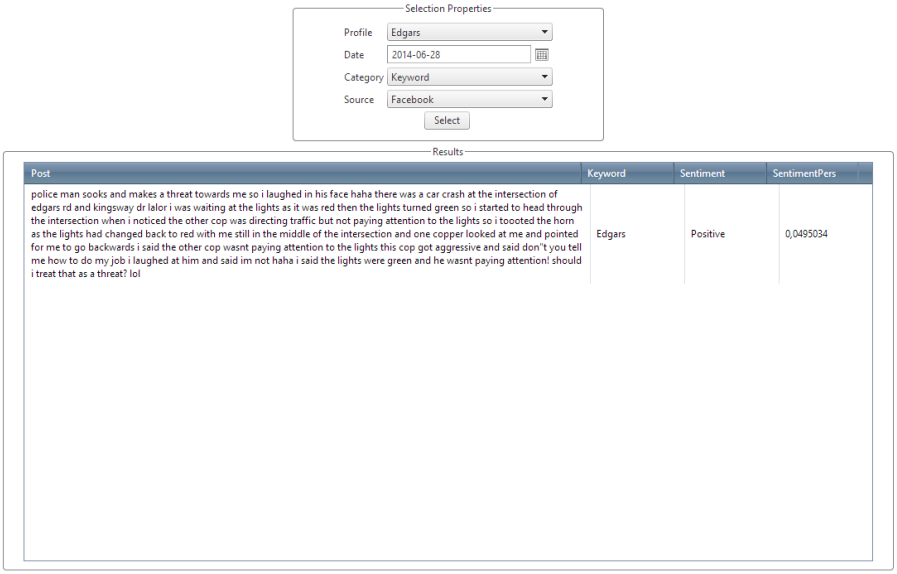

Figure 41: Website results – Facebook

Figure 41 shows the results that have been retrieved from Facebook after they have been processed. In the screenshot the user can clearly see the full Facebook message as well as the keyword that was detected and the calculated sentiment.

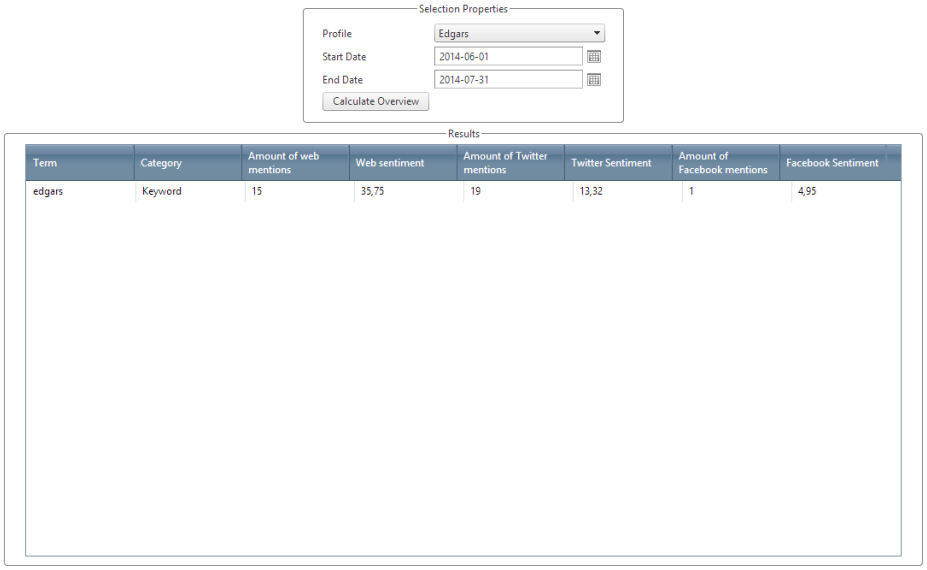

Figure 42: Website - Overall sentiment

Figure 42 shows the final screen that gives the overall reputation of the selected keyword, in this case "Edgars". From the screenshot it can be seen sentiments for the keywords are clearly indicated and are the same as the results provided by the Front-End as seen in *Figure 38*.

Chapter 6

Conclusion and Recommendations

This chapter will provide an overview of the goals of the research before stating whether the goals have been met and whether the research was a success or not. A final thought will be given to any enhancements that may be researched in the future.

In order to determine whether the project was a success its accomplishments must be evaluated against the original goals. As stated in Section 1.3, the goals of this project were to investigate existing online reputation monitoring systems in order to develop a new system that will:

- scan web pages at a sufficient rate to ensure all the results are continually kept up to date;
- scan social networking sites for public opinions;
- analysing results in real time to give the user information such as the location, page, paragraph and sentiment of the results;
- using the analysed results to generate reports;
- present the user with as much control over the entire process as possible.

By using the knowledge gained from investigating BrandsEye and Brand.Com, an online monitoring system was developed that is capable of:

- scanning web pages at a rate of 2.45 web pages per second by using a 1Mb/s internet connection;
- using the information from the web crawlers to detect 200 results for each of 10 random keywords in 2.53 seconds;
- using the web crawler results to generate various reports as well as the original website where the results were located;
- acquiring results from Twitter and Facebook at a rate of 0.22 and 0.035 valid results per second by using 10 random keywords;
- filtering and processing data retrieved in 3 hours 40 minutes from Twitter and Facebook results in 0.53 and 0.85 seconds respectively;
- generating the sentiments of the web crawler as well as the Twitter and Facebook results;
- generating a complete overview of the keyword to indicate how the keyword is seen by the online community.

