2015-03-06

An Open Source Implementation and Evaluation of a Federated Searching Service Based on File Based Index and Relational Database Index

Alan Liu
Technological University Dublin

2015-03-06

An open source implementation and evaluation of a federated searching service based on file based index and relational database index

Alan Liu

An open source implementation and evaluation of a federated searching service based on file based index and relational database index

Alan Jue Liu

A dissertation submitted in partial fulfillment of the requirements of Dublin Institute of Technology for the degree of M.Sc. in Computing (Data Analytics)

March 2015

Declaration

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Data Analytics), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the test of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: Alan Jue Liu

Date: 22/02/2015

Abstract

One of the greatest challenges in the modern information world is the storage of data and the ability to extract usable knowledge from it, to enable enterprises to gain insight that can be leverage to succeed in the competitive environment. Data is constantly being generated and collected at a rapid pace from users and customers via many sources such as intranet, internet and other smart devices. To effectively use the collected data, knowledge must be extracted from it to allow effective and efficient business planning.

While the core challenge of data analytics is to extract hidden non-trivial information form large datasets there is the more immediate concern of how to effectively index information so that both data and the converted knowledge can be recalled quickly and accurately.

This thesis examines the theoretical challenges in reading semi structured data (i.e. a website), converting it to an effective storable format and how to best search the data from an index. With the knowledge gained from the theoretical challenges an investigation will follow to see if it is possible to use only open source components to build a framework that will in part solve the challenges of enterprise indexing. Finally from the experiment a usable framework for a federated search system based on a file based index core and a relational database based core would available for further refinement.

The experiment and evaluation reveals that using the proposed open source frameworks and architecture it is possible to combine both the file based index and the relational database index to form a federated search framework with the best attributes from both as well as solving the big challenges around scalability and systems integration in an enterprise environment.

Despite the slower processing times of a relational database index compared to a file based index both contains advantages and disadvantages from the performance and maintenance perspective. Rather than picking one technology over the other the experiment shows it is possible to have the two working in a non-intrusive way to index the same dataset within one system.

Key words: indexing, schema, open source, federated, search, apache nutch, apache solr, webservice, search, searching, file index, relational database index

Acknowledgements

I would like to thank foremost my supervisor Deidre Lillis for her time and patience in guiding me through the process of writing this thesis. Her attention to detail was invaluable at correcting the many issue with the initial drafts.

I would like to thank my colleagues Jacob, James, Patrick and Sean for throwing ideas around that lead to the architecture used in this experiment.

I would like to thank my family for mind my little girl giving me time to finish this thesis in a timely manner.

Last I would like to thank my wife Suyi in the support she gave me during the late nights of coding to realize the framework used to evaluate the experiment.

Contents

Table of Figures

INTRODUCTION

1.1 Background

In recent year the advent of personal mobile technology and personalized services meant that more information is saved in digital form than ever before. As people uses more software in their daily lives to manage everything the result is that information is often fragmented, sitting in various locations such as laptops, mobile phones (McHugh, et al., 1998) and across the internet such as Gmail, Facebook and so on.

Enterprises are no exception to this data explosion. Storing ever more data across multiple departments and working groups, the management and tracking of these data (often varying in formats and structure) calls for the need of an advanced index solutions (Hawking, 2004).

In the modern world where the notion “knowledge is power” is accepted as true for both people and businesses, the need to store knowledge is ever more important. At the same time we can see that both businesses and government services are moving from pen and paper (or paper and letters) into SAAS (Software as a service) which means that just about all future information is stored in digital format.

As an enterprise the need to be able to store large datasets already provides enough of a challenge without the additional overhead of thinking about indexing and adding meta-data that is necessary to accurately keep track of the knowledge contained within. As a data scientist or the chief data officer (CDO) there is two primary concerns that need to be addressed when it comes to storing data. The first is the need to be able to locate something when users look/search for it. This is known as direct information search. The second is the much harder task of creating association and relevance between data so that users can be made aware of knowledge that they might not be aware of during a discovery process.

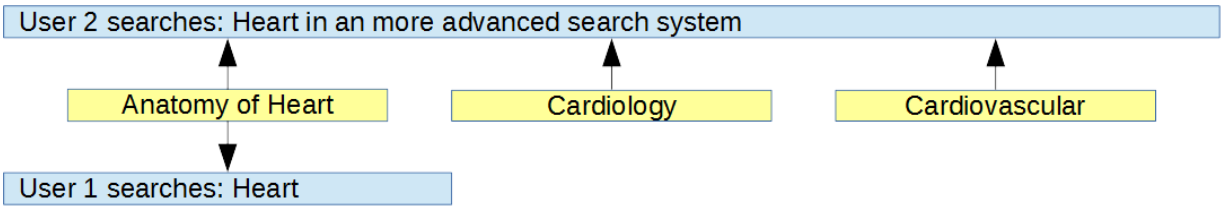


Figure 1 Understanding taxonomy instead of word matches

If looking at the user's perspective, people often rely on using the searching functions within applications to find the right data with the right relevance to their search terms. Daily experiences of people using services like Google, Bing and any major websites has reinforced the idea of searching for information to the point that expectation of a search within any application should be on par with actual search engines. This means that modern website and internet based services simply would not be complete without some form of search functionality within the user experience. Information which is not retrievable via searching is only available to only the few who knows about the content for discovery, as time goes on this data could become lost forever never to be discovered again (Fagin, et al., 2003).

While the interface for a good users search interface feature follows the principle of simplicity is better, the technical challenges behind the searching is often far more challenging. With ever increasing expectation on searching, indexing data has turned from a relatively minor component in any enterprise system to a key component which warrants forethoughts during the initial design phases. (Singh, et al., 2007)

An effective and efficient search function has several additional processing stages that are much more complex than just a direct string match. While the user remains oblivious to the complicated background processing that happens while a term or string is searched, a good search engine would apply a number of calculation and rules to the search term.

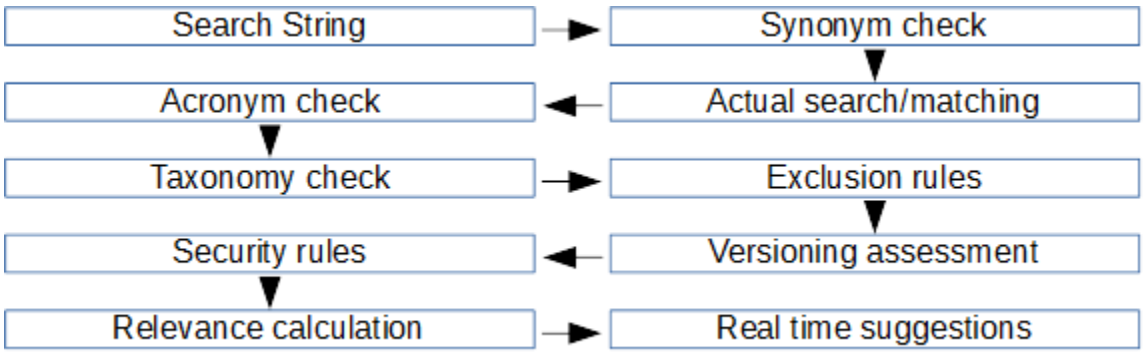


Figure 2 The steps in searching via a string

It is not uncommon to see a system backend to execute a string match, fetching the results and then applying additional logic to rank and order the results. Other features such as an effective management of synonyms and acronyms can quickly make managing a simple index manually very complex (Hawking, 2004). The additional post processing can become a significant performance bottleneck if the returned result set is very large (thousands of matches). This is then increased even further taking into consideration that there could be thousands of concurrent users.

1.2 Overview

During the pre-thesis concept preparation phase, it appears that there is a lot of literature describing the problem of data fragmentation and the theoretical challenges faced when trying to constructing a consolidated index for various file structure and type, none focus on whether it was possible to use the vast open source libraries available currently to construct a solution that might go some way to solve these enterprise indexing problems. During this thesis and the design of the experiment the aim has been to understand the challenges of creating a federated indexing framework and to create a robust web based service to index data via file and database using open source software tools.

The system created should go some way to resolve the problem of amalgamating different data sources and/or existing indexes to create a federated search service. During this experiment common data will be pushed into a file based indexing framework and a relational based database also storing the same indexing (or as close representation as possible) schema to

evaluate their performance. Difference index dataset will also be integrated evaluate the index federation functions when aggregating the results from multiple indexers.

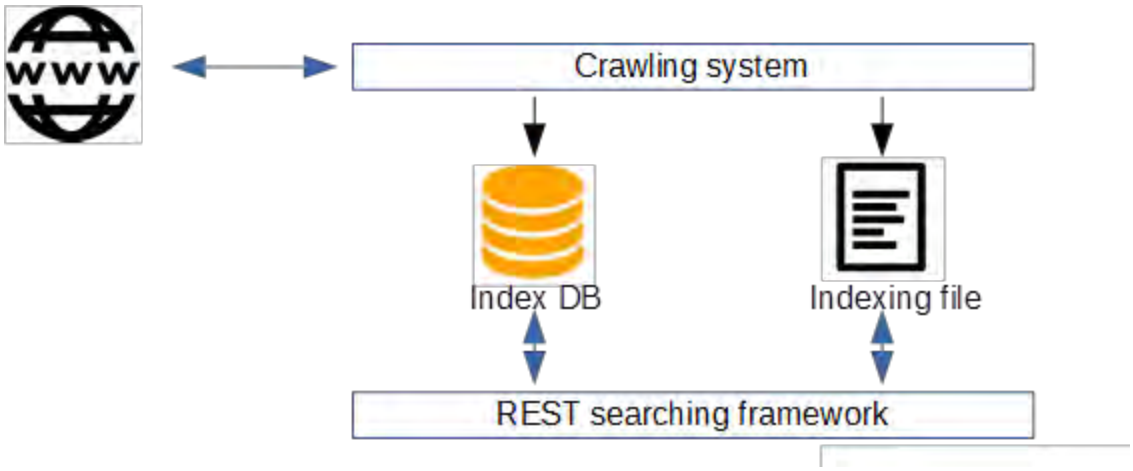


Figure 3 A simple view to the experiment

Another evaluation assed is how to efficiently index semi structured data, for the purpose of this experiment, websites are used because they contains typical page content composed of unstructured text (and pictures) with the page meta-data (structured) but often not uniform across different domains. Due to the varying data formats that exists in a real enterprise system and the time allowed for this experiment, the main focus of data type was limited to webpages for the evaluation of varies components that makes up the finished federated indexing system.

To create a good indexing system, it must be able to achieve three distinct functions. The first is access information and the ability to read it without human interpretation. Followed by the ability to convert this information into an efficient structure and store it. Finally the retrieval process should allow any client system to retrieve the stored information by providing key search terms. For each of these tasks there is often more than one software framework that can be used.

Once a working federated indexing system is deployed, it would be the feasible to evaluate a key performance question which is that given a common software stacks sharing all the components but having two distinct cores at the lowest level, which is the better option in terms of performance and ease of integration for an enterprise.

1.3 Aims and Objectives

The primary aims of this experiment are:

To research the problem of indexing semi structured data across multiple data sources.

To research and evaluate the software components in solving each of the challenges below separately

- Crawling data via the HTTP protocol (commonly used in intranet and the internet)
- Parsing crawled data into structured meta data for indexing
- The storing of index meta data
- The retrieval of index data using key words
- The relevance and speed of searches
- Construction of a parent service calling on multiple individual indexers
- The amalgamation of multiple indexer's results
- The ability to serve the results in a homogenous web service

To utilize existing frameworks and where necessary use custom logic to compliment the frameworks to construct a full software stack capable of providing a federated service with multiple child indexes

To evaluate the complexity of integrating the technologies and frameworks into a homogenous Restful federated indexing web service

To evaluate how well they work in order to index a common dataset (the same dataset will be stored to both types of indexes)

- Ease of use – how easy it is to deploy the framework
- Ease of expansion – How well do the framework scale
- Ease of integration – How easy it is to make the framework work with others
- Speed and performance of the framework – How fast, how relevant, how accurate are the framework in doing what its suppose to do

To evaluate the performance of dedicated file based indexers and custom created relational database based indexing solutions

- The speed of which data can be written into the index
- The speed the data can be read from the index
- The speed that data can be searched by the index

To assemble a working vertical software stack in an effort to solve the challenges and to allow an measurement of the performances

1.4 Research Problem

The core challenges with searching and indexing enterprise system can be broken into two categories. One occurs where an existing system is already deployed without efficient indexing for search to work properly and the second is the design and planning of an efficient indexing service to work with all components of a new system (Singh, et al., 2007).

In the first case, many organizations would have a mixture of own (often internally) developed systems and third party developed systems (likely to be customized) for specific business uses. While there exists standard intercommunication protocol for some types of software such as HL7 for medical software systems, most do not have a common search interconnect standard. As a result the search and index components are often abstracted behind different services provided by the difference venders that offers no uniform conformity.

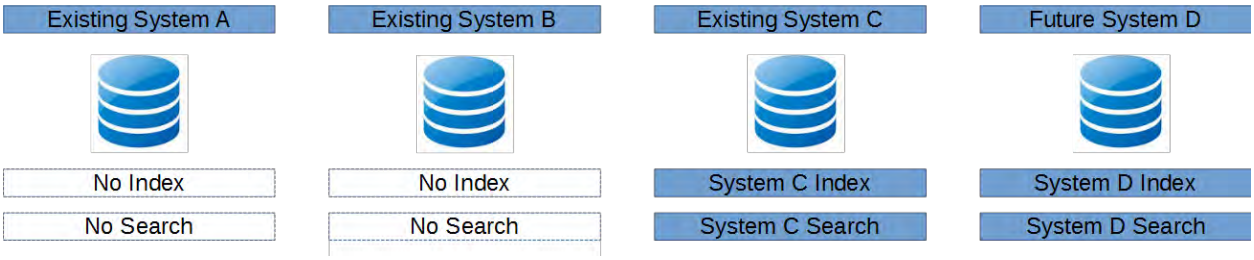


Figure 4 Enterprise systems without consolidated index

To consolidate the search capability for such a scenario would be challenging as a federated service will need to encapsulate all of the existing search services as well as any data that might not exposed through a search service.

The second case often follows the first case which is once an effective search framework is developed and deployed how it can be robust and customizable enough to consume new data sources via service or direct access.

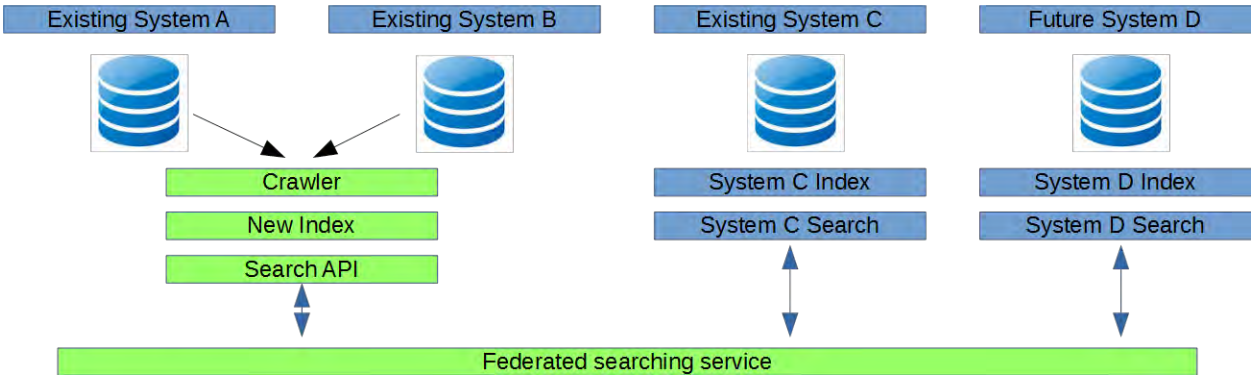


Figure 5 Enterprise system with a federated search index service

McHugh et al (McHugh, et al., 1998) highlights the issue of varied data types requiring bespoke file index schema or specially designed meta-data indexing table for a relational database to enable an efficient tracking and indexing of all the data type in a homogeneous way. As different data contains different attributes, it would be important to introduce the correct structure to store the data's Meta-data to enable the indexes to be common as to have the searching to be complete.

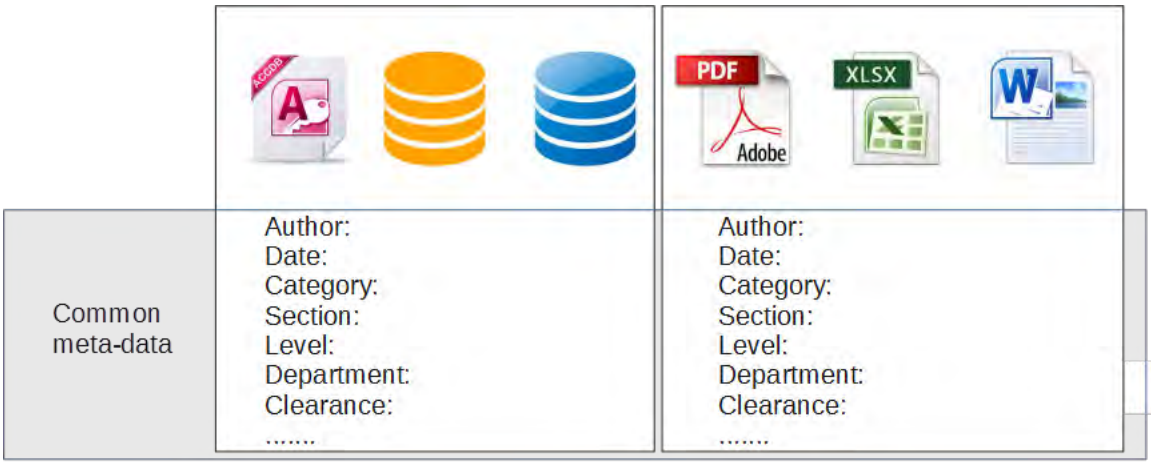


Figure 6 Meta data added to files

Once the challenges associated with the storing of indexed data are solved, Giusti et al (Giusti, et al., 2010) highlights the issue where data stored in multiple locations both physically and virtually would need to be aggregated via introducing a federated service that can somehow collect all the data into a common indexing solution.

Along with the data an enterprise have visibility and knows about, Hawking (Hawking, 2004) points out that there could potentially be multiple versions with different up to date status of enterprise files containing additional insights stored within the local file system which data governance might be oblivious to and are unable to access directly without effective crawling solutions. In such cases an effective crawling system is crucial.

File based indexing system also introduce additional capability which might not be thought of initially such as semantic word matching, ordering and sorting and real time word boosting (Rafalovitch, 2013).

At a high level the ideal indexing system will need to be able to provide two components:

- An indexing component of some sort that can be operated on the CRUD (create, read, update and delete) principles
- A external component which can "crawl through various datasets" and interact with the above service component

The security implication when accessing resources for indexing should follow the approach that the crawler will be given super user or root access. Subsequently the security layer should be implemented as the governance rules governing the actual access to the indexes. This approach is preferable than not including certain restricted documents during the initial crawl as that would mean they would need other forms of indexing to keep track of the contents.

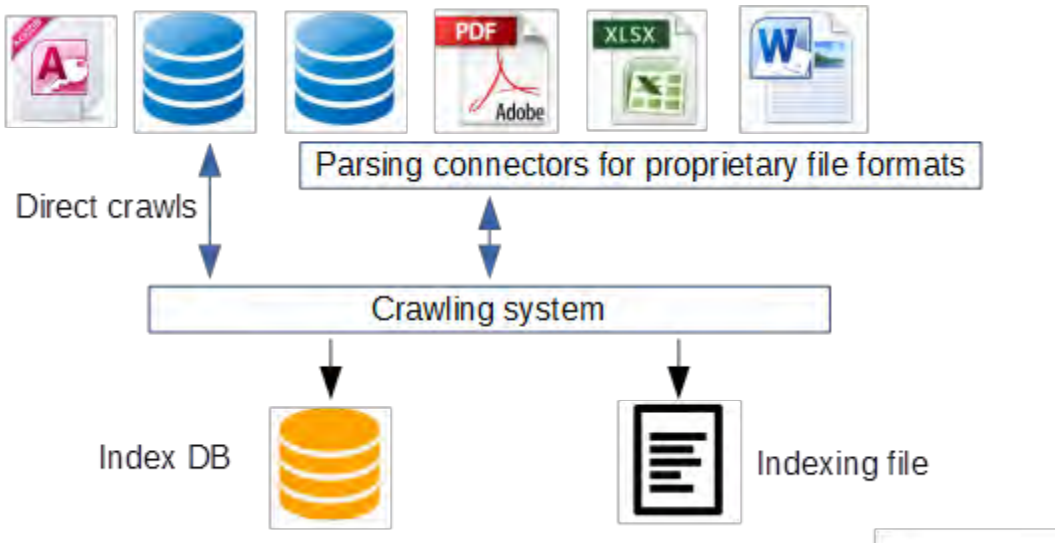


Figure 7 The ideal crawler

In order to have an efficient and effective solution for these problems, an analysis is required to understand the modern architecture and implementation perspectives on solving these problems via technical white papers and academic journals. At the same time experiences within the industry will also be used to supplement these architectures and implementations.

Rather than focusing on all possible architecture combinations, this project will focus specifically on using DB Schema as indexing schemas versus File based indexing system schemas due to the time limitation. The key contribution of this project is to provide a basis framework in which a federated searching service can be built on both existing data and new services. Following on will be to further investigate the interaction and roles that relational databases and file based indexing can play in complimenting each other or in some cases replace each other.

1.5 Organization of The dissertation

1.5.1 Literature Review

The purpose of the literature review is to understand the challenges in indexing multi sourced information systems and content libraries across different geo locations for an enterprise. Following the understanding of the challenges is to assess the amount of research that has been given into the various aspects and if possible serve as a guidance in the development of an indexing framework.

Ideas and existing concepts are analyzed in this review section and any conclusion gathered towards a finding is concluded and included in the design of the experiment and setup. Any findings that are not included in the experiment due to scoping are noted too.

1.5.2 Framework proposal and design

The purpose of the framework proposal and design section is to design how the federated Restful service will be architected and deployed to meet the requirements gathered in the literature review section.

This section also introduce the necessary components that is needed to setup the experiment. Each of the chosen component is further explained and how it fits into the overall goals of the experiment.

1.5.3 Experiment setup

Based on the design of the previous section Experiment setup goes into detail the setup process of each of the chosen software framework in a practical manner. This section would then provide insight into the questions asked around how easy it is to extend the modular components to add additional functional components such as Crawlers, file index cores and database table cores.

1.5.4 Evaluation

Evaluation documents the results of the experiment of which some are numeric and some conceptual. The numerical element of the result discovers the performance aspect of the framework namely the speeds in which everything performs at. The conceptual discoveries are results governing issues such as how easy it is to add additional software component such as cores, crawlers and data sources.

2 LITERATURE REVIEW

2.1 Introduction

The purpose of the literature review serves two purposes. The first is to increase the understanding of how existing crawling system work (or should work) and the second is to see what the current research is like for the various topics in the indexing field.

The primary research method is via literatures provided electronically through ACM, Google scholar and Online references at the Apache Software foundation. The resources at the Apache software foundation mainly consists of how to articles to explain how certain software technology is used for specific indexing problems.

The core literature reviewed was focused on the topic of how to deploy and create indexing frameworks for organizations that is of large size. Attributes applied to such organization are geo location spread, the need to integrate indexes with existing and new software/data sources and the best way to deploy crawling and new indexing cores.

The literature of indexing can be split into two distinct areas. The first type of papers deals with the conceptual challenges with managing and organizing data or what is commonly termed as taxonomy. Among the analysis of these we can find the main challenges that enterprises and companies face in managing its data. The second type of the research papers deals with the technical challenges in creating systems to manage taxonomy effectively, these include proposed architectures and focused solutions to smaller subsets of the overall architecture. Some of the reviewed papers covers both.

The growing amount of data being collected around the world by companies is growing at a phenomenal rate. According to EMC (EMC, 2013) the digital data collection will increase by 40% year on year or effectively doubling every 2 years. This represents both a challenge to business and an opportunity in extracting new and useful data to support business decisions and increase revenue. With any amount of large datasets the key in making it useful comes from the ability to extract business intelligence from it (Ranjan, 2009). In order to make the data accessible and consumable the data it must be index so that it can be retrieved and stored correctly. According to David hawking and the IDC report entitled "The high cost of not finding information" (Hawking, 2004) enterprises can lose a vast amount of efficiency in information mismanagement where previous vital information is lost to new employees joining the company. As such the result is often wasted initial effort to document a process or in worse cases poor business decision due to inaccurate information. Many enterprises with already deploy systems can perform poorly with new out of the box indexing solution where the architecture may prove difficult in integrating with the new protocols.

In the paper "Challenges in enterprise search" the author David Hawking presents a good overview of the problems posed in enterprise search. His paper deal within the problem envelope and do not include any solution both in theory or technical to solve the problems that he documents.

The next few sections describes the key attributes that must be considered in building a federated searching service.

2.2 Accessing the data

The first problem shared by many research is the ability in accessing the information for indexing. While it can be assumed that smaller companies or organizations has a consolidated point for storing data such as a centralized file store and several databases this would not be the case for larger international organizations. Most modern day large organization share information using Content Management Systems (CMSs) which is typically accessed via the HTTP protocol similar to the internet (Livne, et al., 2010). This means that to access this

information Web browsers can be used and the viewable content is similar to the internet. The following is an analysis on how to access such HTTP based systems (Fagin, et al., 2003).

2.2.1 Crawler navigation strategy

Crawling strategy is a key attribute that affects the performance of a crawler (Hafri & Djeraba, 2004). The crawling strategy of a crawler defines several attributes of how it access information and how to understand the content of the crawl. Navigation strategy can be thought of as how the crawler initiates the crawl when it is given a parent URL(s) represents the source HTML webpages to which the crawl would start on. Upon completion of indexing theses seed URL(s) the crawling strategy would go on to decide how navigation occurs to the child URLs that is parsed from the parent URL(s). Rules can be introduced here as to what kind of URL(s) should be excluded or treated differently to all other URL(s) (Najork, 2009). Below is an analysis of the crawling strategies encountered during the literature review.

2.2.1.1 Breadth first crawling

The most common way of crawling and the default method by which Apache Nutch crawls is breadth first crawling (Baeza-Yates, et al., 2005). In this strategy a parent URL is crawled and any hyperlinks gets queued up and crawled subsequently. As such level one is the head which is the starting URL and level two would be the hyperlinks on the head with third level created from the second level URLs.

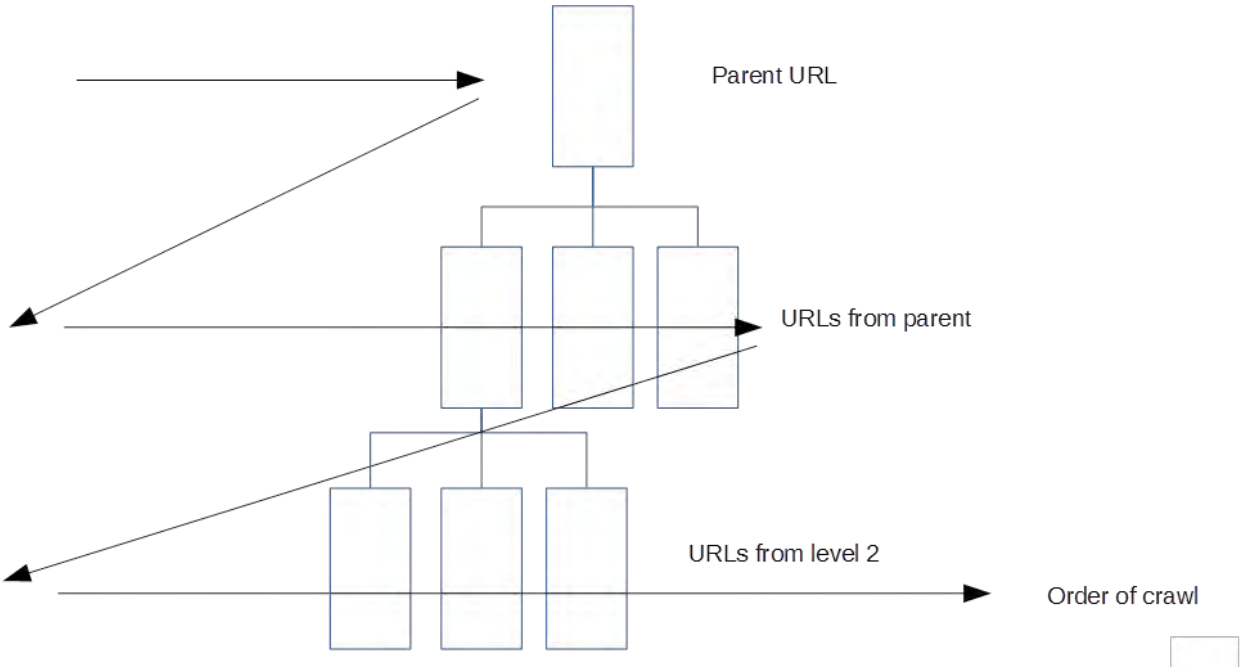


Figure 8 Breadth first crawling

2.2.1.2 Repetitive crawling

For data that are critical and must be kept up to date repetitive crawling (Hafri & Djeraba, 2004) allows a crawler to be setup so that it would crawl the limited amount of URLs over and over without attempting to index anything outside its crawl resource set.

2.2.1.3 Deep web crawling

For deep web crawling (Hafri & Djeraba, 2004) the crawler would navigate similarly to the breadth first crawling strategy but instead of only focusing on HTML content it would also attempt to index content found on the webpages such as text files, pdfs and word documents.

For this to be achieved parsers and content connectors are necessary which will be covered section 2.3

2.2.1.4 Conclusion findings

Based on the review and the documentation based on the principle of the apache Nutch crawler the out of box behavior of Nutch is the breadth first crawling which will serve as the crawling

strategy of this experiment as it is easy to understand and sufficient in providing a comprehensive index from a master URL.

The choice of using Nutch is mainly due to limited alternative if the software stack remains purely Java based. The only Java alternative returned via google search are Crawler4J (Yasser Ganjisaffar, 2014) which the API documentation showed far less functionality than Apache Nutch.

2.2.2 Dealing with multiple data sources at different locations

When an organization spans across the globe, knowing what information is stored where is further complicated by the fact that geographically the data is distributed. This means that either each of the location treat its data and index separately and then combine them or that the data would have to be made available to the master indexing system (which can be thought as a high security risk if it is across continents). Below are an evaluation of the most common index setups.

2.2.2.1 Central global index and crawler

Based on the Googlebot (Nancy Blachman, 2014), this is the most common way to index openly available information. Bi-directional traffic is required for the index to access the data belonging to each of the companies at different locations and then (going the other way) each of the company sites need to access the global index to find data that is not of local source (and local because it does not store local index).

Particular attributes of this architecture are a single schema for all forms of data. When cross continent this also means that the index schema would need to deal with multiple languages.

Complicated security is required to ensure confidential and sensitive data is only accessible by authorized crawlers from the designated indexing system (i.e. stopping the likes of Googlebot accessing internal system information).

Massive bandwidth requirement for the crawler to crawl through what could be terabytes or petabytes of data (Kennedy, 2014).

Reverse look up uses unnecessary high bandwidth due to the index not being in the same location. This can be potentially solved by replicating the index such as what Google do with google maps. Again this introduces overhead.

Single point of failure unless the indexing system is replicated across not only servers but locations. Otherwise a routing issue to the location of the index could cause the searches from all locations to fail.

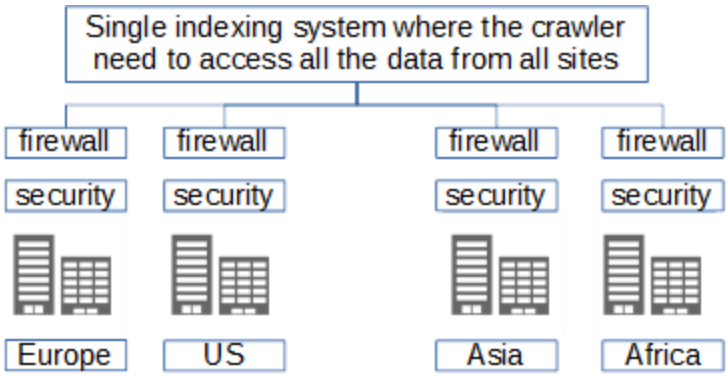


Figure 9 Single indexing system to index and crawl all data (i.e. Google)

2.2.2.2 Central global index and local crawlers

In the second type of information access the information is indexed locally by local crawlers (Giusti, et al., 2010) thereby circumventing the problem with the earlier design so that local knowledge can be applied in know where information is. Once the index information is prepared and readied by the crawlers then it is pushed into a global index via HTTP reset or SOAP services.

Using this architecture there is management overhead as each of the crawlers will need to be aware of the global schema and its changes. Additionally each of the local crawler will need to manage the API to the global indexing service.

The main benefits of this architecture are a reduction in crawling bandwidth as the crawlers are now local and can access data directly over LAN or WAN. Security can also be simplified due to a single point into the index instead of multiple ones at each company location.

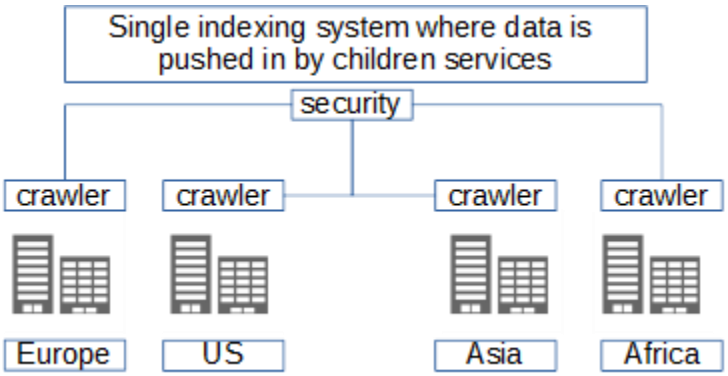


Figure 10 Single indexing system to track all data, local crawlers

2.2.2.3 Conclusion

The following architecture setup is the result of researching and analyzing the previous architectures. In an effort to minimize the disadvantages of the previous architectures it is possible to design an architecture so that each of the companies can focus on what its needs are while adjusting the local index to suit this.

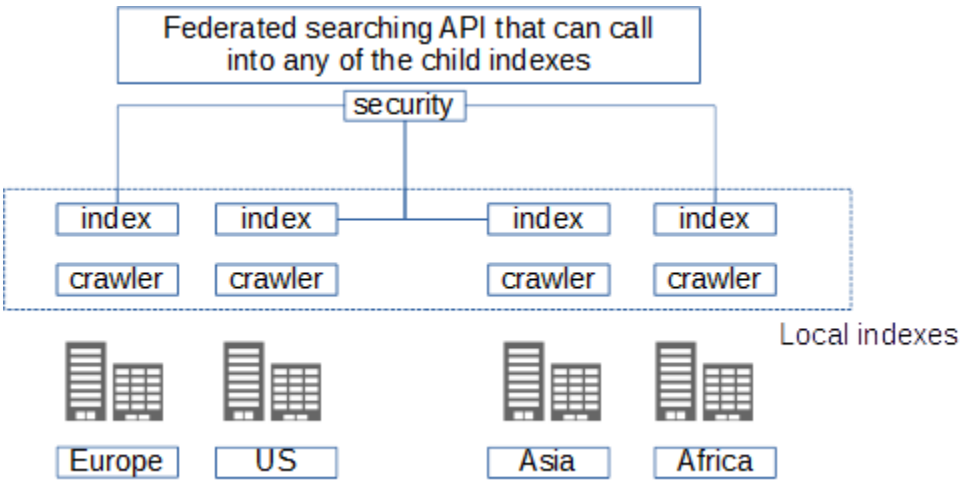


Figure 11 A federated system where local indexes are separate and combined index

This also solves the particular issues dealing with different language and character sets. Because the index is local the data access can be customized for the local crawler obeying local access rules and controlled within the boundaries of the local office network or country bound wide area networks (WANs).

The local crawler and index needs minimum bandwidth to populate the local instances because the data to be indexed is local and the index access is also local compared to previous architectures. At the same time, Redundancy is increased as searches should place local content above global (unless the search indicates otherwise). This means that is local content fails to match then the global index would be access thereby minimizing bandwidth requirement. This backup also mean if the global index goes down then the local one would still be available to serve what could be most of the search requests.

This architecture also poses some disadvantages such as the higher maintenance over head to keep what effectively two indexes is operating. The local one would need local domain knowledge to keep it running effectively while the global one would need interfacing to all the child indexes to operate correctly.

The design of the interface between the global index and the child index can be challenging if the children indexes differ too much in its content information and met-data schemas. To solve this problem the global index would need to be at a less granular level than the children indexes (for example storing more abstract descriptions about location i.e. Country/Department vs local ones which would have this information and the subunit within the department). This means that the child indexes would inherit the global index.

The bandwidth to submit the local child indexes to the global index still need to happen and is not saved completely.

2.2.3 Indexing data from multiple systems

In many ways a local environment can be thought of to be very similar to the location problem. Instead of continents or location the problem arises with multiple system. Within an organization each of the system typically outputs different content i.e. a business portal and Content

Management System (CMS) typically puts out Hyper Text Markup Language (HTML) content while an accounting piece of software in sales would save data/records via a relational database. Most of the software contains a self-provided search but this might not be true in all cases. To aggregate this information an efficient and effective design would be required.

2.2.3.1 Individual systems with own searches

Figure 12 Individual separate systems that offers limited search capability

The problem of using software out of the box with the built in search system is more evident in larger enterprises where the system has many user than smaller single user desktop applications (i.e. Photoshop) (Chernov, et al., 2006). In the above figure, three systems are presented where the first writes and stores information in a relational database (or BLOB fields for large text), the second to files (such as HTML) and the third to files but no built in search capability.

Different system contains and stores different meta-data (McHugh, et al., 1998). This means that for example if searching by a user who created a particular document it could be in a field called author or a field called created by. This poses a problem because there is no guarantee there is uniform convention for the system taxonomy (categorizing).

The third system do not have a search function therefore looking for information in this system is either manual or by user knowledge, making data inaccessible by new users and wasting domain knowledge.

Where the number of system grow tracking down information would turn into a time consuming affair where multiple system search is needed manually by a user to track down information or to collect information spread across the different applications (Ranjan, 2009).

Incomplete information is sometime more destructive than complete information due to the fact that business decision can be potential made on incomplete information giving the decision maker a false sense of understanding (or lack of the complete picture). Having disjoint search and index architecture can further exaggerate this problem.

2.2.3.2 Single index for all data held

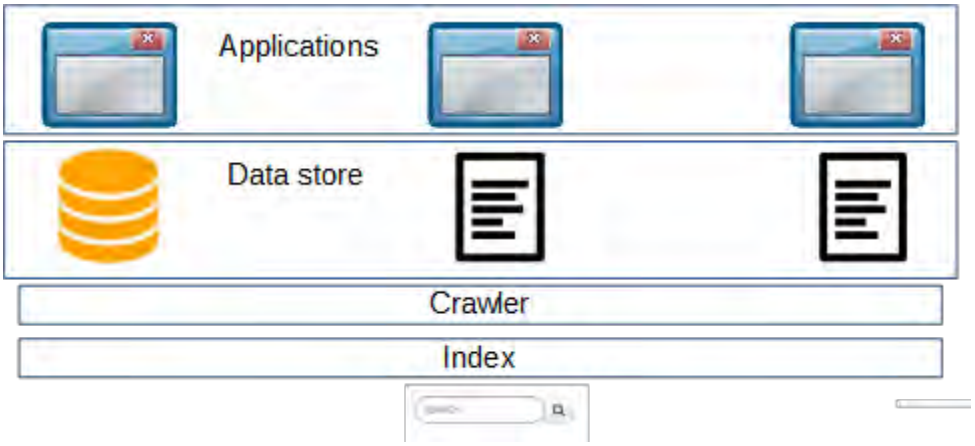


Figure 13 A common crawler for all the data stores within one location.

The easiest way to have a uniform index for all the data store is to crawl all the data stores and push the indexed information into a single index (Regli, 2010). While simplistic looking it poses its own challenges such as the different data from different sources need to be mapped to a single schema. This mean solving the problem aforementioned where the author and created by are technically a single meta-data destination type so it should be mapped to a single field in the index schema i.e. “creator”.

The crawler in this setup will need to be highly adaptable in crawling different protocols. It will need to crawl via different target systems i.e. the local intranet and public internet. With each new system the crawler and indexer need to be updated to accept new crawl sources and schema changes thereby introducing downtime to the system.

2.2.3.3 Conclusion

Having understood the above issues and the understanding gained from the multi-location index challenges it is possible to design an architecture that can facilitate both easy of expansion and the dynamics required to effectively index different data sources.

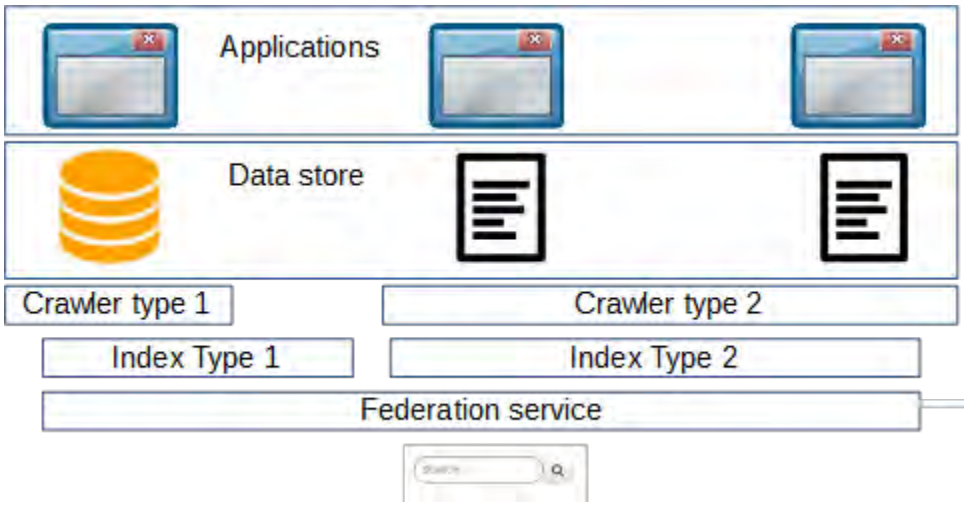


Figure 14 A federated service serving multiple child indexes and index types

With an introduction of federation to the index service this effectively allows the flexibility of using multiple crawler and indexes to suit the data being crawls and indexed. In the above example the crawler for the database based data sources can be crawled by a custom crawler especially created for that purpose where the other HTML crawlers can be the same as they share the data type. Once this is setup it would be much easier to add in additional crawlers, indexes and data sources for any number of new application. The main work would be to interface the index to the federation point similar to the interface between the global indexes to the child indexes.