As can be seen from the results all the original goals were met and therefore the project was a success but several areas could be improved upon. Currently the main stumbling block is the use of AlchemyAPI in order to calculate the sentiments of the results. In order to use AlchemyAPI the user must pay a montly subscription fee which will become expensive. Therefore sentiment analysis tools can be investigated in the future to see whether it will be possible to develop a custom sentiment analysis tool that will minimize if not get rid of all the subscription fee costs.

The dtSearch Engine is another possible area of investigation; while the dtSearch Engine only has to be purchased one time it does not allow the user a lot of flexibility. While this is not a problem at the present time additional functions might be added in the future that the dtSearch Engine may either

prohibit or cannot perform. It would be ideal if the system could use a custom web crawler specifically made for an ORM system over which he or she has full control.

Lastly a thought must be given towards the social networking sites that are currently scanned. At present Twitter is generating sufficient data for an ORM service but due to its internal search processes Facebook may not always provide the user with sufficient information, as indicated by the "Edgars" example in Section 5.3.3. Additionally Facebook has upgraded its Graph API in April 2014 which has deprecated the post scanning functionality the current ORM system makes use of. Currently Facebook still provides the post scanning functionality by means of the previous API version but it is unknown how long this API will remain supported. As such either a new method of interfacing with the Facebook network must be investigated or the system must drop support for Facebook and instead make use of another system such as Google Plus.

Appendix A
Index files

This add-on chapter will give a quick overview of how index files are generated, used and what their advantages are.

Index files contain all the words that were detected within the web pages and their respective locations. The example below demonstrates this principle; it should be noted that this may not be the exact algorithm the dtSearch Engine uses and the example is intended for demonstration purposes only.

Original Text

The end game to South Africa's big platinum strike is drawing near after the producers said they would take their latest wage offer directly to employees after marathon wage talks to end the 13-week strike collapsed on Thursday.

Source: "Platinum firms take offer to workers" by Reuters on 28/4/2014, published on News24.Com. Link: http://www.fin24.com/Companies/Mining/Platinum-firms-take-offer-to-workers-20140428

Index File:

Word	Location
13-week	34
Africa's	6
After	13,27
Big	7
Collapsed	36
Directly	24
Drawing	11
Employees	26
End	2,32
Game	3
Is	10
Latest	21
Marathon	28
Near	12
Offer	23
On	37
Platinum	8
Producers	15
Said	16
South	5,25,31

Strike	9,35
Take	19
Talks	30
The	1,14,33
Their	20
They	17
Thursday	38
To	4
Wage	22,29
Would	18

Example 1: Index file demonstration

By the table in *Example 1* the location of all mentions of a specific word can be quickly retrieved, allowing the user or a software application to quickly scan a document. However, each index file implementation uses its own algorithm hence making it incapable for other indexing applications to read each other's files. The dtSearch Engine is no different; when the dtSearch Engine indexes web page information it creates the index files on a designated storage device. It would be ideal if this data could be saved to a local database instead of a hard-drive, but due to the unique dtSearch algorithm it would be impossible to read the data from the index file with a custom application and the files are too large to consider saving the entire file to the database.

Appendix B

Conference Presentations

This add-on chapter will indicate where this research has been presented and provide any articles that have been written.

Title: Development of an online reputation monitor

Format: Poster presentation

Presented at:

Southern African Telecommunications Networks and Applications Conference (SATNAC)

Stellenbosch, Western Cape, South Africa

2 – 4 September 2014

Title: Development of an online reputation monitor by using existing components

Format: Full-paper

Presented at:

Southern African Telecommunications Networks and Applications Conference (SATNAC)

Port Elizabeth, Eastern Cape, South Africa

1 - 3 September 2014

Development of an online reputation monitor

G.J.C. Venter, W.C. Venter, A.J. Hoffman
School of Computer and Electronic Engineering,
North-West University, Potchefstroom Campus,
Private Bag x4001, x442, Potchefstroom 2520
Tel: +27 76 358 8241, Fax: +27 18 299 1977
Email: mgmGertV@gmail.com, willie.venter@nwu.ac.za, alwyn.hoffman@nwu.ac.za

Abstract- Customer opinion about companies are very important and companies often get customer feedback via surveys or other official methods. Some customers prefer to voice their opinion on the internet where they take comfort in anonymity. Currently this form of customer feedback is not closely monitored. This project aims to address this shortcoming by developing a system capable of monitoring various web and social networking sites for customer feedback.

Index Terms— Online Reputation Monitor, Web crawler, Facebook, Twitter, dtSearch.

I. INTRODUCTION

Advertising is a tool that is used to establish a basic awareness of a product or service in a potential customer by providing selected information [1]. Research shows that advertising has a major influence on customer preferences and can help consumers to make decisions [1][2]. However, according to Reichheld [3] the tremendous cost of marketing makes it hard for a company to grow profitable. Reichheld [3] believes that the only path to a profitable growth rate lies in the company's ability to get loyal customers to promote a company by sharing information about the company's products and the customer's experience [4].

Getting customers to talk about a company is not enough as customers may either promote or advise against the company. As such the company needs to monitor its public reputation. Most companies know this and often employ techniques such as focus groups and surveys to generate various statistics but these methods are not always effective. People often feel under pressure when their opinions are personally asked and therefore adjust their answers to avoid any potential confrontation. Instead many customers prefer to voice their opinion on the internet where they take comfort in anonymity. Therefore companies need to determine their online reputation.

Determining online reputation is not a new field and has been done for years by organizations such as BrandsEye and Brand.Com. However, the services these companies offer have certain limitations. The services normally require a monthly fee and companies are not allowed to purchase the online reputation monitoring (ORM) software. The user also has very little control over the software itself: such services search for user specified keywords over a range of websites, yet the user cannot specify which websites to monitor or change the keywords without restarting the service. Additionally the services rarely make use of historical data, meaning a company's reputation can only be analyzed from the present day onwards.

To solve this problem a new ORM system is proposed. The system will allow the user to scan a number of web pages and social networking sites such as Twitter and Facebook at a sufficient rate to ensure that all results are continually kept up to date. Once results from the web and social networking sites have been gathered the user will be presented with various methods capable of analyzing the results. The user will be capable of retrieving information regarding the result's source such as the website, page and paragraph, the date at which the result was generated as well as the sentiment and the category of the result. Once all the results have been analyzed the user will have the option to generate various reports that will indicate how his or her company is seen by the online community.

II. BACKGROUND

The basic functionality of ORM involves scanning a variety of web pages and social networking sites, analyzing the results to determine their relevancy, using the analyzed results to generate various statistics and visualizing the results. Clearly an ORM system isn't a single process, but a collaboration of several processes that includes a web and social networking crawler, a method of analyzing the results and a sentiment analysis tool.

A. Web crawler

Web crawlers are programs that explore the World Wide Web, retrieve information and store the results for future use [5]. This process is known as *web crawling*. The type of data that is extracted from web pages depend on the implementation of the web crawler. Some web crawlers are configured to extract only specified phrases [6], while others extract and index each word in a web page for future use [7].

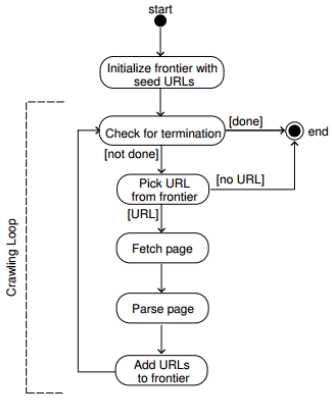

Figure 43: Basic Crawler Architecture

Figure 3 shows the architecture of a basic web crawler [5]. Before a web crawler is initialized a user must specify a list of seed URLs which is stored in the *frontier*, a list of unvisited URLs. When the web crawler starts, it will load the first URL in the frontier, download and scan the contents and store any results in a database or on a local storage device. Once the crawler has finished scanning the page it will load the next URL from the frontier and repeat the process. This is known as the *crawling loop*.

Many web sites contain multiple pages, which in turn contains additional subpages. This is illustrated in *Figure 4*.

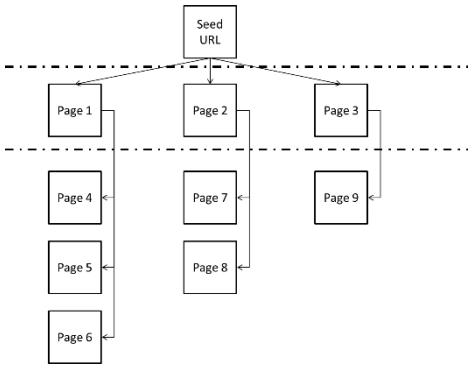

Figure 44: Crawl Depth Illustration

Web crawlers are often configured to scan only a certain *depth*. This is the extent to which a web crawler scans the page and is specified when the user adds the seed URL. If the user wish to scan only the original page the crawl depth will be set to 0. If the user sets the crawl depth to 1, the web crawler will scan the seed URL and add the URLs for *Page 1*, *Page 2* and *Page3* to the top of the frontier. Once the web crawler has finished scanning the seed URL, it will proceed to download and scan *Page 1*, followed by *Page 2* and finally *Page 3*. The same process will be followed for *Page 4-9* if the user sets the crawl depth to 2.

Web crawler execution ends when all the links in the frontier have been scanned. At this stage timers are usually initialized which will redeploy the crawler at a specific time or after a specified interval.

B. Social Networking Crawlers

Unlike regular web sites, web crawlers cannot scan social networking sites. Social networking sites only present data once the user has registered and even if logged in the site will only show data regarding the people the current user is connected to. Normal web crawlers do not possess authentication capacities and therefore cannot access social networking sites and those that do will only have access to limited info.

Therefore various social networking sites have designed an interface that allows registered applications to access the site's public data. This interface is known as an Application Programming Interface (API).

All social networking sites have different API's and therefore different methods of extracting information. As such it will not be possible to design a universal method of interfacing with all social networking sites. The project will only be able to interface with some social networking sites due to time constraints. It was decided to interface with Facebook and Twitter as they had the highest Alexa rank at the time, which is an indicator of network traffic.

C. Sentiment analysis tools

Once the web or social networking crawler returns a result it must first be analyzed to determine whether the result contains any relevant data before it can be presented to the user. However, the user often wishes to know whether the sentiment of the result is positive or negative.

There are two main methods used to calculate the sentiment of some text, namely the lexical approach and the machine learning approach.

A system that is based on the lexical approach uses a dictionary of pre-tagged words. Each word in the text that has to be analyzed is compared against the dictionary to find its polarity. This indicates whether the word has a positive or negative sentiment as well as the strength of the sentiment. After the polarity of each word has been determined, the polarity of the given text is calculated by summing the polarity for each word. According to Annett [8], the accuracy of such a system varies between 64% and 82%, depending on orientation of statistic metrics and the dictionary that was used.

A system that uses the machine learning approach uses two components; a series of feature vectors and a collection of tagged corpora which is a collection of documents that the system uses to train itself [9]. Feature vectors are usually a variety of *uni-grams*, single words from within a document, or *n-grams*, two or more words from a document that are in sequential order. Other features that are often proposed include the number of positive words, number of negative words, and the length of a document. Both the feature vectors and collection of tagged corpora are used to train a classifier, which can be applied to an untagged document to determine its sentiment. According to Annett [8], the accuracy of such a system varies between 63% and 82%, but the results are dependent of the features that were selected.