2.2.4 Information volume and the speed of change

The volume of data that is subject to change also varies from location to location. For example Google' Googlebot (the indexing system that google uses for indexing the internet) accesses and processes over 20 Petabytes of data per day (Nancy Blachman, 2014) (Kennedy, 2014).

Naturally when the information is as big as the whole internet it is not feasible (yet) for google to keep pace at the speed that the content on the internet is published. It simply cannot detect changes and update of all the people blogging and publishing at real time so the Googlebot kicks into action not based on the speed of change but to its own programmed routine.

Below is an analysis of the two ways which the Solr core (or any other index) can be updated. It essentially divides up into waiting for a crawler to detect the changes or some system API directly calling the index to make an update. Akin to the object relational mapping terminology where a database is either updated instantly or in batch the names Lazy and Eager will be used to describe the updates. The sources of this concept is from the Apache Solr API (Apache Software Foundation, 2015).

2.2.4.1 Indexing strategy

Typically index is updated on a scheduled basis similar to when a crawler is setup to start and finish. Using this approach the information would be only as old as the times between crawls which would be fine for relatively static data but not for fast changing high volume data where the index would be out of data much quicker. For information that is primarily read only i.e. non transactional data lazy indexing would work quiet well. For a company of average size with an efficient crawler the crawler can be setup to run twice or three times per day. Like most companies where the information can be both slow changing and fast changing then a combined use of lazy indexing and eager indexing should be used.

The alternative for fast changing or critical up to date data it is possible to index at real time. Typically this involves the application collecting and storing the data to publish any and all changes to an indexer thereby negating the use of a crawler. If a system does to its own internal index already then rather than crawling this index for federation it should be customized if possible to keep the real time indexes available to the federation service.

Figure 15 eager indexing

2.2.4.2 Application calls in eager indexing

Within the application there would be the need to containing an Application programming interface (API) to the indexing service, this will serve as the API for eager application calls (Apache Software Foundation, 2015). For example for the Java language Solr indexing framework provides the SolrJ API to manage the SOLR index in real time (for full details refer to the later technical section). SolrJ allows the application to directly call into the index and make adjustments (Apache Software Foundation, 2015). This effectively allows the application to embed the indexing logic into the business logic of the application. For example a process might look like:

- User wants to create new user
- User is brought to the create user screen
- User inputs all the information related to the new user
- The system saves the user to the database
- On success the system triggers a call to the index and pass the just-saved information to the index to be indexed before return to the user to say the creation was successful

With this approach the index would be kept up to date for any critical information that the implementation deem necessary to be updated in real time. It can also leave out any business logic that would not require real time updating.

2.2.4.3 Conclusion

In the scope of this thesis only lazy indexing will be applied, as the crawler will only index existing information and will not be able to apply listeners on real time web changes. The Eager

aspect would only apply if the content being indexed is typically controlled by the indexing entity or that a full time crawler is started in rapid cycles.

2.2.5 Security

The notion of security in the indexing sense applies to two distinctly separate processes. The first is the ability for the indexing system to access files or data that is not open to all and has security applied to its access. The second is the process where the indexed data is bound to the security as defined when it is being indexed.

During the design phase a common question that needs to be answered is whether the indexing system should index all data with security meta-data or be restricted to index only data that are not top secret/subject to restrict access. During the research it is shown that indexing all data with subsequent query stage security verification in place to be more common (Singh, et al., 2007) than limiting indexed datasets.

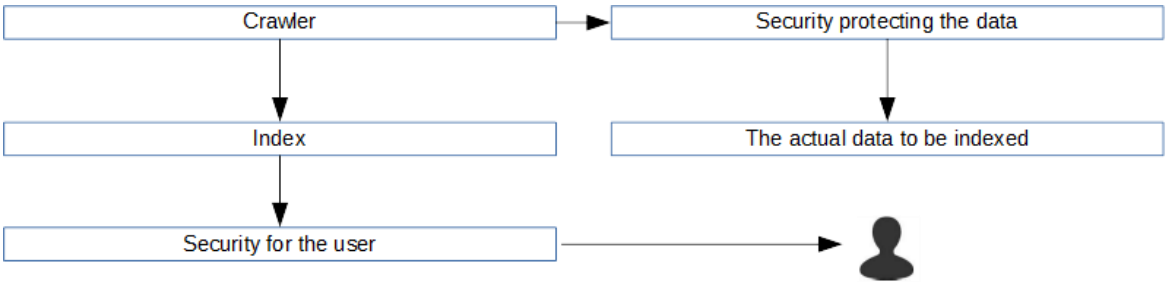


Figure 16 The two security stages in crawling and searching

2.2.5.1 Crawling secured content

During a crawling process, the crawler uses a wide range of protocols that can access information stored in different systems and locations such as HTTP, FTP Samba etc. (Hawking, 2004). According to Singh the typical enterprise stores around 85% of information in flat file structure meaning that the databases used in non IT domain companies store only 15% of the company's knowledge (Singh, et al., 2007). To access files on a file system which generally are either Unix/Linux based poses a challenge in itself due to the need for root access and the

different read, write permissions associated with the maze like structure of a typical enterprise (Singh, et al., 2007).

Depending on the type of crawler used there are several approaches that can be taken to allow the crawler to effectively access the resource to be indexed. The first and most simple approach is to give it root access (to mark the crawling process as a super user process) thereby giving it the rights to access all areas of a system.

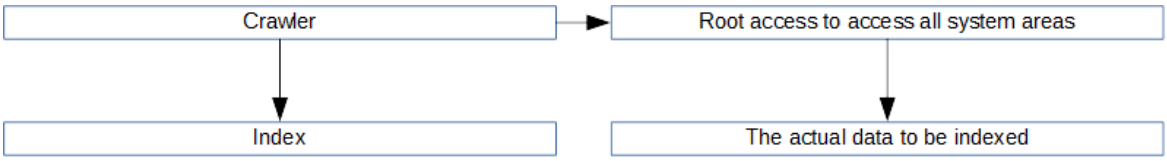


Figure 17 Crawler with root access

If the root access is too dangerous or that indeed there is data exclusion in place where certain areas of the file or data store is off limit to the indexer then granting root would be a security risk. In such cases multiple threads of the crawler can be forked to crawl local directory with individual required permission. The major drawback to this is the maintenance overhead of keeping a record of each of the crawler's setup for different parts of the file and data stores.

Figure 18 Multiple crawlers with own access permissions

2.2.5.2 Security meta-data generation

To effectively manage the security credentials for each of the indexed data resources, security meta-data will need to be applied to the content itself (Singh, et al., 2007). This would be composed of attributes such as:

- The file attributes provided by the file system like creator, dates, access group and the read, write and executable permissions.

- The file location or URL should also be able to compliment the security policies. For example the crawler setup for crawling the /secured/top_secret/ folder should automatically inject meta-data to make the indexed data as a top secret resource.
- Where an independent system is used to keep track of permission there might need to be custom hooks required to get the crawler to do a lookup using the resource location to get the correct security meta-data to be applied to the resource.

Once the security meta-data is in place accessing the indexed data would effectively mean matching the user provided security credential to the meta-data and restrict what the user do not have permission to view or access.

2.2.5.3 Accessing indexed data with security applied

On the searching end of the process, access control applies to users who might have different credentials to see a different representation of a restricted component of information or may be denied it completely (Hawking, 2004). Under such situation two separate system is often employed handle the task of doing the indexing and filter the results with the aid of a permission system for example LDAP, Active Directory etc. (Ranjan, 2009).

As mentioned in the previous part data that is indexed effectively should contain enough security meta-data to effectively manage the security policies necessary. Working in conjunction with a security policy system such as LDAP this meta-data would be matched off to the corresponding LDAP attributes so that the searching service would be able to know if a result should be shown to the user or not.

2.2.5.4 Access Control Aware Search (ACAS)

In the paper “Efficient and secure search of enterprise file system” an analysis is presented in relation to the efficiency and security of the systems that uses two separate components i.e. the indexed data with security meta-data and the security system versus a notion called Access Control Aware Search (ACAS) (Singh, et al., 2007). To understand their argument ACAS needs a brief explanation.

Up to his point the security model has been a single index with security meta-data working with a security access system to determine what the user has privilege to see in the search result. With ACAS the initial indexing phase bears the load of establishing security to user as its uniqueness is that it operates by creating multiple indexes **per use case**.

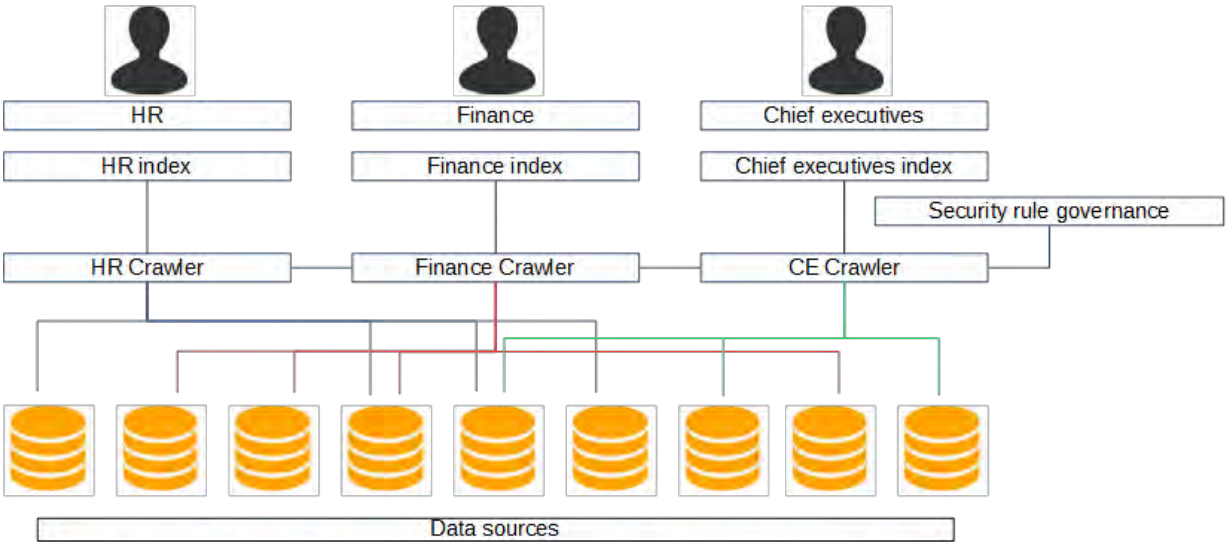


Figure 19 The crawling and indexing phase of ACAS

During the ACAS indexing phase the crawlers are programmed to only access the data sources which it is told to index. This is setup as per use case scenario such as (in the above example) a view for HR, Finance and Chief Executive. If the CEO is to search for data he/she might only see the chief executive index in the above example whereas if a Chief Finance Office was to do the same he/she would be able to see both the Finance index and the Chief executive index. This notion of mixing and matching applicable indexes depending on the user's credential is a key attribute of ACAS.

The two main advantage present in the paper for ACAS is that it hold more robust security than post index security filtering and the speed of not having to do post security filtering (Singh, et al., 2007). The disadvantages or the above system can be seen as:

- High Possibility of duplication of indexed data due to the way data is indexed which means post searching logic will need to negate this effect.

- Higher storage requirement due to duplication of data within index. If the concept is taken to the extreme and each user has his/her own index then the index could grow exponentially when new data is added/indexed.
- Heavy processing requirement at the crawling phase due to far larger number of crawlers and the need for supporting security services.
- Higher maintenance overhead due to the number of crawlers and the security rules setup for each of the crawlers. For example one schema change would require all crawlers and indexes to change.

2.2.5.5 Conclusion

The scope of this experiment will focus on the crawling of selected internet pages so there will be no metadata applicable during the experimentation phase. Having understood the security implications and apparatuses around the pre and post security processing mentioned above it would not be complicated with security Meta data to apply a second security layer on top of the experiment search services in order to control access control. For example meta-data of work groups can be provided to the indexing service to match a permission array of work group eligible to access within the index meta-data. This would allow only users that have the correct working group to retrieve data for them.

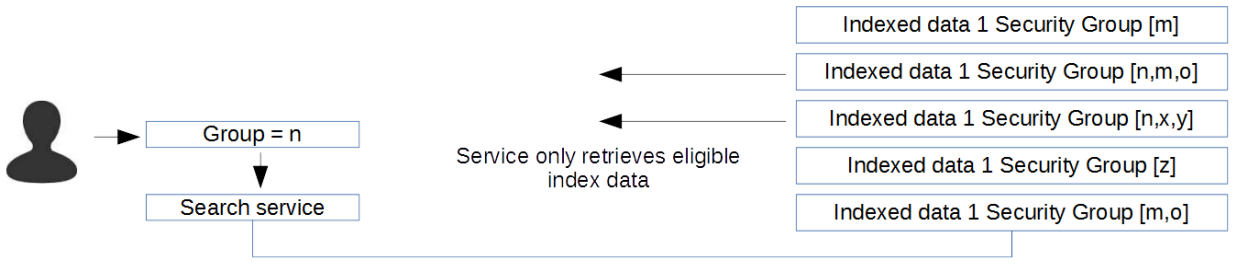


Figure 20 Intelligent service with Security

2.3 Understanding the data

Different business nature of organizations and enterprises means that the information generated and stored can vary in type and content. This also means that the crawling ability of an effective indexing system needs to include the capability to read different formats of data including

proprietary types such as pdfs, word documents and semi binary data formats (Excel files) (Hawking, 2004). While most administrative and information services store primarily raw text data, companies that deals with large quantities of magazine, art and photographic images would require an alternative way to index binary data. To index these data two methods are employed for binary and binary textual data.

2.3.1 Binary data

Binary data are the likes of images, photographs and executables. They have no meaningful textual content within the file itself that can be identified without the aid of either human intervention or some form of intelligence such as image detection AI or analysis (Chernov, et al., 2006). As such the primary piece of information comes from the file attributes itself. The filename, creator and modification attributes are the first things to be indexed. Outside of these the meta-data would have to be added by the crawler based on predefined human rules. File location and URL can also complement in adding the meta-data of the content.

2.3.2 Binary textual data

Binary data that contain text such as adobe acrobat's PDF, Microsoft's WORD and OpenOffice's ODT files. Such files contains textual information but also its own set of proprietary data for rendering purposes and as such cannot be directly opened for viewing by traditional text editors such as VIM, VI and NotePad. A slight variation of these files are files that contains a mixture of textual and non-textual data such as Microsoft's excel files. XLS files can contain numbers, formulas, commands, images and text which causes indexing these files more challenging than normal text files.

In order to index binary textual data the concept of connectors are introduced for crawlers (Apache Software Foundation, 2015) (Apache Foundation, 2014). Connectors are effectively key algorithms for the crawler to open certain file formats in order to then read the text inside (very much like how users open PDFs with Adobe or any other PDF reader to read the actual text content in the file).

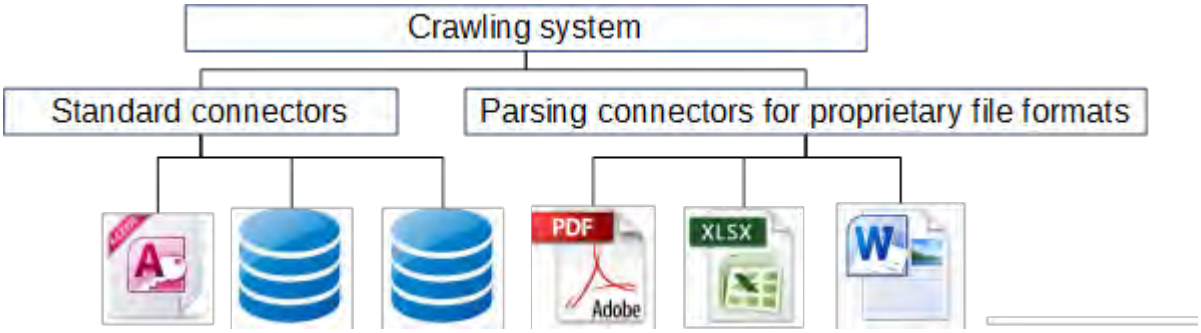


Figure 21 Crawler with non-standard connectors for parsing

One of the most widely used connectors for parsing binary text data for the Java programing language include the apache Tika framework (Apache Foundation, 2014) which supports a wide range of file formats including

- Hyper Text Markup Language
- XML and its derived formats
- Microsoft Office documents
- Portable Document format (PDF)
- Rich Text format (RTF)
- Various Audio and multimedia formats

Source: (Apache Foundation, 2014)

2.3.3 Dynamic content in internet 2.0

Most of the earlier internet websites contains static XML representing the HTML which can be parsed easily by crawlers. With the introduction of internet 2.0 where rich applications runs within the HTML5 web browsers this is no longer the case. Web sites such as Gmail, Facebook and LinkedIn meant that there is now as much dynamic content on the internet as there is of static content (Duda, et al., 2008).

The core attributes of a dynamic content is that the information presented to the UI is no longer confined to the HTML content. HTML5 uses the JavaScript language heavily which means that

the webpage or the HTML representation of a webpage may be no more than just a simple container for the dynamic content retrieved and presented by the corresponding JavaScript.

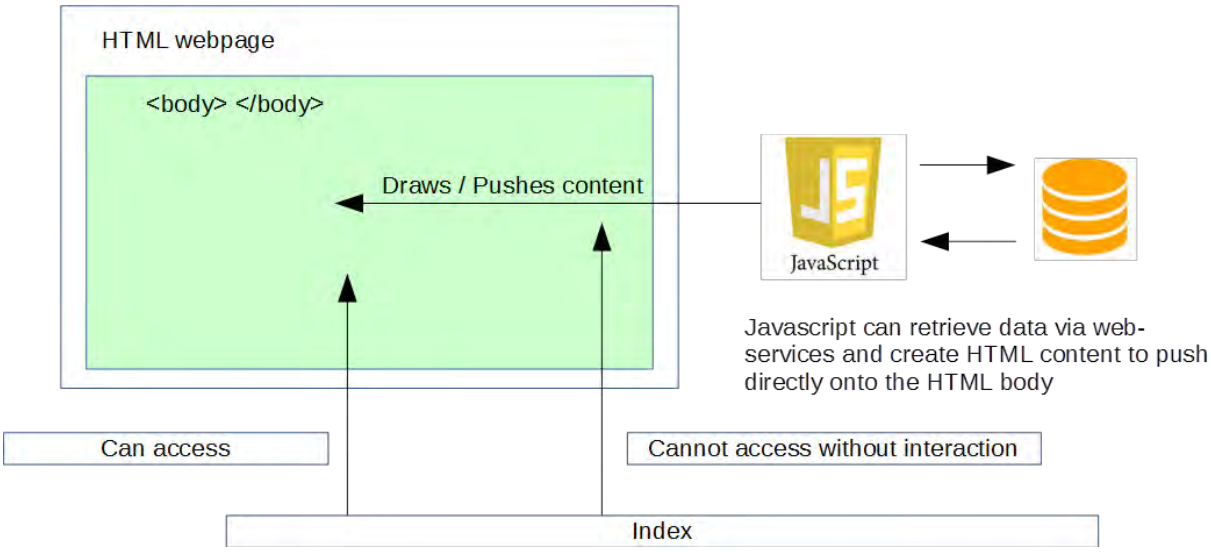


Figure 22 Dynamic web content

The author identifies the interaction between the user and the web application as a defining number of states. The information content would be different in the states in and after a transition. In order to index the content presented at different states it is important to improve the crawler so that it can trigger transition in a controlled way (Duda, et al., 2008).

In effect this would mean that the crawler would need to be aware of the interaction requirements of the web site/application so that it would be able to interact with it via something like a script and the sites' states by identifying the HTML components on screen at each and every state.

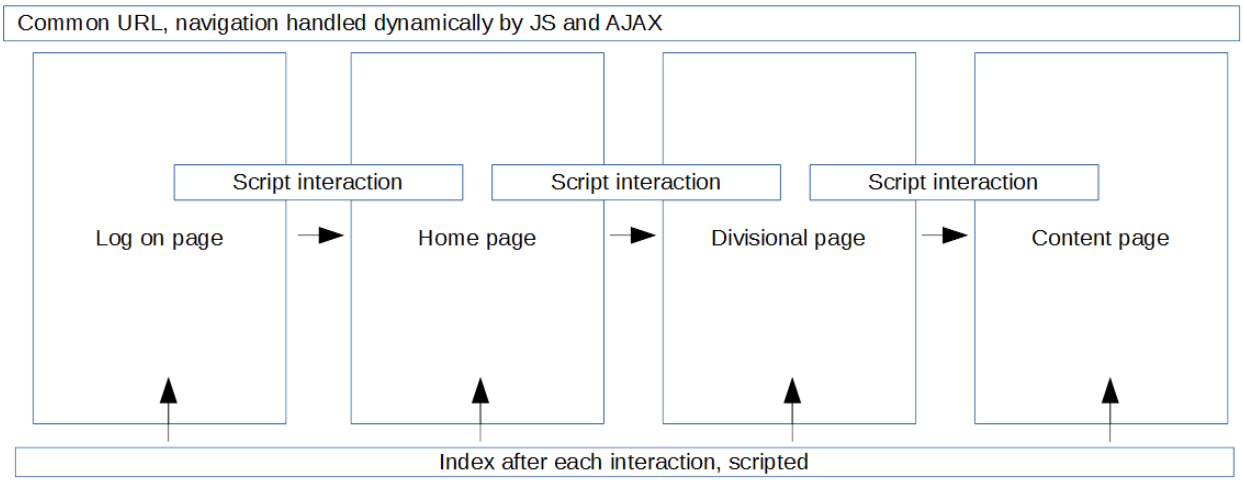


Figure 23 Indexing dynamic content

The concluding idea of indexing data in this manner is if the source information is dynamic and that the states and transitions are known then it would be possible to script this process albeit requiring a large amount of manual human intervention at the start. For source information that is for example the internet where the states and transitions cannot be known beforehand then this process fails. For example to index a site that needs user log in it is simply not possible without first registering a username and password to log into the site if it is a dynamic site. For this reason it would be impossible for a web crawler to index Gmail content without knowing the user's credentials.