III. DESIGN

As stated in the introduction, existing ORM system have various technical limitations that limit their usefulness and therefore it was decided to implement a new ORM system. To include all the functionality of ORM as discussed in Section II it was decided to develop a system consisting of three parts, namely a Back-End, a Front-End and a Website.

A. The Back-End

To begin the ORM process information must be retrieved from the internet. This will be done by the Back-End, which will initialize and maintain all the web and social networking crawlers.

Before the ORM process can be started the user must specify a list of seed URLs, their respective crawl depths as well as a series of keywords. The URLs and the crawl depths will be used by the web crawler whereas the social networking crawler will use the keywords to find any results from Facebook and Twitter.

Information gathered by the web and social networking crawlers will be saved either to a database or a local storage device. The storage location will depend on the type of web and social networking crawler that will be used. Once the Back-End has finished crawling the web and social networking sites it will proceed to wait for a predetermined amount of time before restarting the process.

The Back-End will be designed to function with as little human interaction as possible. The only interaction that may be required would be to verify that the Back-End has not encountered an error or to turn off the system in times of maintenance.

B. *The Front-End*

After the data has been acquired by the Back-End the results need to be processed. This is the goal of the Front-End. The Front-End will proceed to load the stored results for a specified date range and process the data by identifying the paragraphs in which the keywords feature before passing the paragraphs through a similarity filter and a sentiment analysis tool. Afterwards the user will be able to verify the results before saving them to a database.

The similarity filter will filter out results that are too close to the keywords, such as tags within a web page. This will lessen the amount of results that will have to be processed by the sentiment analysis filter. The sentiment analysis tool will use the results that have passed through the similarity filter and determine whether the results are positive or negative. These results will be used to determine the overall opinion of the company or brand that is being investigated. Once the sentiment of the results have been calculated they will be saved back in the database.

The Front-End will be designed to allow multiple instances of the software to run in parallel. This will allow multiple system administrators to access the system at the same time which will increase the amount of results that can be processed at once. While it will be possible to fully automate the Front-End, it would be wise to allow some degree of human interaction to verify if the results that pass through the filters are correct. The main reason for this is the fact that the meaning of words can differ depending on the context which it is used in. Using the keyword "Kalahari" to monitor the public opinion of the online marketplace "Kalahari.com" may yield results for both the online marketplace and the Kalahari Desert, which will influence the results and statistics.

C. *The Website*

Once the results have been processed the system administrator will require a method to display the processed results to the user or any other interested party. This can easily be accomplished via a web site as it will allow the data to be accessed anywhere.

IV. COMPONENT SELECTION

The implementation of the ORM system will make use of existing components - there is no use in redesigning the wheel. Available components will be evaluated and component selection will be based on the results of the evaluation.

A. *Web crawler*

Ideally the ORM system would make use of a powerful web crawler such as *GoogleBot* or *BingBot*, respectively developed by Google and Microsoft. Unfortunately these crawlers are not available for public use. An investigation discovered many free and commercially available web crawlers available on the internet. A commercial text retrieval engine as well as several open source web crawlers were acquired and evaluated.

The dtSearch Engine is a commercial text retrieval engine with many features that provides the user with a web crawler that is actively maintained. The user has no control over the internal functionality of the dtSearch Engine as it is not open-source but the dtSearch Engine can be manipulated by using the settings that are presented. The dtSearch Engine creates index files when scanning the internet which can be accessed by a result generator included with the web crawler.

The HTML Agility Pack (HAP) is a free library that allows the user to parse web pages directly from the web. The HAP does not have any internal web crawling capabilities but provides the user with components to build a custom web crawler and result generator. The HAP is open source; as such the HAP is completely cost-free but support for the library may stop if the community stops working on the project. A second open-source web crawler named *Abot* was evaluated. The *Abot* crawler contains pre-programmed functionality that is based on the HAP while still presenting the user with as much functionality as possible.

Various aspects of the crawlers were compared using a weighted average to determine the web crawler best suited for this project. The aspects considered were the speed with which the web crawler is able to scan the web, the time it will take to incorporate the web crawler into the ORM system, additional features and the customizability of the web crawler as well as its cost.

The results are shown in *Table 14*. For a full discussion of the weight assigned to the different aspects as well as the scores of the different web crawlers for each aspect please refer to the dissertation "Development of an Online Reputation Monitor" by G.J.C. Venter at the North-West University.

Table 14: Web crawler comparison

Category	dtSearch Engine	HAP	Abot
Speed (30%)	26	27	19
Implementation time (30%)	30	6	12
Existing Features (15%)	15	2	7
Customizability (15%)	7	15	12
Cost (10%)	0	10	10
Total (100%)	**78**	**60**	**60**

From *Table 14* it can be seen that despite its cost the dtSearch Engine will be best suited for the ORM service. While the HAP and the Abot web crawlers may be almost as fast and free, they lack in the "implementation time" and "existing features" categories.

B. Social networking crawlers

As already explained, to extract sufficient information from social networking sites the ORM system will have to make use of the social network's API. The API provided by Twitter enables the user to extract data in one of two ways: by performing a search query via the REST API or getting a continuous feed of data from the Streaming API. This project will make use of the Streaming API because querying the RESP API is slow and creates significant internet overhead.

Two free Twitter components are available to allow the user to access the Twitter Streaming API, namely TweetInvi and Linq2Twitter. The differences between the components are marginal; both return only results which match a specific number of conditions.

Various aspects of the components are compared using a weighted average to determine the component best suited for the project. The aspects considered were the amount of results, the time it will take to implement the component within the ORM system, additional features and customizability.

The results of the components are shown in *Table 15*. For a full discussion of the weight assigned to the different aspects as well as the scores of the different components for each aspect please refer to the dissertation "Development of an Online Reputation Monitor" by G.J.C. Venter at the North-West University.

Table 15: Twitter API component comparison

Category	TweetInvi	Linq2Twitter
Amount of results (30%)	27	30
Implementation time (30%)	30	10
Existing Features (20%)	7	20
Customizability (20%)	10	20
Total (100%)	**74**	**80**

From *Table 15* it can be seen that it will be faster to implement the TweetInvi component but the limited information of the results makes it less viable than Linq2Twitter. TweetiInvi would also require additional filters in order to filter out non-English results which would take additional time. In accordance with the results Linq2Twitter was chosen to be used in the ORM system.

Facebook has a different API than Twitter and requires a different interfacing method and a different component. As the ORM system has to be operated within a Microsoft Windows environment the ORM system will make use of the Facebook SDK for .Net.

C. Result Analysis Method - String Similarity Formula

The results of the web and social networking crawlers that contain the specified keywords will be saved, but a certain portion of the results, such as tags within a web page, will have insignificant meaning. Manually filtering out these results will cost the user valuable time, therefore a similarity filter based on the Levenshtein and Kuhn-Munkres algorithms will be used to filter out unnecessary results. There are other string similarity formulas available, as proposed by Mihalcea [10] and Li [11], but are deemed too complex for this implementation.

The Levenshtein algorithm is used to measure the similarity between two strings by calculating the least number of edit operations that are necessary to modify one string into becoming another [12]. The Kuhn-Munkres algorithm is an algorithm capable of solving linear assignment problem (LAP) instances [13].

The similarity between a result and its keywords can be calculated by breaking up both the result and the keywords into separate lists of their individual terms called tokens, followed by comparing the similarity of the tokens within each list against each other before comparing the tokens in the list of keywords against the tokens in the list of results.

The Levenshtein algorithm is used to compare the similarity between the tokens by providing a set of rules that calculates the cost of changing a token to another by using a series of one step operations. Each one-step operation has an associated cost; substitution costs 2 units whereas cost of insertions and deletions is 1 unit.

After the tokens in each list have been compared the algorithm will compare the tokens between the list of results and the list of keywords. This will return a matrix, which will be solved by the Kuhn-Munkres algorithm. For a full discussion of the string similarity tool, please refer to the dissertation "Development of an Online Reputation Monitor" by G.J.C. Venter at the North-West University

D. Sentiment Analysis Tool

Once a record has been generated and processed its sentiment must be calculated to determine whether the message is positive or negative. Various sentiment analysis tools are available on the internet such as the *AlchemyAPI* [14] and the *Salience Engine* provided by *Lexalytics* [15].

To use the AlchemyAPI the user must first register on the AlchemyAPI website to acquire a *software development kit (SDK)* and two registration keys. In order to determine the sentiment of a sentence the user must use the SDK with the 2 keys to verify the application and pass the sentence that must be analyzed as a parameter. AlchemyAPI will proceed to analyze the sentence and present the user with the result.

The Salience Engine from Lexalytics provides the user with a multi-lingual text analysis engine that can be integrated into systems for business intelligence and social media monitoring. The Salience Engine includes many features, such as sentiment analysis, named entity extraction and summarization.

From features alone the Salience Engine would be better suited for this project, but in order to acquire the Salience Engine the user would have to contact Lexalytics and request access to the system. There are also no pricing opinion available on the website, whereas the AlchemyAPI SDK could easily be acquired once registered. As such the AlchemyAPI was used for this project but the Salience Engine can be investigated in the future.

V. Implementation

The advantages and disadvantages of different implementations of the chosen tools are discussed in this section.

A. *dtSearch Engine – Web crawling*

The scanning of thousands of web pages using the basic dtSearch Engine implementation will take quite long because the dtSearch Engine scans a single page at a time while the other pages are kept in a queue. To overcome this problem multiple instances of the dtSearch Engine are initiated in the ORM and the web pages are divided amongst them.

Multiple instances requires the use of *threading* at the cost of more system resources. To investigate the threading option the number of threads were increased while the CPU and memory usage as well as the internet bandwidth usage were monitored. All tests were done using a 1Mb/s (128 KB/s) internet connection. The results are shown in *Table 16.*

Table 16: Threading comparison

Threads	1	2	4	8	16
Time (s)	2381	1259	722.37	573.41	430.03
CPU Usage (%)	1.5	2.5	4.1	5.0	5.1
Memory Usage (%)	1.4	2.5	4.1	7.08	14.5
Bandwidth (kB/s)	23.67	57.02	98.16	127.18	133.28
Websites loaded	788	788	788	788	788
Stable	Yes	Yes	Yes	Yes	Yes

From *Table 16* it can be seen that as the number of threads increased the amount of time required to scan all the sites decreased, but the usage of the system resources increased. More threads could be used but at 32 threads there were too many simultaneous web connections which caused some of the web crawlers to time-out as they could not receive information fast enough. This problem can be solved with a faster internet connection.

B. *Social networking sites - Twitter*

The Linq2Twitter component provides results when received from the Twitter API. The number of results are not dependent on the computer speed and multiple instances are therefore not needed.

During testing it was noticed that some results from the Linq2Twitter component cannot be used by the ORM service, e.g. non-English results and results that have been previously detected. Saving these results to the database will increase the execution time of the Front-End as the results have to be filtered out each time the results for a specific keyword are requested. To prevent the saving of these results an additional filter was developed that only saves English results. The record has not been detected before and the message of the keyword does contain the keyword. Results failing any of the checks were discarded.