2.3.4 Automating access to Content repositories

Content libraries and repositories of companies are typically managed by some form content management software such as FileNet P8 (IBM), Documentum (EMC), LiveLink (OpenText), Meridio (Autonomy), Windows shares (Microsoft), and SharePoint (Microsoft) (Apache Software Foundation, 2015). In such cases the content management teams within the company would use these tools to aid in the deployment of company content and as such a lot of time and effort in managing the data goes into such repository management system. From a lot of perspectives this process is not much different than the creation of an index in that the user will have to tag content with keywords for semantic association and security layers for controlling access. Rather having such information wasted during the indexing phase a second form of

connector would ideally be available for crawlers to query such file repositories and extract the associated meta-data while indexing the content in these repositories.

The Apache manifold CF (Connector Framework) project is one such open source Java connector that allows the crawler to query not only repository content but the repository itself for content meta-data to aid in the indexing of repository content (Apache Software Foundation, 2015).

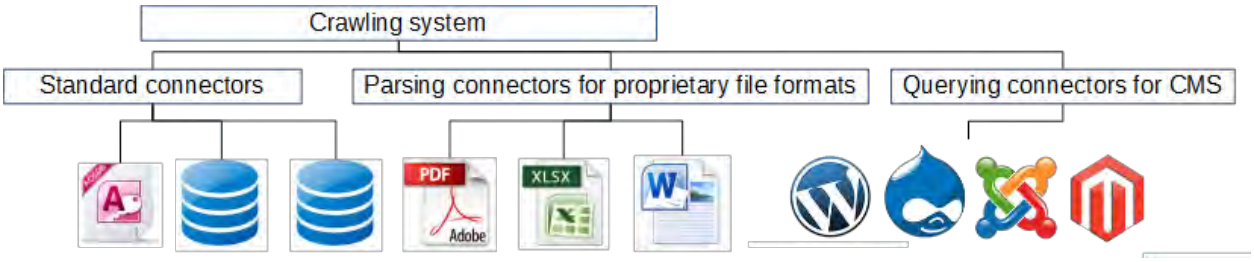


Figure 24 Advanced crawler with various connectors

- To present a uniform view to content that may be very different in structure. To achieve this the use of Meta data from different sources must be applied and the stored in a common structure to these that may already have such information available (Hawking, 2004).
- Advanced system will allow the user to customize the search and result views to their own preference (Hawking, 2004).

There is a well know publically available e-mail demonstrating the frustration that users would often experience when the indexing system misses out on the importance of linking simple term to what the user wants to search. In this case the user happens to be Bill Gates of Microsoft and the term he was searching for was “Movie” in reference to Microsoft’s movie maker. In his own words “I tried scoping to Media stuff. Still no moviemaker. I typed in movie. Nothing. I typed in movie maker. Nothing.” After the frustrated Mr Gates contacted the people in charge of maintaining the search engine “They told me to go to the main page search button and type movie maker (not moviemaker!).” (Bishop, 2008)

Two of the most importing attributes when users evaluates the search results is the relevance of the information returned and the importance associated with the information. These are also

some of the hardest challenges to solve due to the contextual nature of the person who is doing the search. Similar keywords would mean different things to different people. The ability and challenge therefore is for the indexing system to be smart enough in the meta data association both from the index itself and the user identification attributes to know what are the most likely results that matches the user's search term (Hawking, 2004).

2.3.5 Information relevance

Information relevance can really be only thrived from a good meta-data setup during the search phase (Ding, et al., 2012). The more meta-data available to the actual indexed content the better the chances for the indexer to return the correct search results. For the successful application of good meta-data, data from the search source is also critical. This information may not only be the search term that the user has inputted but also any other attribute the system is aware at search time that may aid in the search matching the meta-data. For example if a user is on the Revenue's website and is looking for information on capital gains tax then the user will enter the term into the search field and expected results related to capital gains tax. If however the user is on the forms section of the website after navigating there from the homepage then the search would be expected to put the forms for capital gains tax in a higher ranking. This should be achieved by the website's search field being aware which page that the user is on and not having to rely on the user to add "forms" to the search term manually.

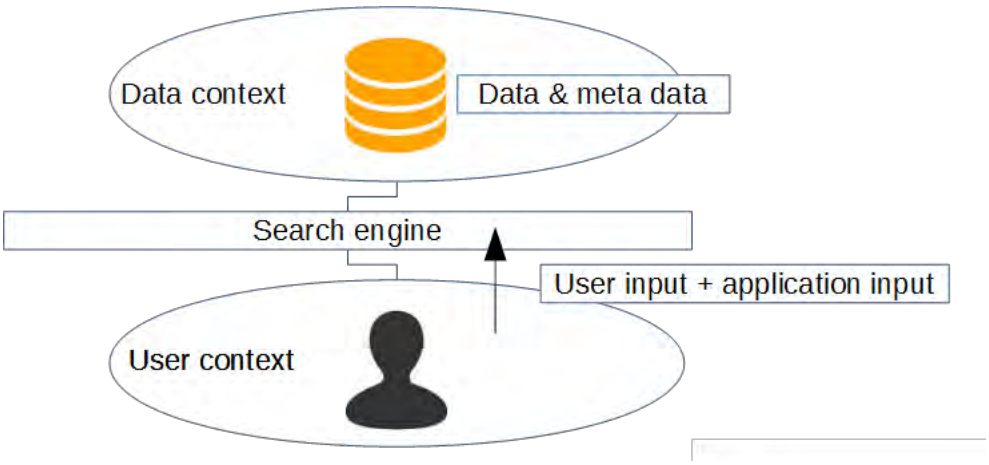


Figure 25 Understanding user and data contexts

2.3.6 Information importance

Information importance is the notion that while the data might not be the most relevant it could be the most important. In the area of law and legislation this can be particular applicable (Ding, et al., 2012).

A good example of this is data sets that has time as its attribute. Series data is especially prone to the need for this kind of meta-data. For example if there is a database of economic data broken up into several classifications. The tendency will be over time this dataset will grow and each one will be marked as belonging to a certain time frame. If the user is to search for data on this economic data then the timestamp on each of the findings will have special importance as data from 20 years ago would not be as relevant as data from the current fiscal year.

Data source with importance weightings can be created in two distinct ways. One is to embed meta-data into the index directly to mark its importance, the problem with this is that as the importance flags grow it becomes increasingly hard to manage all the tags and to work out just how important relatively two pieces of content is. Some form of mathematical formula will be needed to achieve this. The second way is to keep a separate list of meta-data tags against a weighting table. When a term is found/matched its meta-data will be split and summed up according to their weights from the weight table (if the meta-data term is not present it gets a 0 weight i.e. not contributing to the weight).

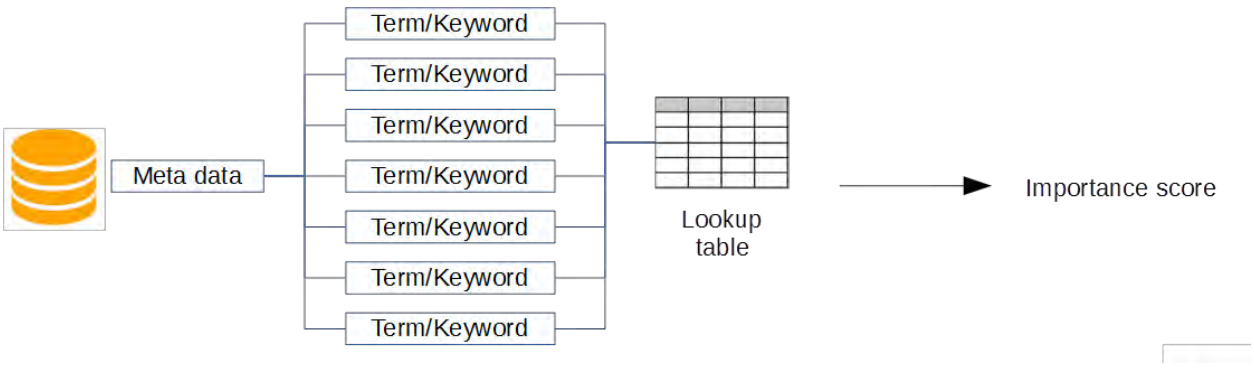


Figure 26 How to understand and weight keywords

2.3.7 Conclusion

The candidate of indexing for the experiment is the website Wikipedia. Image files will be present in the URL content and will be indexed as a reference to i.e. the actual image binary will not be saved but the URL to the images including the file type (image in this case) will be index aware and searchable in the federated searching service.

Wikipedia is not web 2.0 enabled it uses minimum JavaScript interaction unless in edit mode so JavaScript interaction between the crawler and Wikipedia is not required. CMS connectors are also not required in the experiment setup.

Nutch by default support the configuration of business taxonomy by allowing the configuration of synonyms and key phrasing word boosting. Both of these features will not be activated in the Solr indexer as they do not represent a default index setup.

2.4 Design and scaling of crawling units

Depending on the size of the source data that needs indexing and the time available for indexing an enterprise would have to decide on the amount of resource invested into indexing its data stores. For limited data sets a single server with multiple crawlers would be sufficient if they divide up the data sources up between them in a managed way (Hafri & Djeraba, 2004). In this approach the crawlers will need to be started by a pre-defined process and pushes the indexed data into a single targeted index. If the enterprise is sizeable and/or the time available is not long enough for multiple crawler to crawl the data completely, then a structured managed approach is needed. A possible solution to this scaling would be a URL parser and URL processor pattern where the parser are the units that manages the URLs (and children URLs) and the processor reads the resource at a URL given to it by the parser for pure indexing functions.

In the real implementation the units should be smart enough to be both as required. From the onset a single URL parser would be created to parse URLs within the first parent page exclusively. On completion a new URL list will be generated where the role of the parsing will fall on new parsers and the actual reading of the content data into indexed data will be performed by the processors.

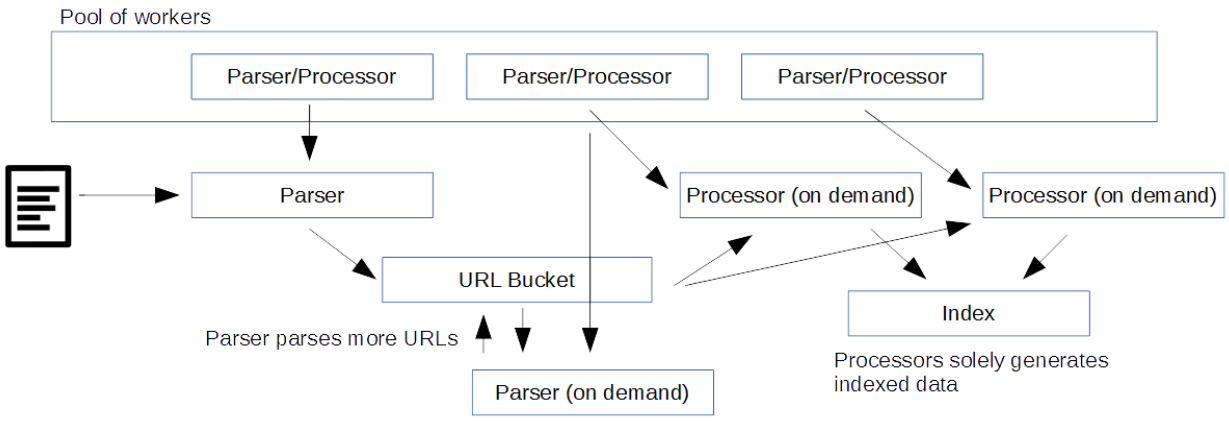


Figure 27 Producer consumer model for scaling crawlers

2.4.1 Conclusion

From the initial conception of the experimental architecture the core challenge of a federated searching service is the ability to add crawlers to push into new indexes as easy as possible. And to allow the search service can connect to the new indexes as easy as possible.

The handling and scaling of the crawler and its parser children is fully configurable within the Nutch instances. A range of numerical values will be tried to achieve optimal number of parsers. The key is to create enough so that no parsing will be held up while not having too many idle parsers.

2.5 Accessing the index and advanced features

Search engines came in prominence around the time that people started to use the internet. Back before the search engine and browser wars there was many various search engines and no one in particular stood out the way they do today. In the early days of searching people used and often used several search engine to search for information that a single search engine might have missed and from that perspective it is still true to this day.

The most popular search engine known around the world is of course Google that just happens to be the number three most valuable brand in the world (Forbes, 2014). But google is not just a single search engine. If the content one seeks is in say Chinese then google will not return the

same results from the .hk (Hong Kong) domain as it would from say the .ie (Ireland) domain. The reason is quiet simple they operate off two different indexes and ranking algorithms (Google inc., 2015). To mirror the earlier days a user with a query in Chinese might even be tempted to bypass google when it comes to Chinese searches, instead he or she might find that Baidu (baidu.com), a Chinese based search engine would provide more relevant results and ranking again due to the index and the search algorithms. Failing that there is other alternatives such as Micorsoft's Bing and Yahoo's own search engine available to the end user.

From these examples it is clear that the index on its own is not enough for a full on user experience. The search processing algorithm also plays a pivotal role in getting the results that an end user seeks.

2.5.1 Auto complete

Autocomplete works by prompting the user for the complete word/search string that the user is *typing into the search box* ***before*** *the user has finished typing the whole word. Auto complete has the advantage of helping the user to correct misspelled words or to highlight to the user of a* more defined match to his or her search query. For example if a user intends on finding out about oxymorons, they would start the search by typing in "oxy" at which stage the google search engine would try to predict what word starting with "oxy" the user wishes to search for (Google inc., 2015).

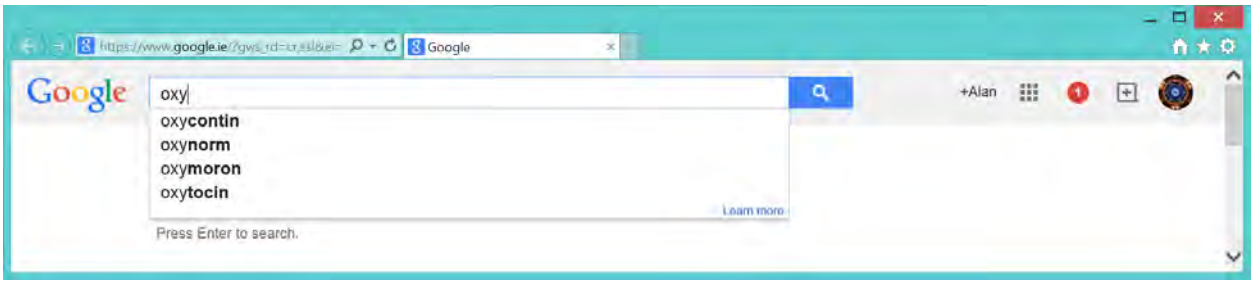


Figure 28 Search auto complete

To enable this functionality an index of keywords is generated and stored in the data index schema. The result is the matching of the characters of the search phrase and the significance of each of the keywords/phrase. The initial match must be character for character but the ordering

of all the phrase matching the characters would be derived from the combined importance rating of each of the phrases. This can be the weighting of the words themselves or a combination of other attributes like information source and the meta-data associations with the phrases.

2.5.2 Auto suggestion

Similar to auto completion of search words this concept works at the word phrasing level.

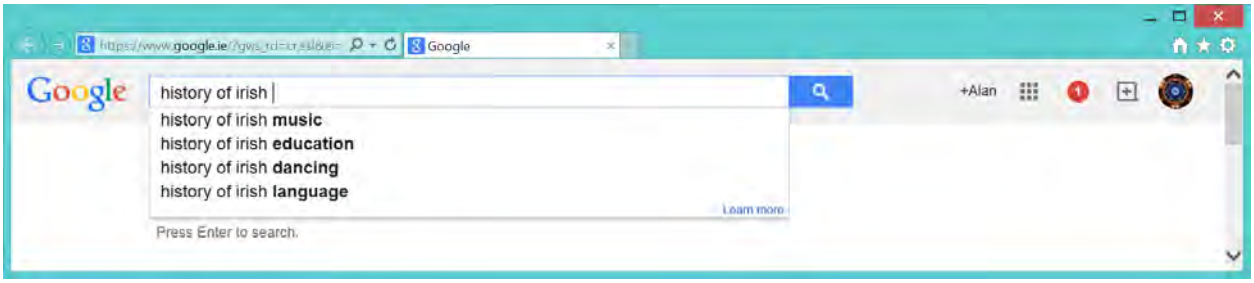


Figure 29 Search auto suggestion

To enable auto suggestion, the index would need a phrase map created from the source indexed data as to what words tend to appear before and after a certain word. Once this is constructed the prediction logic would try to create a list of suggestion based on the most commonly encountered before and after words and walk through them to create the next link.

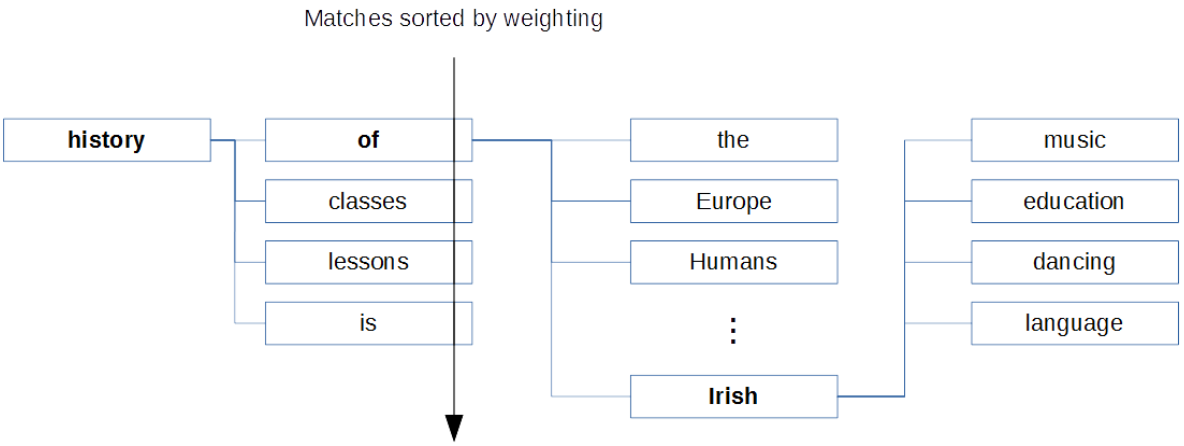


Figure 30 How auto suggestion works in Solr

To implement the above the map would be a many to one and one to many word link. Once a word is inputted the logic would attempt to retrieve the words that appears after the inputted

word and show it based on weighting. Complex system would also take into consideration the full phrase and not just the node at each stages of user input.

2.5.3 History tracking

History tracking would be to store the terms that the user have previously searched for and if the input pattern matches the character ordering of the previous search then the previous searched term would become first suggestion (Google inc., 2015).

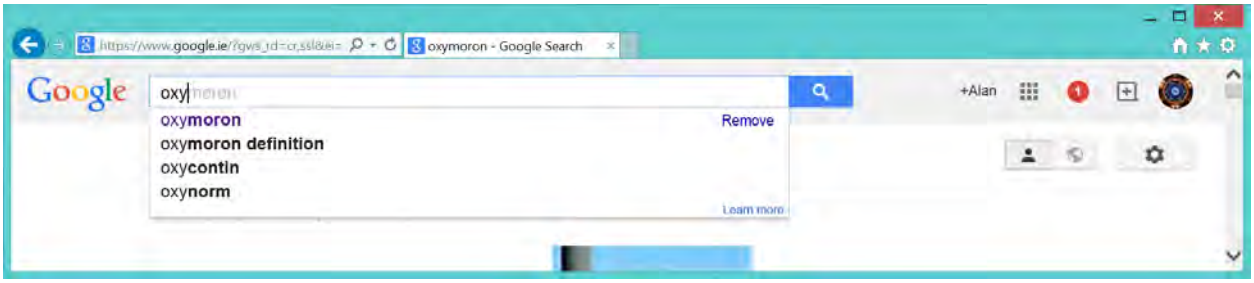


Figure 31 History tracking

Google shows this behavior by highlighting the previous searched term in purple.

2.5.4 Adaptive learning

Adaptive learning is the ability for an index and search system to customize the search prediction engine to a specific user (Palma, et al., 2012). The prerequisite for this is that the search engine must be aware of who the user is. In Google's example the user is known by the logged in credentials proved to google. Without this google would not be able to implement adaptive learning as it has no way of know who is searching and when a user switch has occurred.

To implement adaptive learning the searching behavior and history of users would need to be stored in a database or an additional index system. This data set would then be complimented into the search engine during the result matching and ordering phases of the search.

Attributes that are tracked to customize a search experience would be:

- Who the user is

- What the user normally search for, is there any category or classification preferences
- What kind of file formats the user normally search for, is it forms, papers or images
- The times searches take place, do the user search for different things at different times for example during the workday and after work

2.5.5 Conclusion

Studying the Solr API it can be seen that it supports out of the box auto completion and auto suggestion via child Restful services. This means that when a user inputs a search string it will be sent off to a Solr service URL which will get all the auto completion and alternative suggestions so that it may be presented to the user via a HTML5 interface.