Table 17: Twitter Filter Results

	Test1 1	Test 2	Test 3
Correct	1377	1657	1942
Incorrect Language	3426	3055	3184
Duplicate record	295	830	658
Total	**5098**	**5542**	**5784**
Percentage Correct	**27.0%**	**29.8%**	**33.5%**

The effect of the filter can be seen in *Table 17*. Each result was passed through the filter as it was received. It can be seen that only about 30% of the results are indeed viable for the ORM service.

C. *Social networking sites – Facebook*

Unlike Twitter, the Facebook streaming components do not provide the user with continuous posts but with limited information regarding user and page status *updates*. To retrieve information regarding the API the user will have to perform a search request using another component of the Facebook SDK called the Graph API.

The Graph API will provide the user with various information regarding the latest Facebook public posts such as user ID, the message and the date at which it was created and will always return the latest 25 results. During integration it was discovered that if the Graph API is queried once per minute the ORM system will not lose any results as results are generated very slowly on the Facebook system, but will receive various duplicates. In order to prevent the saving of these results an additional filter was developed to filter out any duplicate results. To test the filter 5 queries were executed each a minute apart. For the 1st query the filter reported 70.4% unique results, for the 2nd 16%, for the 3rd 16.8%, for the 4th 14.4% and for the fifth 24.8%. The initial query contains the most unique results and subsequent queries contain fewer unique results as more results are duplicated and filtered out.

VI. *Results*

A. *Web Crawling*

The Web crawler was loaded with 250 websites, crawl depth of 1. The results for the crawler test are shown in *Table 18.*

Table 18: Web crawler results

ID	Time	#Seed URLs	#Links detected	#Links scanned	Crawl Rate
1	3 hours 31 minutes	16	6332	2877	0,23
2	3 hours 43 minutes	16	4221	1585	0,12
3	3 hours 17 minutes	16	4604	2075	0,18
4	2 hours 29 minutes	16	3182	1175	0,13
5	2 hours 32 minutes	16	3352	1239	0,14
6	2 hours 55 minutes	16	4056	1773	0,17
7	2 hours 40 minutes	16	4085	1555	0,16
8	3 hours 19 minutes	16	4091	1458	0,12
9	2 hours 53 minutes	16	3897	1690	0,16
10	2 hours 50 minutes	16	3611	1317	0,13
11	3 hours 29 minutes	15	4302	1617	0,13
12	3 hours 11 minutes	15	4805	1805	0,16
13	2 hours 5 minutes	15	2741	1184	0,16
14	3 hours 6 minutes	15	6631	2194	0,2
15	3 hours 34 minutes	15	3683	1326	0,1
16	2 hours 4 minutes	15	3014	1028	0,14
All	2 hours 59 minutes	250	66607	25898	2.41

From *Table 18* it can be seen that the web crawler scanned the web pages at a rate of 2.41 web pages per second. This will result in scanning 1000 web pages within 40 minutes. If a higher scan rate is required it can be realized by using a more powerful computer with a faster internet connection. To test whether the web was scanned correctly a single keyword was chosen and the number of instances found were verified against the original web page. All instances were found. The test was executed on a computer with a 3GHz CPU, 16GB RAM and a 1Mb/s internet connection.

It should be noted that less links were scanned than detected. This occurred because of duplicate links that were detected within web pages such as style sheets, as well as links that are not allowed to be crawled, as dictated by the website.

B. Twitter Scanning

The Twitter Scanner was loaded with 26 keywords and executed for 1 hour 35 minutes. During the execution 74 201 results were detected at an average rate of 13.01 results per second. Of the 74 201 results 51 556 results were in a non-English language and 2 882 were duplicates of previously detected results. When passed through the similarity filter 6 542 results were removed, which provides the ORM service with 13 221 viable results. Due to the nature of social networking sites it will be impossible to verify the results against the Twitter database and therefore only the results as presented by Linq2Twitter can be verified.

C. Facebook Scanning

The Facebook Scanner was loaded with 26 keywords and executed for 1 hour 54 minutes. During the execution 13 250 results were detected at an average rate of 2.12 results per second. Of the 13 250 results 11 469 were duplicates of previous detected results which left 1 781 results. When passed through the similarity filter 766 results were found viable for the ORM service. Due to the nature of social networking sites it will be impossible to verify the results against the Facebook database and therefore only the results as presented by Facebook SDK can be verified.

D. Sentiment Analysis

To test the sentiment analysis tool the sentiment of 125 random results from each of the three web crawlers were calculated and manually verified. For the web crawler 91.8% of the sentiments were calculated correctly, for the Twitter Scanner 91.2% and for the Facebook Scanner 95.2%.

VII. Conclusion

This paper demonstrated an online reputation monitoring system capable of monitoring both the web and various social networking sites. The system is capable of extracting information from the internet in real time from various user-defined web sites as well as Twitter and Facebook and can determine which results are most likely to contain any relevant information via various filters. This improves upon existing ORM applications as neither BrandsEye nor Brand.Com can simultaneously scan the internet, Twitter and Facebook nor can any of the mentioned systems grant the user immediately access to any historical data. For a full discussion of the results and comparisons of the ORM system, please refer to the dissertation “Development of an Online Reputation Monitor” by G.J.C. Venter at the North-West University.

While this system is capable of performing ORM services there are various areas available for future research. Development of an offline sentiment analysis tool as well as a custom web crawler will greatly improve the cost-efficiency of the system. Some of the included filters could also be further improved upon; while the Twitter Scanner will automatically filter out the non-English results, this feature is not available for the Facebook Scanner.