The Database table index unfortunately do not have this functionality out of the box (as it is a simple database table representing an index). As such the scope of the experiment will not include a HTML5 drop down component to allow the user to select a auto referenced term.

3 FRAMEWORK PROPOSAL AND DESIGN

3.1 Introduction

The primary objective of the experiment is to construct a federated indexing solution highlighting the ways to solve existing enterprise challenges and to compare file based indexing system to a relational database indexing system.

After the initial research, design, prototype and implementation of the overall software stack the second comparison criteria will fall under two main areas; the raw performance provided by the two indexes and the ease of use to integrate them into a solution for what would be a federated enterprise setup.

From researching the papers into federated indexing for enterprises some common themes can be seen in the structure of the software stacks that are normally used to provide federated searching services in enterprise situations. Due to the different language nature of the various services (.NET, JAVA, JavaScript) a heterogeneous technology should be used as the boundary interface to whatever technology might be needed to connect to the indexing service. This will ensure no

single technology will hinder the interaction between other technologies to the indexing service. The most common technology for this is Restful web services.

A key technology in open source indexing is the apache SOLR family of products. SOLR itself provide a relatively advanced Restful interface to both pushing and extracting data out of its file based indexes also known as cores. But to make the boundary interface common to all outside services (i.e. external callers should not have to know if it is hitting SOLR or some other technology) the interface should conform to only one set of specification. In order to do a custom made Restful API will be wrapping both indexing solutions.

The ease of integration to a custom Restful service stack for both indexes will be one of the key measurement. It will signify the time and other resources an enterprise would potentially need to deploy one of the two indexes and the challenges in working with what could be an existing language.

Running along the evaluation of the development efforts to integrate will be two software sub stacks to representing the file based indexes and the relational database based indexes. While the specific of the two core stacks will be different, an effort to make sure that as much of common components will be used in both.

3.2 System Architecture

Before the analysis of the individual software framework used, this section explains where each of the framework fits in the bigger architecture and its corresponding role. At the highest level contains the components can be split into the following:

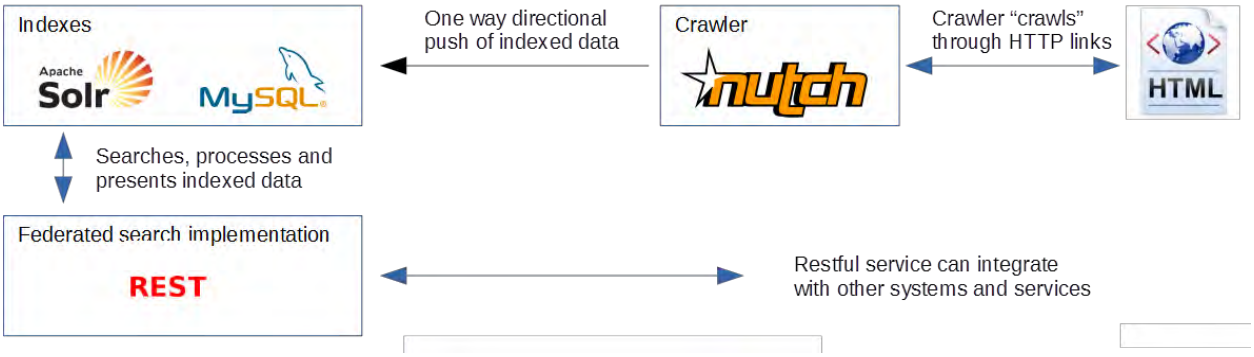


Figure 32 Proposed experimental architecture

The **index** is where data and meta-data is stored. This contains a superset of the data being indexed. In this experiment two indexes are used, Apache SOLR represents the file based index and MySQL represents the Database index.

The **Crawler** in this experiment is the framework that is responsible for seeking out HTML content and fetch the content using HTTP. It is also capable of detecting URLs and accessing the related HTML much like a tree transversal algorithm. The crawler then parses the data into an associated schema and pushes it into an index conforming to the index interface (written in Java)

The **Federated search implementation** is a custom developed software stack written using Java and Java based frameworks that will allow the access to the indexes. The idea would be using custom connectors any form of index can be integrated into the federated service and that to the outside callee of the service the index would appear to be a single uniform index core.

The Nutch crawler version one family has built in support for pushing indexed data into the Solr Lucene based indexes via the Solr restful interfaces.

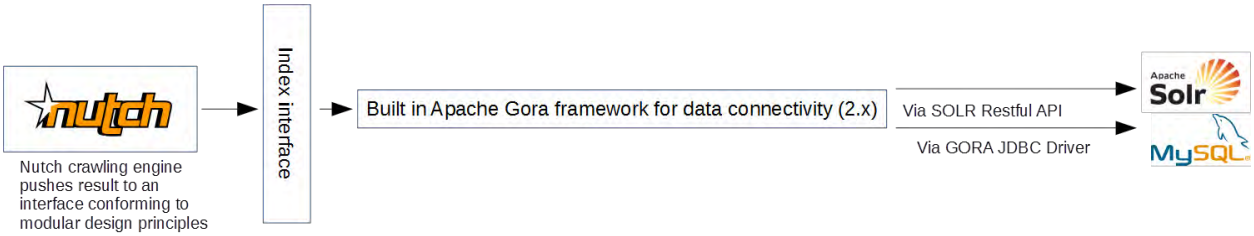


Figure 33 How nutch indexes the DB and Solr

In version two of Nutch the interface for writing the indexed data to an index was expanded to include Apache Gora framework which meant that the index can now be relational database using SQL or a NoSQL (like Cassandra) data store.

The language used for the federated search implementation will be based on Java (the same language that the majority of the Apache software libraries are based on). As such it includes many other Java frameworks for various purposes outline in the architecture diagram.

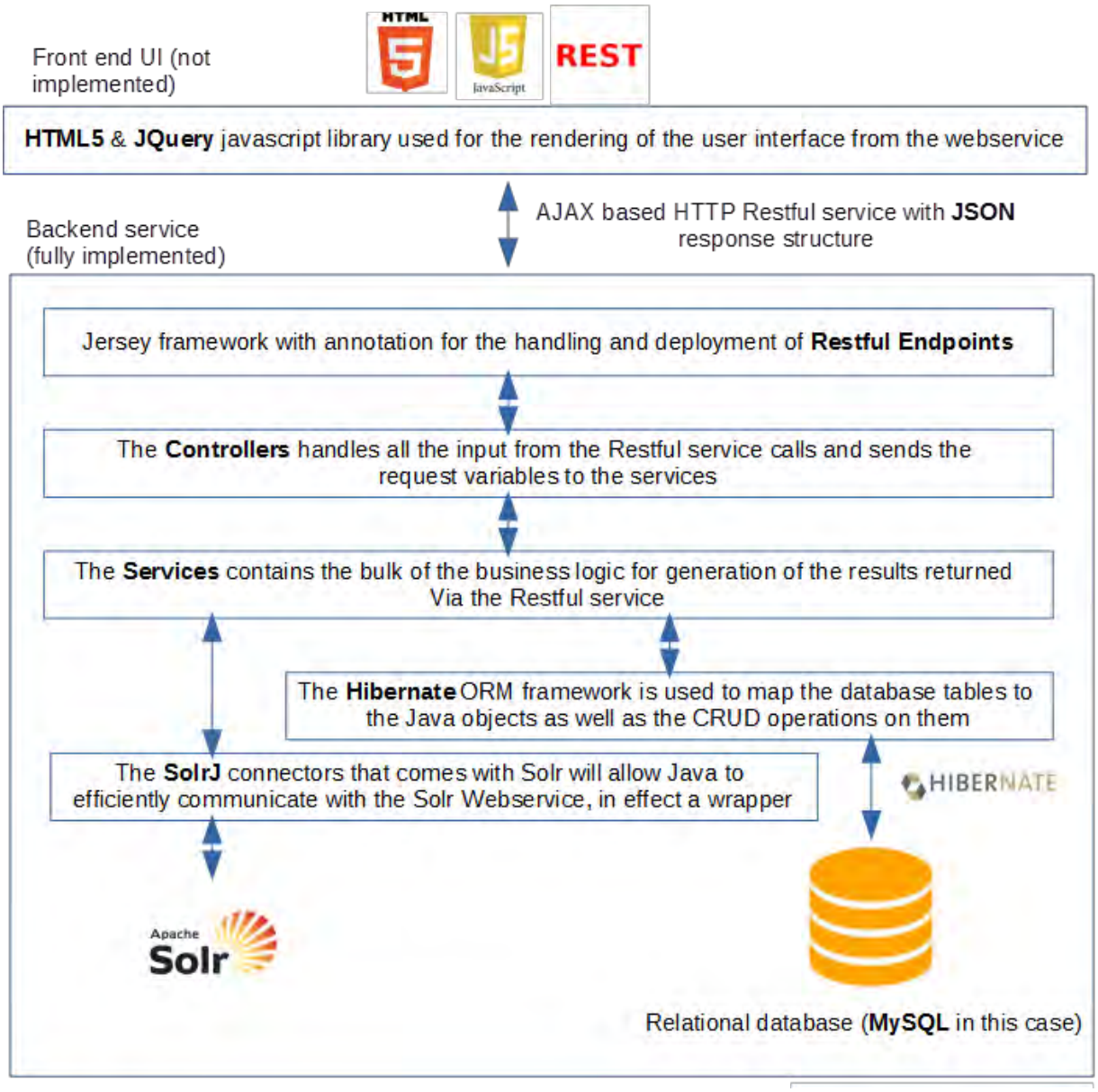


Figure 34 Full system stack

3.3 The Core Technologies

3.3.1 Apache SOLR

Apache Solr is a JAVA based file indexing system with a Restful service interface. SOLR was born out of the Apache Lucene projects and at its core the actual indexing containers came straight from the Apache Lucene. Wrapping the Lucene core is a mini web container allowing a Restful service to manipulate the data held in the indexes or cores. This with the Lucene cores makes up the apache SOLR project.

Apache SOLR 4.X family is the most recent version of SOLR where the index technology under the Restful service is built with Lucene in mind, as such it forms the actual file based indexes in one of the software stacks.

3.3.2 Apache Nutch 2

Apache Nutch is a C based open source crawler that can crawl both file systems and remote protocols such as HTTP. As such it primarily runs with the Linux bash environment and is the crawler of choice for this project. Being a member of the Apache foundation it has strong support to work with SOLR, indeed to redirect the output results of the crawler to an accessible SORL index it only requires minimal effort at the command line level once the cores are setup to accept the schema output from the crawler. Due to the Restful nature of SOLR it is possible to run the crawler anywhere that has access to the URL of SOLR to integrate both, this will be especially important as windows user would not be able to run Nutch effectively without the use of a Linux or OSX environment.

With the advent of big data and the need to work with big data Apache Nutch 2.X allows the underlying crawl database to be a NoSQL database. Common usage would be to use Cassandra or Mango DB for such jobs as they are especially suited to the unstructured nature of storing crawl results from multiple sources. With the use of the experimental Apache Gora adapters it is now possible to use a relational database to store the crawl results very much in a table structure similar to the schema designed for SOLR.

3.3.3 Apache Tomcat

Apache tomcat (Apache Software Foundation, 2015) will be the default web container for running both the new Restful service framework and the SOLR instance for the file indexer.

Apache is chosen because it is the most lightweight web container to support the full Java stack required in the federated searching service.

3.3.4 MySQL

High performing, free database for storing relational data. Supported by MySQL workbench for data verification purposes.

3.3.5 Hibernate

To effectively interface with the MySQL tables storing the Nutch crawl data and representing the relational database version of the crawl data index as Java based JPA (Java persistent API) compliant framework called Hibernate will be used. This allow the manipulation of the underlying tables at the Java language level and reduce the need for development and configuration of DAOs (Data Access Objects) and DTOs (Data Transfer Objects) in pre ORM software setups. It also allows the dynamic evolution of the database schema necessary to absorb new crawl data schemas.

3.3.6 Spring Services Framework

The spring framework is a framework for rapid application development in JAVA. Spring is used so that Restful services can be quickly deployed and without spending too much time on the scaffolding of a typical JAVA Restful service running within a web container.

3.3.7 CodeHaus Jackson Libraries

The CodeHaus Jackson libraries are the primary Marshaller and unmarshaller for the Restful service so that the HTTP responses from the REST requests can be represented in a JSON format. If necessary the responses can be adjusted into XML based.

3.3.8 SOLRJ

SOLR is represented by a Restful interface which is language neutral, as such to integrate with SOLR different languages would often have to implement the language specific version of a service consumer to interact with SOLR. Fortunately this problem has already being solved by the Apache in what are the SOLR connectors, using the right connect (in our case JAVA) it is possible to include the connector into the service project and directly interact with SOLR without having to worry about the marshalling and unmarshalling or the HTTP request/response construction to and from SOLR (Grainger & Potter, 2014).

4 EXPERIMENT SETUP

4.1 Introduction

This section describes the process of setting up the experiment so that the evaluation process can be evaluated against a working federated searching system.

The main setup components are the crawler, the indexes which are directly populated by the crawler and the Java based service handling the search retrieval from the indexes and present the matched results via a Restful interface.

4.2 The file based index setup

The file based indexing software stack has the common custom Restful interface at the top going through common custom business logic. It is just under her where the split occurs with the file based index going to SOLR via the SOLRJ connectors. The results are then interpreted by the business logic and a result is returned from the custom Restful service to the caller.

Out of the box, SOLR is a generic index. This means that it has no notion of what data it is indexing, it could be from Nutch or any other combination of data. As such each and every SOLR core must be customized for the data that each core will be storing. In this experiment we only need one core as we do not intend on storing more than one version of dataset schema, but such possibility can exist if the indexing system need to index vastly different data formats and more importantly separately for different business purposes.

Within each of the SOLR cores reside a critical file representing the schema. This file tells the Lucene indexer what information is being stored and what format the various information are in. They can be created and destroyed by changing this file which is in a XML format. Changes take effect when the SOLR instance is restarted.

The schema used in this experiment will represent the Nutch Crawl result output format. While a standard schema file is provided by the Nutch, due to the quick evolving open source nature of the project it did not work for the most up to date version of SOLR. A more in depth understanding of this file and what it represents is necessary to configure it correctly for SOLR.

While the common Restful service allows the retrieval from the SOLR cores it will not in itself provide the means to push data into the SOLR cores, for this Nutch 1.9 (the latest member of the 1.X family) will be used. Despite this it is possible to push new data in from the common Restful service but that would be outside the immediate scope of this experimental findings.

4.2.1 Installing SOLR

SOLR can be downloaded freely from https://lucene.apache.org/Solr/downloads.html. The download itself is a compressed file with all the binaries and Java classes contained within itself. Uncompressing this binary is all that is required for installation since an installer is not used by SOLR. Below shows the content of SOLR 4.1.

4.2.2 Configuring SOLR

The default out of the box SOLR contains a default core called collection1, for this evaluation this is sufficient and no additional cores are needed. If there is a need for more cores this is the level where the folders holding the indexes will reside. Within each core is the data folder holding the actual binary index files and the configuration (conf) folder for all the configuration for the specific core. The conf folder looks like this.

The key file that needs changing is the schema.xml which holds the format of the data that the Nutch crawler will be sending to SOLR for indexing. Of all the changes in this file, the important

ones are the changes from the default generic SOLR fields to the Nutch specific ones. For example within the types tag under schema sits the field declarations in the xml:

```
<schema name="nutch" version="1.5">
        <types>
                <fieldType name="string" class="Solr.StrField" sortMissingLast="true" omitNorms="true"/>
```

Each of fieldType represents a field (that can be thought of as a column in a relational DB) for storing values. The class declaration is how SOLR classes will handle/process that field along with other attributes for processing. When this is setup SOLR will load up the core with the corresponding schema ready to data entry. For references to the full capabilities and configurations for solr refer to Solr in Action (Grainger & Potter, 2014)

4.2.3 Installing Nutch 1.9

Unlike the Nutch 2.x family which needs to be compiled into runtimes 1.9 comes in a pre-compiled compressed file (http://nutch.apache.org/downloads.html). Again uncompressing it is all that is needed to ready it for use from the installation perspective. The uncompressed package contains the following.

4.2.4 Configuring Nutch 1.9

The first step to run Nutch is to make sure that it is running on a Linux environment (OSX works well), because of this a VM is recommend if Linux or OSX is not the native operating system. Java is also required for running Nutch so make sure 1.7+ is available in the shell environment. With Java installed, Nutch requires that JAVA_HOME must be first set (OSX can do something like: export JAVA_HOME=$(/usr/libexec/Java_home and verify it by echo $JAVA_HOME).

After Java is ready Nutch requires the user to setup a folder at the root level called URLs, within this folder is a file called seeds.txt. In the seeds text file each line represents a URL that the crawler need to crawl, for example www.amazon.ie.

Lastly the nutch-site.xml file which contains all the non-default settings for Nutch will need an http.agent.name

```
<configuration>
        <property>
                <name>http.agent.name</name>
                        <value>MyBot</value>
                        <description>MUST NOT be empty. The advertised version will have Nutch
                        appended.</description>
        </property>
```

4.2.5 Crawling

Once Nutch is setup, start the Solr mini web container by going into the example folder and execute Java –jar start.jar. This will setup SOLR on port 8983 (default, can be changed) to listen to any incoming Restful requests.

With SOLR running on localhost:8983 Nutch can start via the command ./bin/crawl urls/ testCrawl/ http://localhost:8983/Solr/collection1 1. The command composes of the actual binary executable crawl, the folder of the seeds.txt, the folder name of the output data (created automatically), optional SOLR indexer and finally the level which means how deep the crawl should go.

The idea of the level is interesting as it controls what happens to URLs on the page being crawled. If it is 1 it will not go into any of the URLs on the page where if greater than one each depth is counted as 1. So 2 will allow a crawl of each page linked off the main page. If this was a folder the same concept applies to the hierarchy.

4.3 The relational database index setup

4.3.1 Setting up and configuring MySQL

MySQL can be downloaded from http://www.mysql.com/ and the installer setup is very straight forward. Once downloaded a few things need to be tweaked in order for the DB to hold the data that Nutch Crawler will push into it. Looking through the MySQL options in the configuration file (varies with systems, on OSX it is /etc/mysql/my.cnf) the following needs to be set.

```
innodb_file_format=barracuda
innodb_file_per_table=true
innodb_large_prefix=true
character-set-server=utf8mb4
collation-server=utf8mb4_unicode_ci
max_allowed_packet=500M
```

Restart the MySQL service and create a new schema instance called say “index”. In this new schema create a table for storing the Crawler results using SQL (the preferred way to do this is using the MySQL UI workbench also available for free on the MySQL site).

```
CREATE TABLE `result`
(
`id` varchar(767) NOT NULL,
`headers` blob,
`text` longtext DEFAULT NULL,
`status` int(16) DEFAULT NULL,
`markers` blob,
`parseStatus` blob,
`modifiedTime` bigint(24) DEFAULT NULL,
`prevModifiedTime` bigint(24) DEFAULT NULL,
`score` float DEFAULT NULL,
`typ` varchar(32) CHARACTER SET latin1 DEFAULT NULL,
`batchId` varchar(32) CHARACTER SET latin1 DEFAULT NULL,
`baseUrl` varchar(767) DEFAULT NULL,
`content` longblob,
`title` varchar(2048) DEFAULT NULL,
`reprUrl` varchar(767) DEFAULT NULL,
`fetchInterval` int(16) DEFAULT NULL,
`prevFetchTime` bigint(24) DEFAULT NULL,
`inlinks` mediumblob,
`prevSignature` blob,
`outlinks` mediumblob,
`fetchTime` bigint(24) DEFAULT NULL,
`retriesSinceFetch` int(16) DEFAULT NULL,
`protocolStatus` blob,
`signature` blob,
`metadata` blob,
PRIMARY KEY (`id`)
)
ENGINE=InnoDB
ROW_FORMAT=COMPRESSED
DEFAULT CHARSET=utf8mb4;
```

The columns of this table matches line by line the fields using by SOLR in the collection1 core’s schema. Now MySQL is ready to store the results from Nutch 2.

4.3.2 Installing Nutch 2

Nutch 2 can be downloaded from the same location as Nutch 1.9. Unfortunately the Nutch 2 distribution do not come as a runtime, it requires compilation under the environment which it is to be run on. This process was found to be tough depending on the version of OS and Shell that it was ran on. In general there is a requirement for IVY and Ant to be installed and configured correctly before the compilation can finish successfully.

The process starts with the editing of ${APACHE_NUTCH_HOME}/ivy/ivy.xml file. This file contains reference to the gora-core which is the data-store for the Nutch indexes. For the latest version of Nutch this is set to version .3 but this will not work with MySQL and need to be reverted back down to version .2.

```
<dependency org="org.apache.gora" name="gora-core" rev="0.2.1" conf="*->default"/>
```

To push the index data into MySQL the connector for gora-sql (Apache Software Foundation, 2015) is needed

---

```
<dependency org="org.apache.gora" name="gora-sql" rev="0.1.1-incubating" conf="*->default" />
<dependency org="mysql" name="mysql-connector-Java" rev="5.1.18" conf="*->default"/>
```

Because the build has not occurred yet, it is possible to set these dependencies so that ant will pull them from the public repositories when the build command initializes. With the dependencies set the actual connection details to the MySQL database will need to be set in ${APACHE_NUTCH_HOME}/conf/gora.properties

---

```
################################
# MySQL properties #
################################
gora.sqlstore.jdbc.driver=com.mysql.jdbc.Driver
gora.sqlstore.jdbc.url=jdbc:mysql://localhost:3306/nutch?createDatabaseIfNotExist=true
gora.sqlstore.jdbc.user=xxxxx
gora.sqlstore.jdbc.password=xxxxx
```

The Gora mapping will also need to set the primary key id length to the max possible value match the column definition in the crate table SQL earlier by change the file ${APACHE_NUTCH_HOME}/conf/gora-sql-mapping.xml.

```
<primarykey column=”id” length=”767”/>
```

Finally, similar to Nutch 1.9 change add to the file ${APACHE_NUTCH_HOME}/conf/nutch-site.xml the default http.agent.name. Finally build the runtime using ant runtime. This will create a new runtime folder with the binaries similar to the out of box file structure of Nutch 1.9.