VIII. Bibliography

[1] A. B. Ayanwale, T. Alimi and M. A. Ayanbimipe, "The influence of advertising on consumer brand preference," *Journal of Social Science,* vol. 10, no. 1, pp. 9-16, 2005.

[2] S. J. Hoch and Y.-W. Ha, "Consumer learning: advertising and the ambiguity of product experience," *Journal of consumer research,* pp. 221-223, 1986.

[3] F. F. Reichheld, "The one number you need to grow," *Harvard business review,* vol. 81, no. 12, pp. 46-55, 2003.

[4] N. Hu, L. Liu and J. J. Zhang, "Do online reviews affect product sales? The role of reviewer characteristics and temporal effects," *Information Technology and Management,* vol. 9, no. 3, pp. 201-214, 2008.

[5] G. Pant, P. Srinivasan and F. Menczer, in *Crawling the web*, Springer, 2004, pp. 153-177.

[6] Web2Mine, "Easy Web Extract Software," Web2Mine, 2013. [Online]. Available: http://webextract.net/. [Accessed July 2013].

[7] dtSearch, "How dtSearch Works," dtSearch, [Online]. Available: http://www.dtsearch.com/PLF_howdtworks.html. [Accessed March 2013].

[8] M. Annett and G. Kondrak, "A comparison of sentiment analysis techniques: Polarizing movie blogs," in *Advances in artificial intelligence*, Springer, 2008, pp. 25-35.

[9] R. Feldman, "Techniques and applications for sentiment analysis," *Communications of the ACM,* vol. 56, no. 4, pp. 82-89, 2013.

[10] R. Mihalcea and P. Tarau, "A language independent algorithm for single and multiple document summarization," *Proceedings of IJCNLP,* vol. 5, 2005.

[11] Y. Li, D. McLean, Z. A. Bandar, J. D. O'shea and K. Crockett, "Sentence similarity based on semantic nets and corpus statistics," *Knowledge and Data Engineering, IEEE Transactions on,* vol. 18, no. 8, pp. 1138-1150, 2006.

[12] RosettaCode.org, "Levenshtein Distance," 25 January 2014. [Online]. Available: http://rosettacode.org/wiki/Levenshtein_distance. [Accessed 3 March 2014].

[13] D. Dasgupta, G. Hernandez, D. Garrett, P. K. Vejandla, A. Kaushal, R. Yerneni and J. Simien, "A comparison of multiobjective evolutionary algorithms with informed initialization and Kuhn-Munkres algorithm for the sailor assignment problem," in *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, ACM, 2008, pp. 2129-2134.

[14] AlchemyAPI, "About Us," Alchemy API, 2014. [Online]. Available: http://www.alchemyapi.com/company/about-us/. [Accessed 17 April 2014].

[15] Lexalytics, "Salience Engine," Lexalytics, 2014. [Online]. Available: http://www.lexalytics.com/technical-info/salience-engine-for-text-analysis. [Accessed 28 04 2014].

G.J.C. Venter received his undergraduate degree from the North-West University (NWU) in Potchefstroom in 2012. He is currently studying towards his Masters in Engineering at the same institution. His research includes web data extraction and online reputation monitoring.

Bibliography

[1] N. Hu, L. Liu and J. J. Zhang, "Do online reviews affect product sales? The role of reviewer characteristics and temporal effects," *Information Technology and Management,* vol. 9, no. 3, pp. 201-214, 2008.

[2] F. F. {Reichheld, "The one number you need to grow," *Harvard business review,* vol. 81, no. 12, pp. 46-55, 2003.

[3] E. Garner, "Customer Feedback Techniques," Business Know-how, [Online]. Available: http://www.businessknowhow.com/marketing/customer-feedback.htm. [Accessed April 2013].

[4] BrandsEye, "About Overview," Brandseye, 2013. [Online]. Available: http://www.brandseye.com/about-us/overview/. [Accessed January 2013].

[5] Brand.Com, "Brand.com," Brand.com, 2013. [Online]. Available: http://www.brand.com/biz-control-plans.html. [Accessed September 2013].

[6] A. Engelbrecht, Interviewee, *BrandsEye employee.* [Interview]. 02 August 2014.

[7] A. B. Ayanwale, T. Alimi and M. A. Ayanbimipe, "The influence of advertising on consumer brand preference," *Journal of Social Science,* vol. 10, no. 1, pp. 9-16, 2005.

[8] S. J. Hoch and Y.-W. Ha, "Consumer learning: advertising and the ambiguity of product experience," *Journal of consumer research,* pp. 221-223, 1986.

[9] E. Amigó, J. Artiles, J. Gonzalo, D. Spina, B. Liu and A. Corujo, "WePS-3 evaluation campaign: Overview of the online reputation management task," in *CLEF 2010 (Notebook Papers/LABs/Workshops),* 2010.

[10] G. Pant, P. Srinivasan and F. Menczer, in *Crawling the web,* Springer, 2004, pp. 153-177.

[11] Web2Mine, "Easy Web Extract Software," Web2Mine, 2013. [Online]. Available: http://webextract.net/. [Accessed July 2013].

[12] dtSearch, "How dtSearch Works," dtSearch, [Online]. Available: http://www.dtsearch.com/PLF_howdtworks.html. [Accessed March 2013].

[13] I. Lunden, "Analyst: Twitter Passed 500M Users In June 2012, 140M Of Them In US; Jakarta 'Biggest Tweeting' City," 30 July 2012. [Online]. Available:

http://techcrunch.com/2012/07/30/analyst-twitter-passed-500m-users-in-june-2012-140m-of-them-in-us-jakarta-biggest-tweeting-city/. [Accessed 11 March 2014].

[14] Pear Analytics, "Twitter Study - August 2009," Auguest 2009. [Online]. Available: http://web.archive.org/web/20110715062407/www.pearanalytics.com/blog/wp-content/uploads/2010/05/Twitter-Study-August-2009.pdf. [Accessed 11 March 2014].

[15] Twitter, "REST API v1.1 Resources," Twitter, 2013. [Online]. Available: https://dev.twitter.com/docs/api/1.1. [Accessed 11 March 2014].

[16] Twitter, "The Streaming API," Twitter, 2013. [Online]. Available: https://dev.twitter.com/docs/api/streaming. [Accessed 11 March 2014].

[17] D. Rushe, "Facebook posts record quarterly results and reports $1.5bn profit for 2013," TheGuardian, 29 Jan 2014. [Online]. Available: http://www.theguardian.com/technology/2014/jan/29/facebook-record-quarterly-results?CMP=EMCNEWEML6619I2. [Accessed 11 March 2014].

[18] T. Nasukawa and J. Yi, "Sentiment analysis: Capturing favorability using natural language processing," in *Proceedings of the 2nd international conference on Knowledge capture*, ACM, 2003, pp. 70-77.

[19] M. Annett and G. Kondrak, "A comparison of sentiment analysis techniques: Polarizing movie blogs," in *Advances in artificial intelligence*, Springer, 2008, pp. 25-35.

[20] R. Feldman, "Techniques and applications for sentiment analysis," *Communications of the ACM*, vol. 56, no. 4, pp. 82-89, 2013.

[21] P. Blinov, M. Klekovkina, E. Kotelnikov and O. Pestov, "Research of lexical approach and machine learning methods for sentiment analysis," Russia, 2012.

[22] CodePlex, "Html Agility Pack," 11 Jul 2012. [Online]. Available: http://htmlagilitypack.codeplex.com/. [Accessed March 2013].

[23] dtSearch Corporation, "About dtSearch Corp," dtSearch Corporation, 2013. [Online]. Available: http://dtsearch.com/dtsoftware.html. [Accessed January 2014].

[24] dtSearch, "dtSearch Text Retrieval Engine Programmer's Reference 7.73: Building and Maintaining Indexes," dtSearch, 2013. [Online]. Available:

http://support.dtsearch.com/webhelp/dtsearchCppApi/frames.html?frmname=topic&frmfile=Overviews.html. [Accessed 07 March 2014].

[25] dtSearch, "ContractIQ Adds dtSearch to their Contract Management Solution in Support of the Commercial Industry and IC Clients," dtSearch, [Online]. Available: http://www.dtsearch.com/CS_Contract_IQ.html. [Accessed 09 August 2014].

[26] dtSearch, "Densan Consultants developed NewsLink," dtSearch, [Online]. Available: http://www.dtsearch.com/CS_densan_consultants.html. [Accessed 09 August 2014].

[27] dtSearch, "ATS Tackles World Trade Law with dtSearch," dtSearch, [Online]. Available: http://www.dtsearch.com/cs_networkats.html. [Accessed 09 August 2014].

[28] CodePlex, "Tweetinvi a friendly Twitter C# API," The TweetInvi Team, 6 March 2014. [Online]. Available: https://tweetinvi.codeplex.com/. [Accessed 17 March 2014].

[29] CodePlex, "Linq2Twitter," 7 Feb 2014. [Online]. Available: http://linqtotwitter.codeplex.com/. [Accessed 11 March 2014].

[30] RosettaCode.org, "Levenshtein Distance," 25 January 2014. [Online]. Available: http://rosettacode.org/wiki/Levenshtein_distance. [Accessed 3 March 2014].

[31] D. Dasgupta, G. Hernandez, D. Garrett, P. K. Vejandla, A. Kaushal, R. Yerneni and J. Simien, "A comparison of multiobjective evolutionary algorithms with informed initialization and kuhn-munkres algorithm for the sailor assignment problem," in *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, ACM, 2008, pp. 2129-2134.

[32] R. Mihalcea and P. Tarau, "A language independent algorithm for single and multiple document summarization," *Proceedings of IJCNLP*, vol. 5, 2005.

[33] Y. Li, D. McLean, Z. A. Bandar, J. D. O'shea and K. Crockett, "Sentence similarity based on semantic nets and corpus statistics," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 18, no. 8, pp. 1138-1150, 2006.

[34] G. V. Bard, "Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric," *Proceedings of the fifth Australasian symposium on ACSW frontiers*, vol. 68, pp. 117-124, 2007.

[35] F. J. Damerau, “A technique for computer detection and correction of spelling errors,” *Communications of the ACM* , vol. 7, no. 3, pp. 171-176, 1964.

[36] V. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals,” *Academician P.S. Novikov*, vol. 164, no. 4, pp. 845-848, 1965.

[37] AlchemyAPI, “About Us,” Alchemy API, 2014. [Online]. Available: http://www.alchemyapi.com/company/about-us/. [Accessed 17 April 2014].

[38] M. Ogneva, “How Companies Can Use Sentiment Analysis to Improve Their Business,” Biz360, 19 April 2010. [Online]. Available: http://mashable.com/2010/04/19/sentiment-analysis/. [Accessed 08 September 2014].

[39] Facebook, “Public Feed API,” Facebook, 2013. [Online]. Available: https://developers.facebook.com/docs/public_feed/. [Accessed 27 March 2014].

[40] CodeProject, “An improvement on capturing similarity between strings,” Code Project, 5 Aug 2005. [Online]. Available: http://www.codeproject.com/Articles/11157/An-improvement-on-capturing-similarity-between-str. [Accessed March 2013].