4.3.3 Configuring Nutch 2

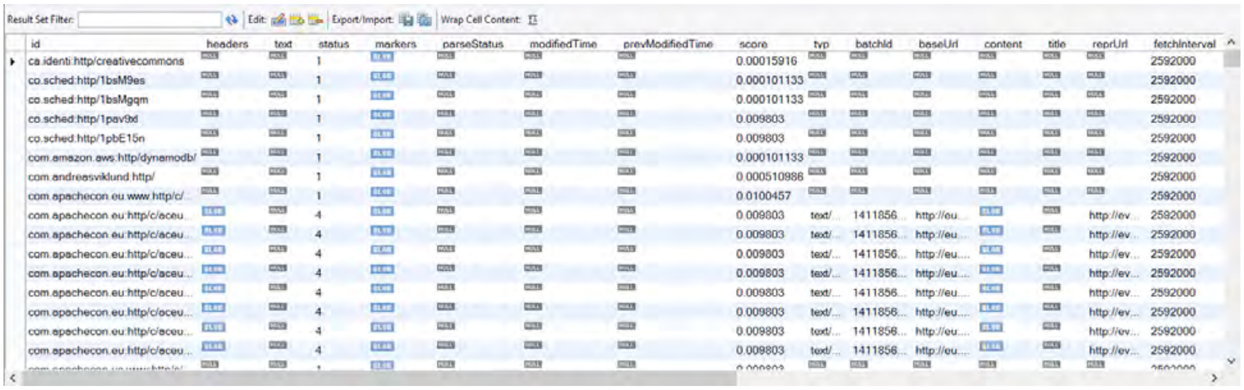
Configuring Nutch 2 is the same as 1.9 one the runtime folder and files are created.

4.3.4 Crawling

While it is possible to crawl the site just like described in crawling using Nutch 1.9 the binary commands for Nutch 2 shows that it is possible to run the crawl in steps to see the result better.

- Inject the seed URLs into the crawl engine: bin/nutch inject urls.
- Generate any child links: bin/nutch generate -topN 20
- Fetch the actual pages: bin/nutch fetch –all
- Parse all the pages that is fetched: bin/nutch parse –all
- Finally store the crawled data into the database/indexer: bin/nutch updated

Test the result by selecting from the index.result table.



Result Set Filter: | Edit: | Export/Import: | Wrap Cell Content:

id	headers	text	status	markers	parseStatus	modifiedTime	prevModifiedTime	score	typ	batchId	baseUrl	content	title	reprUrl	fetchInterval
ca.identi:http/creativecommons	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.00015916	NULL	NULL	NULL	NULL	NULL	NULL	2592000
co.sched:http/1bsM9es	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.000101133	NULL	NULL	NULL	NULL	NULL	NULL	2592000
co.sched:http/1bsMgqm	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.000101133	NULL	NULL	NULL	NULL	NULL	NULL	2592000
co.sched:http/1pav9xl	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.009803	NULL	NULL	NULL	NULL	NULL	NULL	2592000
co.sched:http/1pbE15n	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.009803	NULL	NULL	NULL	NULL	NULL	NULL	2592000
com.amazon.aws:http/dynamodb/	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.000101133	NULL	NULL	NULL	NULL	NULL	NULL	2592000
com.andreasviklund:http/	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.000510986	NULL	NULL	NULL	NULL	NULL	NULL	2592000
com.apachecon.eu.www:http/c/...	NULL	NULL	1	BLOB	NULL	NULL	NULL	0.0100457	NULL	NULL	NULL	NULL	NULL	NULL	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu...	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu...	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu....	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu....	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu....	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu....	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu....	BLOB	NULL	http://ev...	2592000
com.apachecon.eu:http/c/aceu...	BLOB	NULL	4	BLOB	NULL	NULL	NULL	0.009803	text/...	1411856...	http://eu....	BLOB	NULL	http://ev...	2592000

Figure 35 Database table index

4.4 The common Restful web service

4.4.1 The Development environment

To develop the Restful service the IDE of choice is STS or Eclipse. Tomcat is then embedded into the IDE (Integrated Development Environment) to host the web services via port 8080. Notice the initial folder structure in the left hand side of the IDE screenshot showing the initial 3 core components of the project code:

- Domain; this is the package holding the Java objects directly representing the tables in the database (in this case the index table)
- JPA; this is the package holding the various logic for domain persisting and retrieval. HQL (a form of SQL) is presented here for manipulating the index table
- Rest.endpoint; this is the package that intercepts the HTTP calls from the outside via port 8080

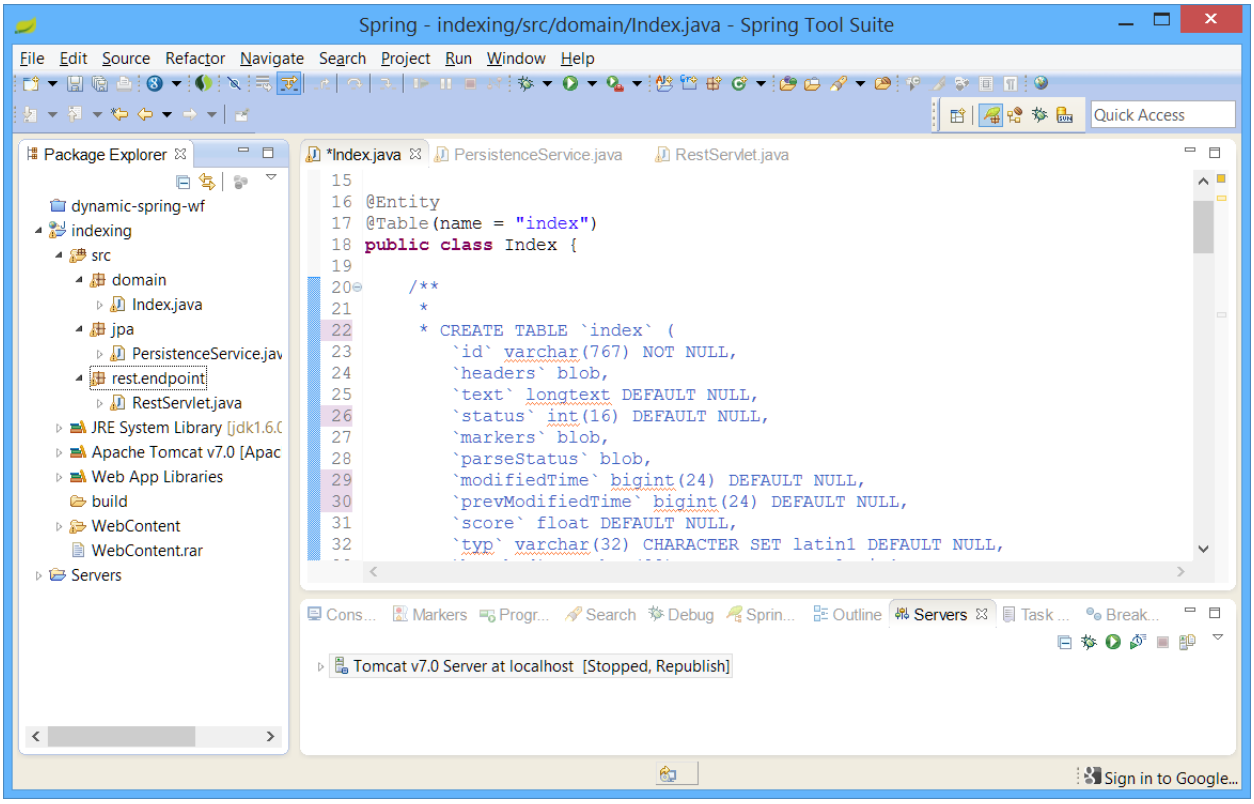


Figure 36 The IDE STS

4.4.2 Persistence project setup

The connection details for JPA to connect to the database is held in the persistence-cfg.xml file along with other configurations for the manipulation of database via JPA objects.

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/nutch_dit_01" />
        <property name="username" value="root" />
        <property name="password" value="root" />
</bean>
```

4.4.3 REST servlet project setup

The Restful interceptors are setup via the rest-servlet.xml which has the top level binding of the services URLs such as these defined in the web.xml.

```
<servlet>
        <servlet-name>rest</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
        <servlet-name>rest</servlet-name>
        <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Effectively this means that anything coming into the URL context of this web app will need to append /rest/ before it is passed into the Restful servlets (explained later). So to access the servlets here the full path would look like http://localhost:8080/appName/rest/endPointName

4.4.4 Rest servlets

The REST servlets are annotated classes which represent the handling of the HTTP Restful calls. For example the primary REST handler has the class annotation:

```
@Controller
@RequestMapping("/json")
@SuppressWarnings("unchecked")
public class RestServlet {
```

The top controller annotation means this class is indeed a controller so it can handle incoming requests. The request mapping designates which URL calls it should respond to, duplicates among controllers is not allowed, in this case this controller will respond to http://localhost:8080/appName/rest/json/.

4.4.5 JPA objects

The coreBase,Java file is the core Java base super-class holding the CORE tables attributes that will allow easy addition of additional cores that is actually mapped to the CORE tables. Additional CoreX.java are then mapped to the actual CoreX DB tables in the MySQL.

The super class is annotated as such:

```
@MappedSuperclass

public class CoreBase {
```

With extension classes extending the above super base class

```
@Entity
@Table(name = "core2")

public class Core2 extends CoreBase{
```

What one instance of the extended classes represents is the Java equivalent of a single line from the index table in the database. To achieve this each column in the database table need to be represented with the correct data format and mapping column names etc.

First the class needs to be mapped to the actual table by using the @Table attribute and along with the entity declaration means this is a table mapping. The table annotation tells of the table name which it is mapped to. Within the class there is fields mapped in similar way:

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "id")
private String id;

@Column(name = "headers", columnDefinition="blob")
private Blob headers;

@Column(name = "text", length=512)
private String text;
```

The column annotation shows that these are columns and the name of the column can be seen here. As the SQL for creating the table came first mapping this JPA object to the table had challenges in the column formats particularly the column definitions.

4.4.6 Deploying SolrJ

The SolrJ connectors are bundled as jar files

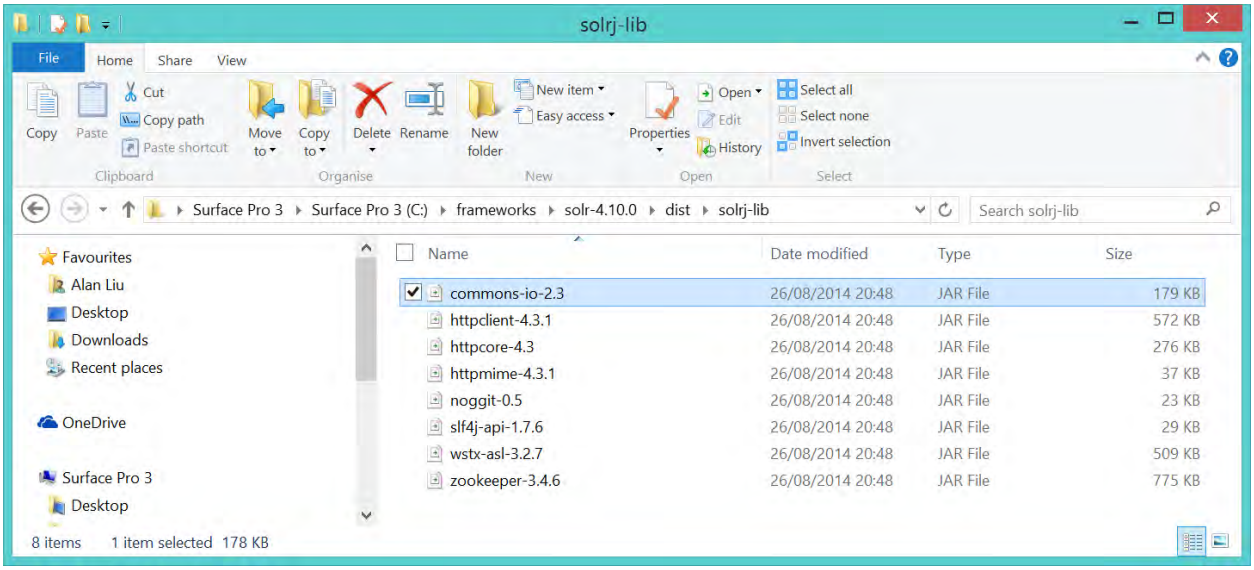


Figure 37 The SolrJ jars

Calling on the SolrJ involves using the Solr classes on a machine with the Solr rest service running (default port is 8983).

4.4.7 Basic indexing endpoint

Once the servlet is setup, connected to the JPA layer a simple REST endpoint call looks like this:

```
@RequestMapping(value = "/match/{fieldName}/{value}", method = RequestMethod.GET)
public @ResponseBody
Object getMatches(@PathVariable String fieldName, @PathVariable String value) {
        List<CoreBase> indexes = persistenceService.match(fieldName, value);
        List<IndexResult> indexResults = new ArrayList<IndexResult>();
        for(CoreBase c : indexes) {
                IndexResult r = new IndexResult();
                r.setId(c.getId());
                indexResults.add(r);
        }
        return indexResults;

}
```

From the code it can be seen that the annotated handler used here for intercepting Restful calls are marked with the @requestMapping attribute on top of the method name.

4.4.8 Accessing the service from external web browser or service can be achieved by hitting the URL such as the test URL below:

---

http://localhost:8080/indexing/rest/json/match/text/wiki

The host name and port during the experiment is local and 8080, the project which is deployed into the IDE and tomcat is called indexing hence the /indexing/ part. Because the project is a spring project it is decided to split the rest handlers to the /rest/ URL. In order to match the result in the JSON format (the only one deployed) the path specifier is /json/. “match” is the handler for the default DB service in which it can match the subsequent /*/ to the word again following /*/ so /text/wiki means match “wiki” in the db field “text”.

For the final deployment this will change to the Restful service handler:

---

http://localhost:8080/indexing/rest/json/search/3/state

Where the Restful service handler will be the search controller with the following two parameters being the core to search for and the keyword to search for (in the above case it is core 3 and keyword state).

5 EVALUATION

5.1 Introduction

The evaluation phase consist of using the established framework to measure the performance of crawling, indexing and searching.

The overall evaluation was conducted between 6 index cores in Solr and the database. 7 cores in total will be created with the first one indexing just a single layer of a seed URL to have a base score for the process overhead. The cores will differ in size growing from approximately 150 entries to 6000 entries to measure the impact of large indexes in the searching performances.

5.2 Crawling

5.2.1 Introduction

Nutch 2 Crawler is the unified crawler that will push index data into both the MySQL database and the Solr4 index.

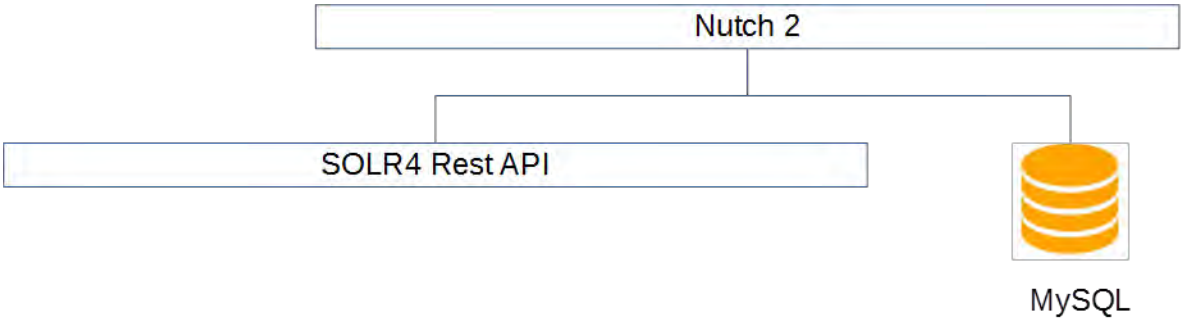


Figure 38 Nutch pushes to the MySQL DB and Solr within the same session during a crawl run

The crawling process is controlled by two key numeric values. The **iteration** (LIMIT) count that controls how many levels of URLs are fetched and crawled and the **topN** value controlling the max number of URLs from a given level is parsed. A detailed explanation of this can be found at (Olston & Najork, 2010)

The crawling script is a customized script based on the default nutch crawl script that gets bundled with Nutch (Apache Foundation, 2015). The custom modifications made to it allows the crawling process to both write to the relational database and the Solr index in sequence. The main function of the script is to manage the four (five including solr) distinct steps in the crawling process and call their binaries within the nutch framework.

---

main loop : rounds of generate - fetch - parse - update

for ((a=1; a <= LIMIT ; a++)) *#for loop that controls the iterations the following command get run*

do

$bin/nutch generate -topN 750 #1 *generate all URLs eligible for crawling in this iteration but no more than this value in total*

$bin/nutch fetch -all -threads 50 #2 *fetch all the URL content*

$bin/nutch parse –all #3 *parse all the content*

$bin/nutch updatedb #4 *push the index data into the database*

$bin/nutch solrindex $SOLRURL –all #5 *push into the solr index*

$bin/nutch solrdedup $SOLRURL *#cleanup solr*

done

In effect this code is processing the crawl shown in the below figure:

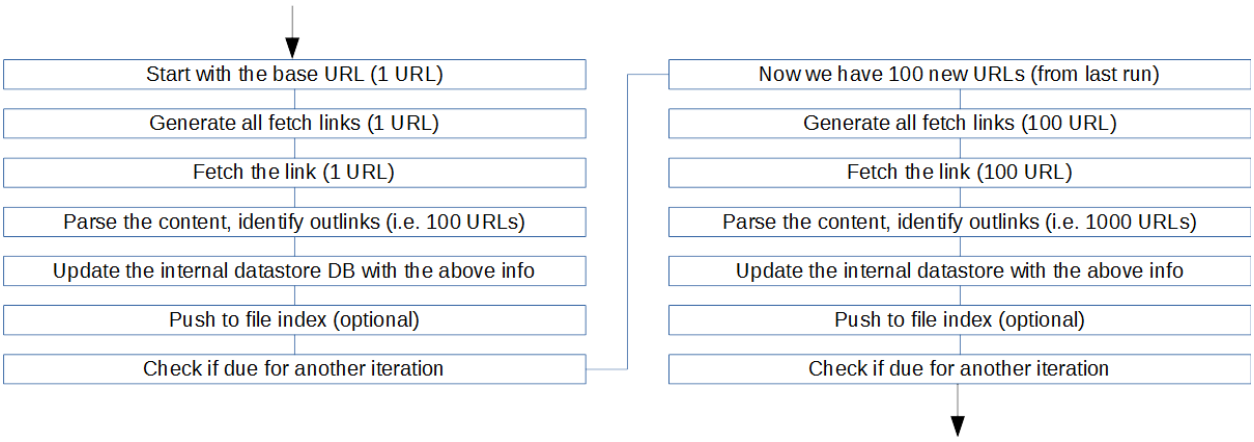


Figure 39 The crawling process and steps

5.2.2 Interpreting the crawl process logs

Due to the fact that a long crawl outputs a log file up to a few MBs in size below shows only 1 URL crawling (Iteration = 1). Because iteration is 1 the topN variable is irrelevant as no outlinks from the top URL will be processed due to not having a second iteration. If the top iteration seed file has more than 2000 URLs then only 2000 will be fetched.

// initiate the crawl process, designate which Solr core to push to and the output log file.

./bin/crawl urls/ wikipedia/ http://localhost:8983/solr/collection1 1 | tee wiki-n1-l1-C1-file.txt
Crawl started: 18:08:30

// Injection of the initial URL(s) into the first crawling iteration
InjectorJob: starting at 2015-02-05 18:08:31
InjectorJob: Injecting urlDir: urls

// unneeded nutch configuration
2015-02-05 18:08:31.499 java[42178:1903] Unable to load realm info from SCDynamicStore
// designating that the indexing store to DB via Gora
InjectorJob: Using class org.apache.gora.sql.store.SqlStore as the Gora storage class.

// no URLs rejected so far
InjectorJob: total number of urls rejected by filters: 0
InjectorJob: total number of urls injected after normalization and filtering: 1
Injector: finished at 2015-02-05 18:08:33, elapsed: 00:00:02

// first iteration (of the only one)
Thu 5 Feb 2015 18:08:33 GMT : Iteration 1 of 1

// generating run variables and the first URL(s) to fetch
Generating batchId
Generating a new fetchlist
GeneratorJob: starting at 2015-02-05 18:08:34
GeneratorJob: Selecting best-scoring urls due for fetch.
GeneratorJob: starting
GeneratorJob: filtering: true
GeneratorJob: normalizing: true
// note the topN is set to 2000 but only one iteration means only the top one will be processed
GeneratorJob: topN: 2000
2015-02-05 18:08:34.505 java[42194:1903] Unable to load realm info from SCDynamicStore
GeneratorJob: finished at 2015-02-05 18:08:36, time elapsed: 00:00:02
GeneratorJob: generated batch id: 1423159714-1457766807

// fetching start
Fetching :
FetcherJob: starting
FetcherJob: fetching all
Fetcher: Your 'http.agent.name' value should be listed first in 'http.robots.agents' property.
FetcherJob: threads: 50
FetcherJob: parsing: false
FetcherJob: resuming: false
FetcherJob : timelimit set for : -1
2015-02-05 18:08:37.285 java[42201:1903] Unable to load realm info from SCDynamicStore
Using queue mode : byHost
Fetcher: threads: 50
QueueFeeder finished: total 1 records. Hit by time limit :0
// note this is the seed URL from the seeds file (only one link)
fetching http://en.wikipedia.org/wiki/Computing (queue crawl delay=5000ms)
-finishing thread FetcherThread1, activeThreads=1
-finishing thread FetcherThread2, activeThreads=1

-finishing thread FetcherThread3, activeThreads=1
-finishing thread FetcherThread4, activeThreads=1
-finishing thread FetcherThread5, activeThreads=1
...
// 50 thread is allocated by default, but because there is only one URL the rest can be shut down
Fetcher: throughput threshold: -1
-finishing thread FetcherThread49, activeThreads=1
Fetcher: throughput threshold sequence: 5
-finishing thread FetcherThread0, activeThreads=0
0/0 spinwaiting/active, 1 pages, 0 errors, 0.2 0 pages/s, 102 102 kb/s, 0 URLs in 0 queues
-activeThreads=0
FetcherJob: done

// this is where the fetched data is parsed to be indexed
Parsing :
ParserJob: starting
ParserJob: resuming: false
ParserJob: forced reparse: false
ParserJob: parsing all
2015-02-05 18:08:45.359 java[42208:1903] Unable to load realm info from SCDynamicStore
Parsing http://en.wikipedia.org/wiki/Computing
ParserJob: success

// pushing the indexed data into the relational DB
CrawlDB update for wikipedia/
DbUpdaterJob: starting
2015-02-05 18:08:48.612 java[42216:1903] Unable to load realm info from SCDynamicStore
DbUpdaterJob: done
Pushing to database took: 0 minutes and 3 seconds

// pushing the indexed data to a solr index
Indexing wikipedia/ on SOLR index -> http://localhost:8983/solr/collection1
SolrIndexerJob: starting
2015-02-05 18:08:51.524 java[42225:1903] Unable to load realm info from SCDynamicStore
Adding 1 documents
SolrIndexerJob: done.
SOLR dedup -> http://localhost:8983/solr/collection1
2015-02-05 18:08:54.444 java[42234:1903] Unable to load realm info from SCDynamicStore
Pushing to SOLR took: 0 minutes and 5 seconds

Total process time took: 0 minutes and 25 seconds

// process finish

5.2.3 Crawling counts

Both the database and the Solr core will be setup to hold the following Cores/Tables for Wikipedia content (including references to files and images). The Wikipedia top link for

crawling is *http://en.wikipedia.org/wiki/Software*. Below shows the iteration, topN and the number of entries the crawls indexed. Note the number of entries are subject to change as the pages content (including the outbound links) are updated from day to day.

Core/Table count	Iteration	topN	Average number of entries for Wiki Software indexed	Total URLs encountered during parsing for the next iteration run up to topN
1	1	500	1	156
2	2	500	153	11483
3	3	500	592	31698
4	3	750	811	39411
5	4	1500	2908	95693
6	4	2000	3817	119630
7	2x4	2x2000	5941	153564

Figure 40 Number of indexed pages and outlinks based on iteration and topN

Figure 43 shows the number of URLs generated from parsing the number of URLs (limited by topN) during each iteration.

The last column indicates the number of URLs encountered during all iterations. They are governed and limited to the **topN** parameter which means for example by the third level/iteration

there could be many thousands of outlink URLs but they will be only processed up to the **topN** parameter, this means if the topN parameter is -1 (unlimited) all the 100 thousand pages would be processed taking up an estimated one week+ time.

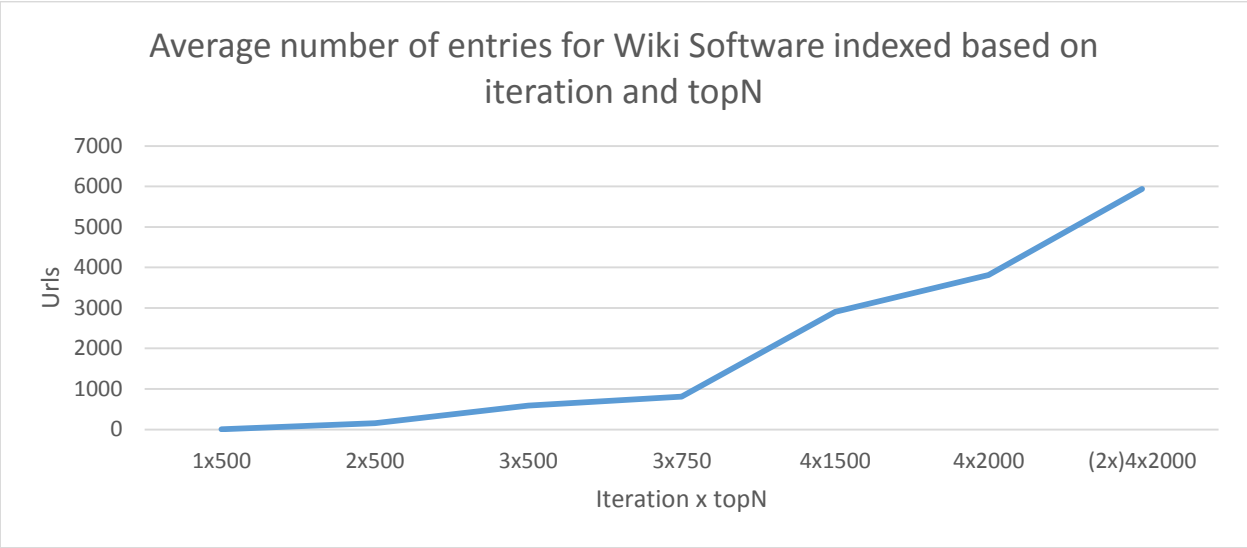


Figure 41 Number of URLs indexed based on Iteration and topN values

As figure 43 indicates the combination of iteration and topN has distinct effect on the number of URLs that gets indexed. No matter what the topN is set to on iteration 1 the number of URLs being indexed will always be equal to the number supplied by the seed.txt file. But depending on the number of outlinks from each of the seed URLs topN will take effect in all iterations following the first.

For example (taking the second iteration and topN combination of 2 and 500) if the total number of outlinks is always less than topN say 100, by iteration 2 the topN limit of 500 will have no effect, total URLs indexed will be 100+1 (from the single initial seed URL) but if it is greater than 500 then the growth of indexed data will be a linear progression as each iteration will return more than 500 but only the first 500 will be indexed from each iteration or in effect depth levels.

This means that to effectively crawl a site it is important to either maximize the topN or to remove it completely (setting it to -1) so that all links are crawled. Having said that Wikipedia is unique in the sense that it is almost an infinite tree of outlinks and to index such a structure it would be impossible to use a topN of -1 as it will go on recursively. By using topN it would

allow the recursion to stop by limiting the depth and fetch count so that when the perfect combination is found it would be fetching and indexing the same webpage more than once.

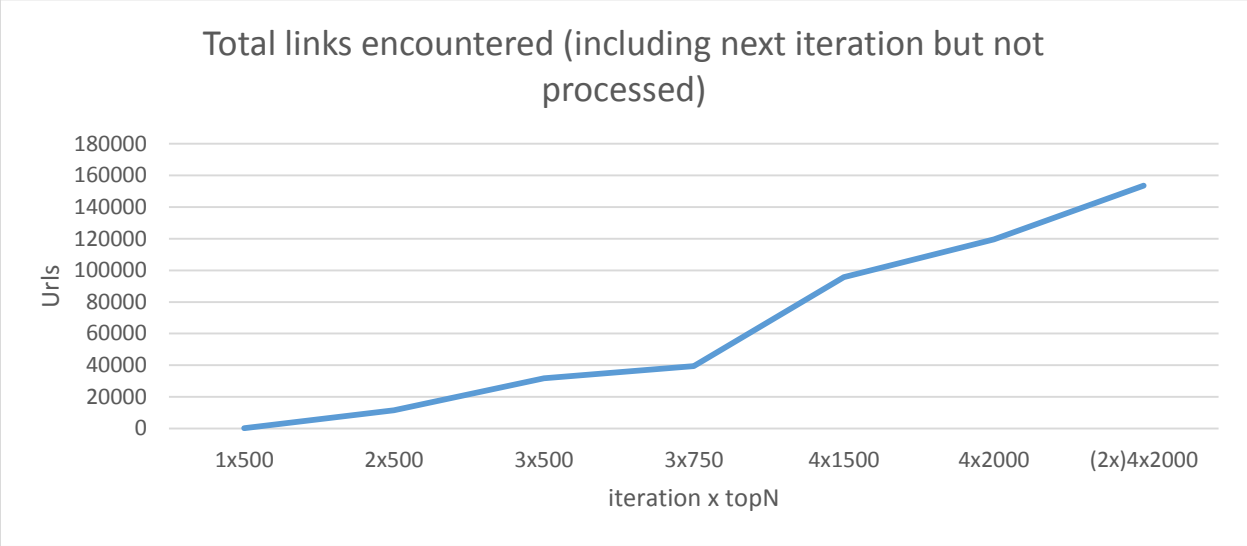


Figure 42 Total links encountered (including outlinks but not all processed) based on iteration and topN values

In the case of using the Wikipedia entry of "Software" as the seed we can see from figure 44 that the number of links on each of the page is very high, by the second iteration the number of outlinks from 156 URLs (the number of outlinks from iteration 1) has grown exponentially to 11483. The increase we see at 3x750 to 3x500 shows that by parsing an extra 250 URLs in the third iteration (we know this as the second iteration only had a maximum of 157, below the topN limit) the number of outlinks grows modestly by almost 8 thousand, but increasing the iteration to 4 and topN to 1500 more than doubles the amount of links encountered.

5.2.4 Crawling speeds

Below is the average time for the overall crawling process on a 10MB internet line vs 100MB internet line to see the effect of line speed on the number of data indexed. Each crawl (for each iteration/topN combination) was repeated three times with the average taken

Number of indexed page	Core/Table No.	Overall crawl and index time on 10MB line	On a 100MB line
1	1	00:25	00:24
153	2	00:15:20	00:15:17
592	3	01:12:55	01:11:45
811	4	01:34:45	01:32:30
2908	5	07:04:09	07:00:03
3817	6	09:11:57	09:10:13
5941	7	20:02:52	19:51:25

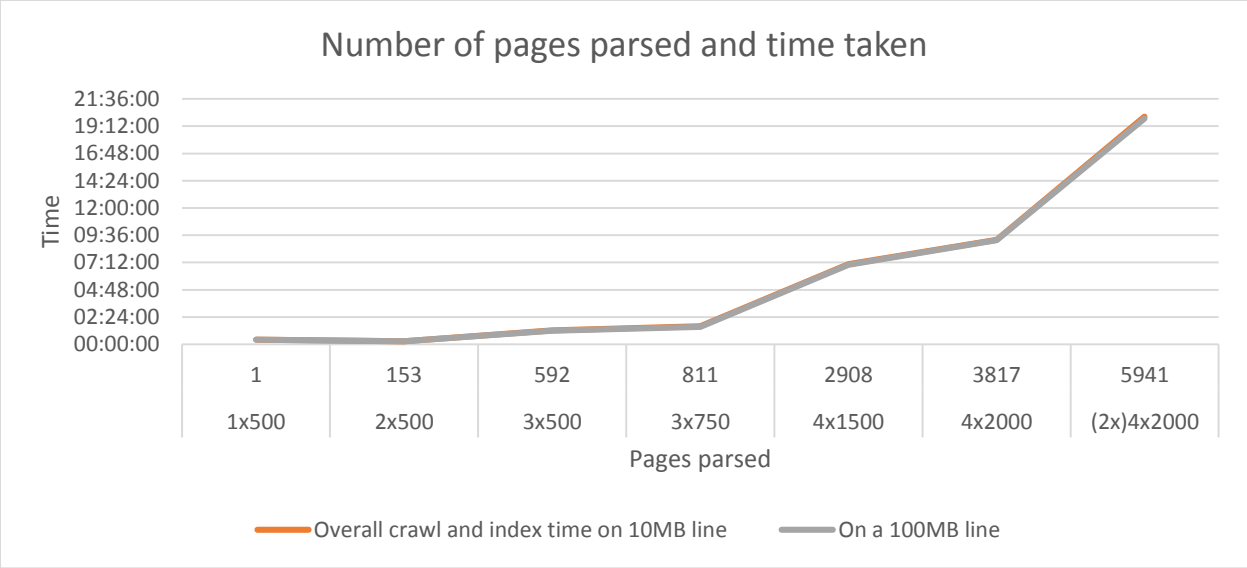


Figure 43 Number of pages index and time taken

Crawling speeds over a 10MB line and a 100MB line do not seems to differ significantly. This is in part governed by the default Nutch setting of respecting websites by limiting a wait time of 5 seconds when the number of fetchers reaches a maximum. If the indexing is done on a private content where the need to respect throttling is not important then the crawlers can be setup to never wait and concurrent crawlers can be setup to index the same content.

During the last core population (7) the 4 iteration of 2000 topN was executed against the Wikipedia entry "Software" and "Computing" what is interesting to see is that despite running for roughly the same length of time the numbers of non-overlapping out linking URLs only increased by 50% also reflected in the number of unique index entries. This is one of the key feature of Nutch and Solr combination in that it will not drop or duplicate content.

5.2.5 Indexing speeds

The following table indicates the average times for the total crawl, the db index inserts and the Solr index inserts

Number of indexed page	Core/Table No.	Average time for DB inserts	Average time for Solr index inserts
1	1	00:03	00:05
153	2	00:09	00:11
592	3	00:22	00:21
811	4	00:25	00:22
2908	5	01:02	00:51
3871	6	01:19	01:03
5941	7	02:51	02:14

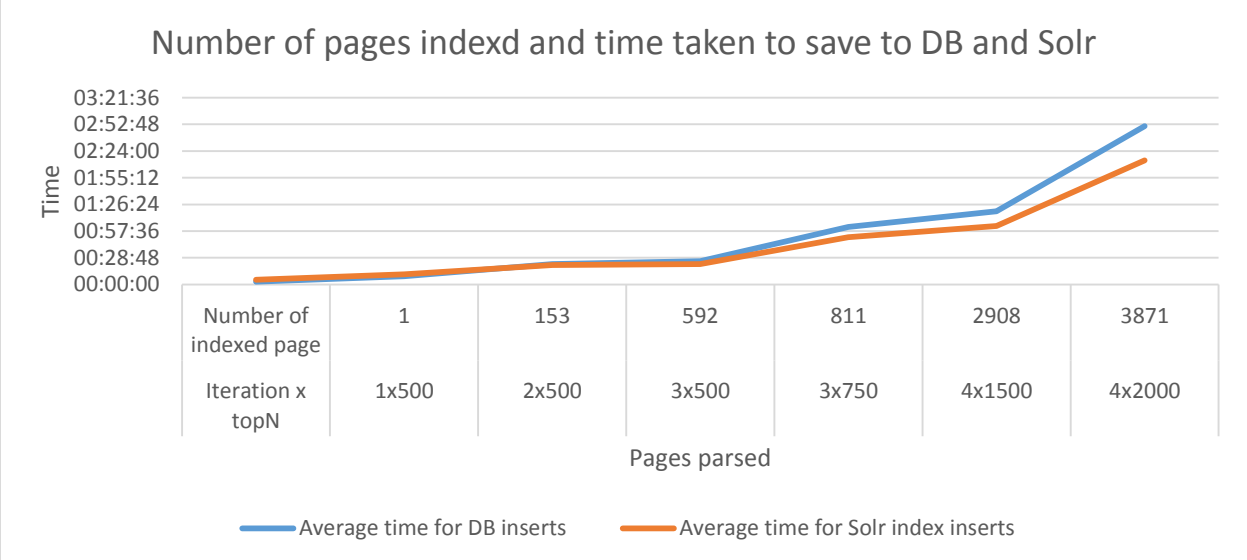
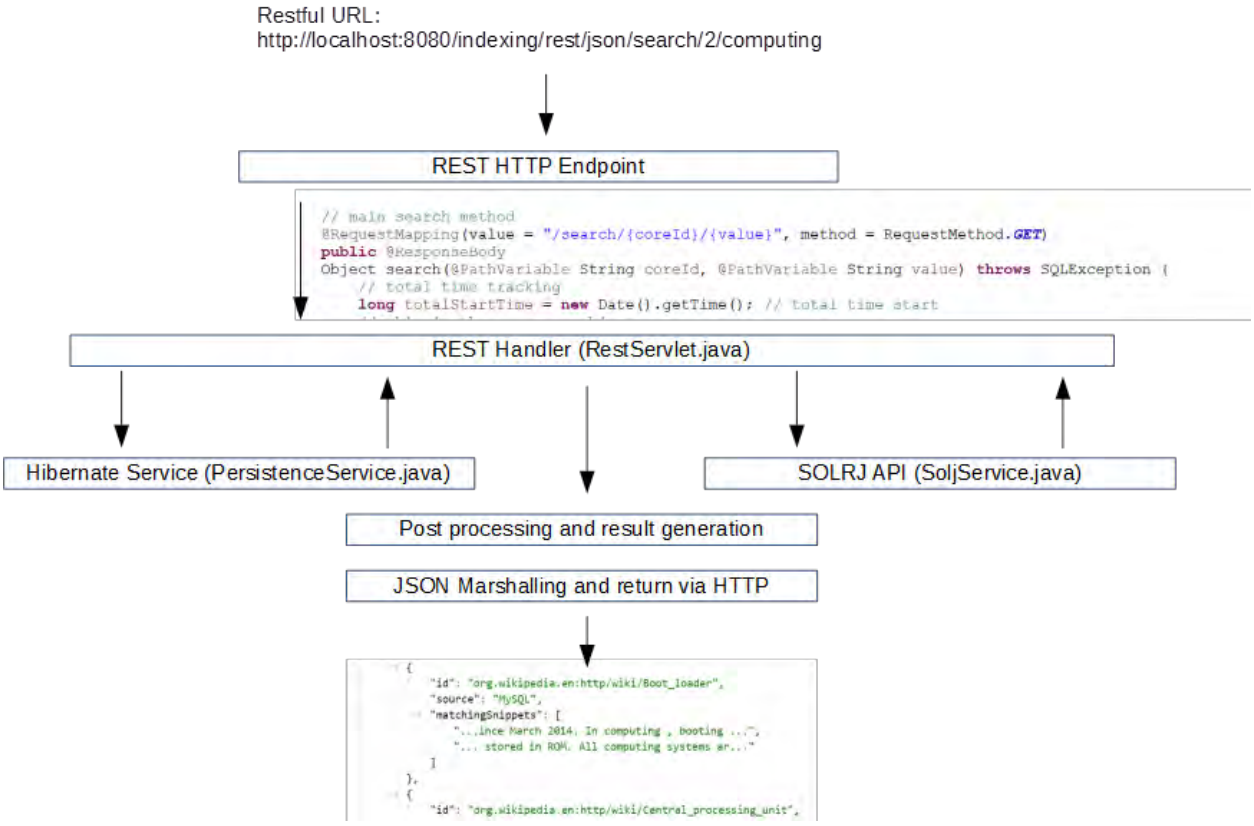


Figure 44 Number of pages indexed and time taken to save to DB and Solr

5.3 Searching

5.3.1 Introduction

With the cores populated from the crawling section the search framework uses the Hibernate framework and the SolrJ connector to retrieve indexed items from the Database and Solr respectively. Random computing related terms are picked and searched again all the cores except the first (having one entry in core one render it useless for assessing searching speeds). Each source then respond with the entries it matches using a combination of framework java logic and the connectors aforementioned. The two phases are executed concurrently with each synchronous call timed and logged before the overall result is returned by the Restful service via JSON.



5.3.2 Key logic analysis

Below is an analysis of the steps in the programming logic in compiling the JSON result of a search.

```
// main search method
@RequestMapping(value = "/search/{coreId}/{value}", method = RequestMethod.GET)
public @ResponseBody
Object search(@PathVariable String coreId, @PathVariable String value) throws SQLException {
	// total time tracking
	long totalStartTime = new Date().getTime(); // total time start
	// this is the response object
	SearchResponse response = new SearchResponse();
	// get the results from db first
	long lStartTime = new Date().getTime(); // time start
	List<CoreBase> dbResults = persistenceService.searchCore(coreId, value);
	long lEndTime = new Date().getTime();// time end
	//set the response
	response.setTotalDBMatches(String.valueOf(dbResults.size()));
	response.setTotalTimeTakenFetchingFromDB((lEndTime - lStartTime) + " ms");
	// two lists to carry the diff'd ids
	List<String> idsInDbNotInSolr = new ArrayList<String>();
	List<String> idsInSolrNotInDb = new ArrayList<String>();
	// process the DB results
	// db ids for diffing, i.e. the differences between the two result sets
	List<String> dbIds = new ArrayList<String>();
	lStartTime = new Date().getTime(); // time start
	List<IndexResult> indexResults = new ArrayList<IndexResult>();
	for(CoreBase c : dbResults) {
		IndexResult r = new IndexResult();
		r.setId(c.getId());
		r.setSource("MySQL");
		List<String> snippets = getMatchingSnippets(new String(c.getText().getBytes(1, (int)
c.getText().length())), value);
		r.setMatchingSnippets(snippets);
		dbIds.add(c.getId());
		indexResults.add(r);
	}
	lEndTime = new Date().getTime();// time end
	response.setTotalTimeTakenProcessingDBResults((lEndTime - lStartTime) + " ms");
	//get Solr results
	try {
		lStartTime = new Date().getTime(); // time start
		// get the solr results via the solrJ logic
		QueryResponse rsp = SolrJService.getSolrQueryResponse(value, coreId);
		lEndTime = new Date().getTime();// time end
		// set the response
		response.setTotalTimeTakenFetchingFromSolr((lEndTime - lStartTime) + " ms");
		// process Solr results
```

```
                // solr ids for diffing
                List<String> solrIds = new ArrayList<String>();
                int i = 0;
                lStartTime = new Date().getTime(); // time start
                Iterator<SolrDocument> iter = rsp.getResults().iterator();
            while (iter.hasNext()) {
                IndexResult r = new IndexResult();
                SolrDocument resultDoc = iter.next();
                //String content = (String) resultDoc.getFieldValue("content");
                String id = (String) resultDoc.getFieldValue("id");
                String content = (String) resultDoc.getFieldValue("content");
                List<String> snippets = getMatchingSnippets(content, value);
                r.setMatchingSnippets(snippets);
                r.setSource("Solr");
                r.setId(id);
                if(snippets.size() > 0) {
                                indexResults.add(r);
                                solrIds.add(id);
                                i++;
                }
            }
            for(String s : solrIds){
                if(!dbIds.contains(s)){
                        idsInSolrNotInDb.add(s);
                }
            }
            for(String s : dbIds){
                if(!solrIds.contains(s)){
                        idsInDbNotInSolr.add(s);
                }
            }
            response.setIdsInDbNotInSolr(idsInDbNotInSolr);
            response.setIdsInSolrNotInDb(idsInSolrNotInDb);
            lEndTime = new Date().getTime();// time end
                response.setTotalTimeTakenProcessingSolrResults((lEndTime - lStartTime) + " ms");
            response.setTotalSolrMatches(String.valueOf(i));
        } catch (SolrServerException e) {
                e.printStackTrace();
        }
        // set the results
        response.setResults(indexResults);
        response.setTotalMatches(String.valueOf(indexResults.size()));
        long totalEndTime = new Date().getTime(); // total time start
        response.setTotalTimeTaken((totalEndTime - totalStartTime) + " ms");
        return response;

}
```

5.3.3 Result output format

The final JSON result response is based on a Java POJO that contains several properties and lists that is used to evaluate the two index cores. From these properties subsequent reading are made so that the average of the two cores can be compared. When the JSON results node is expanded each of the indexed object can be seen with its ID, source and the matching snippets which is used to explain why this entry is returned for the search term. A sample of this is shown in figure below.

Figure 45 A image of the HTTP HTML JSON output from the search service

5.3.4 Result count differences

It can be noted that the response matching entries differ in number from the DB to Solr upon closer analysis this is found to be due to the DB Logic capturing part word matches as its default behavior. For example searching computing on core two yields an id difference shown in the figure below.

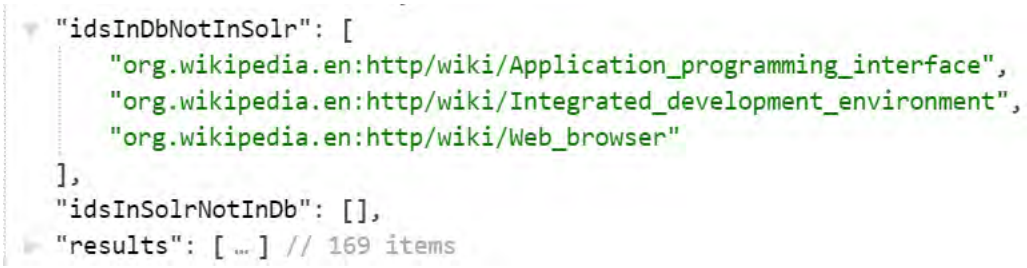


Figure 46 Ids not present in the result from both indexes

If that Id is then looked up in the JSON (for example web_browser) it can be shown that the reason it's in the DB match is due to semi word matching and not whole word matching as shown in the figure below.

Figure 47 Example of partial word matching in DB index

5.3.5 Searching speeds

Keyword "computing" across all cores:

Core index count	Total time (ms)	DB total (ms)	Solr Total (ms)	DB Results	Solr Results

153	447	150	297	86	83
592	1171	893	378	281	265
811	2178	1812	366	357	334
2908	6636	5539	533	1065	1009
3871	7822	6945	877	1279	1202
5941	12387	11108	2329	1727	1658

Keyword “software” across all cores

Core index count	Total time (ms)	DB total (ms)	Solr Total (ms)	DB Results	Solr Results
153	582	253	329	134	131
592	732	380	352	498	469
811	1975	1537	438	665	622
2908	7816	6437	1379	2011	1923
3871	9044	7899	1145	2541	2405

5941	13309	11061	2264	3228	3111

Keyword “debug” across all cores

Core index count	Total time (ms)	DB total (ms)	Solr Total (ms)	DB Results	Solr Results
153	315	80	235	39	16
592	1416	1147	269	102	38
811	1836	1564	272	135	47
2908	5240	5002	308	318	105
3871	6647	6385	262	377	128
5941	8083	7711	372	431	142

The big differences between the result counts is due to the matching of the word debugging by the DB and not by Solr.

Keyword “company” across all cores

Core index count	Total time (ms)	DB total (ms)	Solr Total (ms)	DB Results	Solr Results

153	322	80	242	42	32
592	1485	1188	297	174	141
811	1820	1549	271	226	183
2908	5277	4917	360	723	616
3871	7865	6970	895	932	794
5941	9464	`8607	857	1320	1147

Keyword “systems” across all cores

Core index count	Total time (ms)	DB total (ms)	Solr Total (ms)	DB Results	Solr Results
153	378	109	269	125	117
592	1717	1402	306	459	422
811	2264	1800	464	615	564
2908	669	5729	939	1836	1719
3871	8358	7251	1107	2300	2136

5941	12114	10526	1588	3319	3124

Keyword “algorithm” across all cores

Core index count	Total time (ms)	DB total (ms)	Solr Total (ms)	DB Results	Solr Results
153	320	84	236	42	24
592	1387	1085	302	136	80
811	1724	1445	279	179	106
2908	5078	4687	398	583	370
3871	6798	6203	585	718	461
5941	8942	8371	571	1101	726

The difference above is due to the occurrence of the word algorithms (which the database matches to algorithm and not Solr) and algorithm.

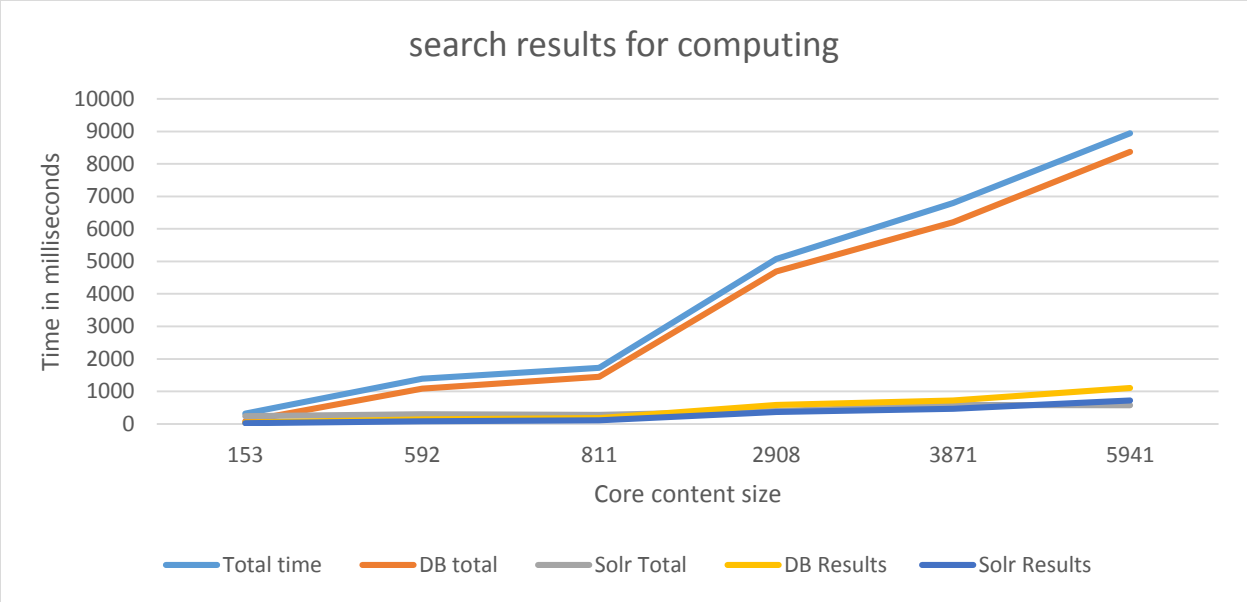


Figure 48 Search times for search term "computing"

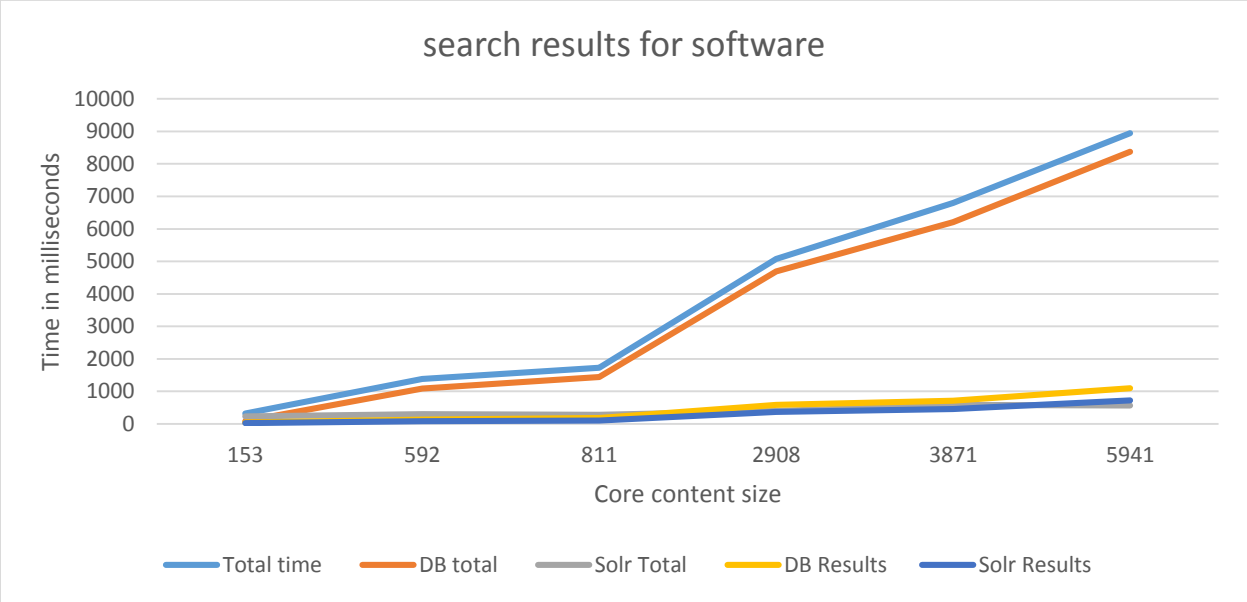


Figure 49 Search times for search term "software"

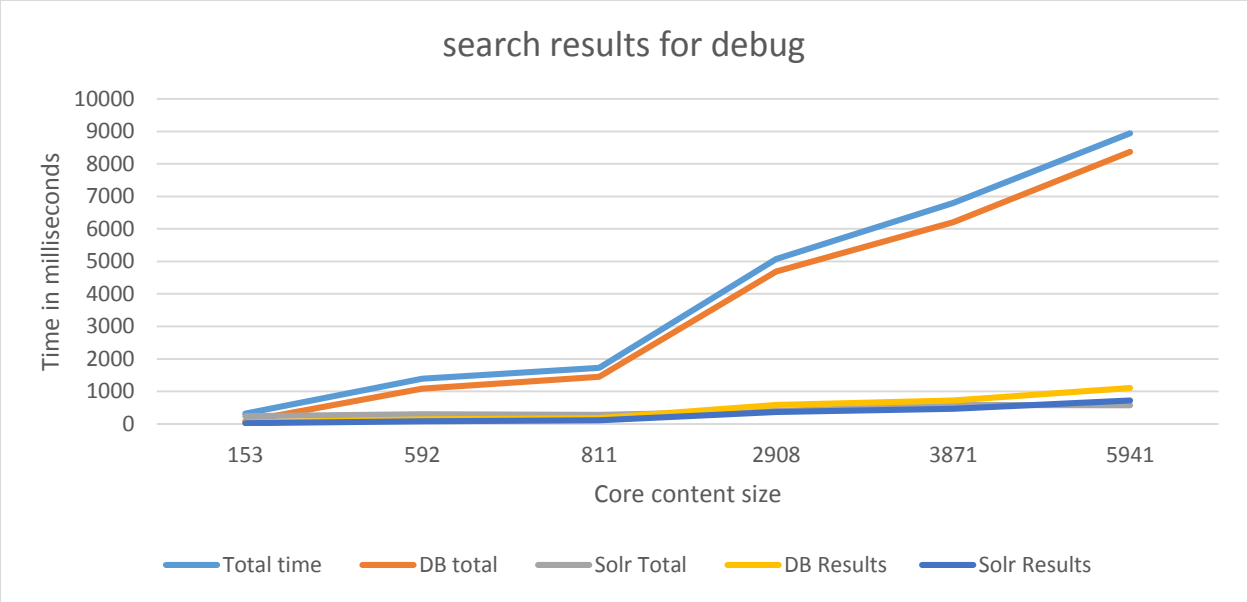


Figure 50 Search times for search term "debug"

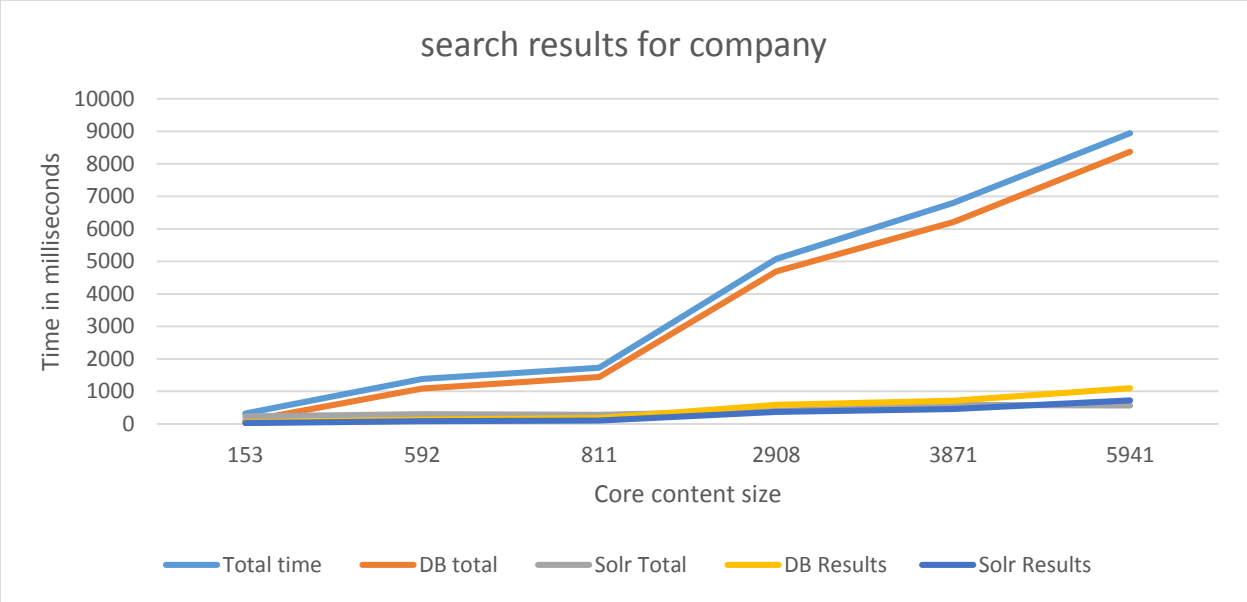


Figure 51 Search times for search term "company"

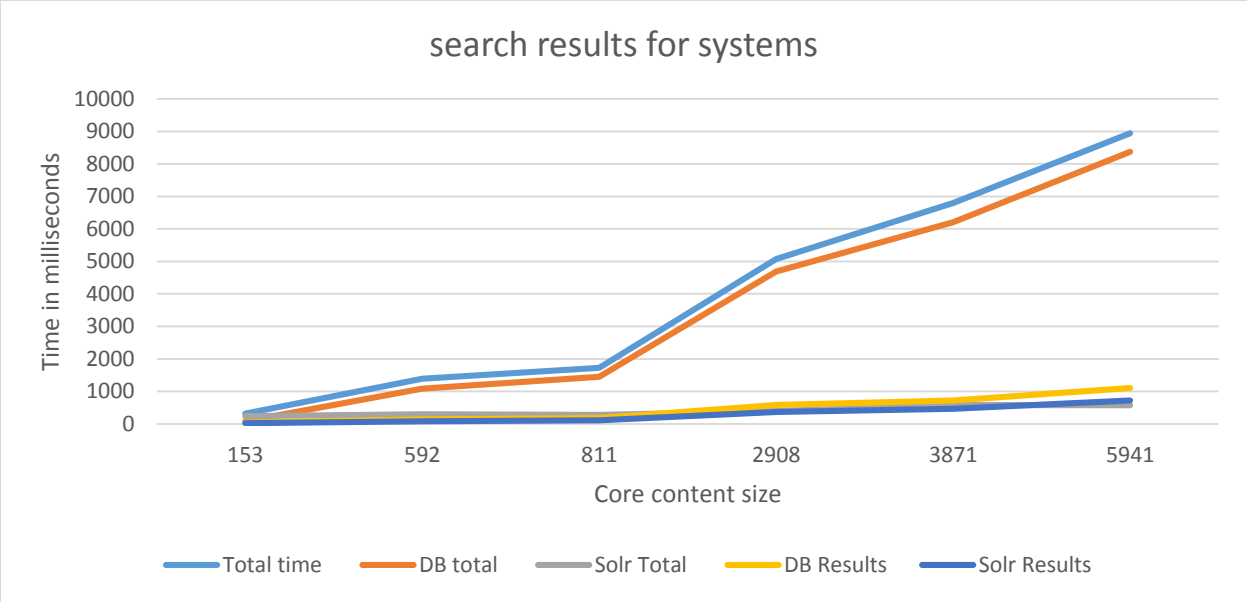


Figure 52 Search times for search term "systems"

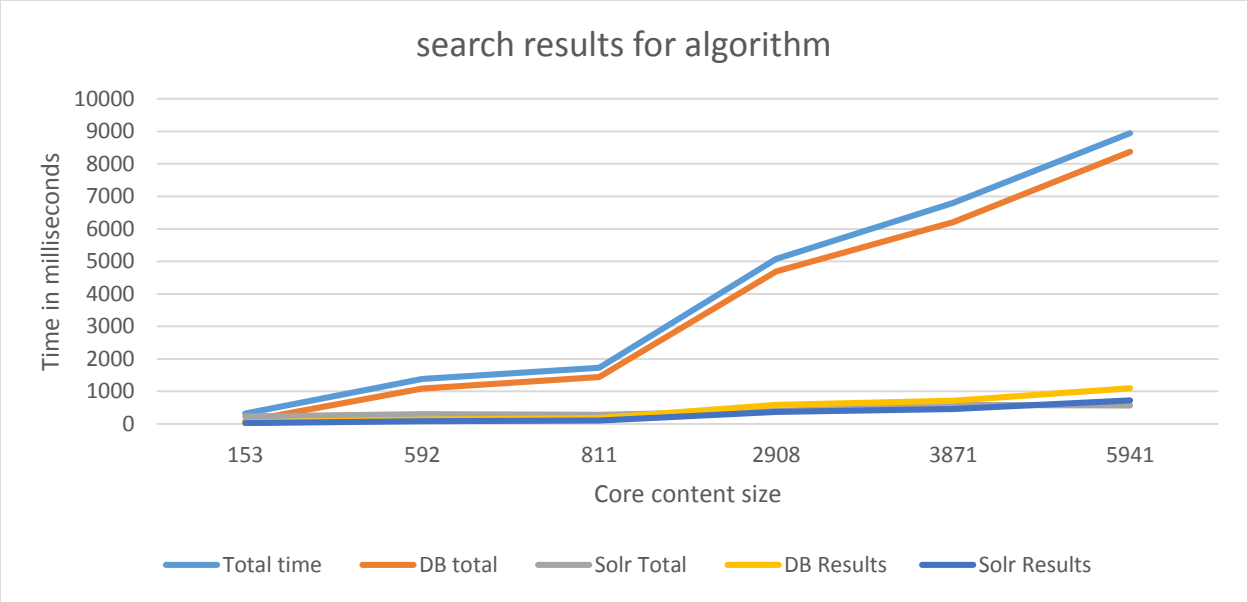


Figure 53 Search times for search term "algorithm"

Analysis of the search graphs shows that the size of the indexed content in the index cores has a relative linear relationship to the total time it take to search the core and processing. This applies to both the database and the file based index. Further breakdown shows the biggest changes are in the DB index fetching and processing times with the overall (blue) time closely following the DB total time.

5.3.5.1 Database index search times analysis

The database searching times grows at a much rapid pace than the equivalent Solr index. As the number of indexed documents grow in the core and the number of matches increase the database search time also increases. From the graph it can be shown that when the number of database indexes are increased by a factor of ten (592 in core 2 to 5942 in core 7) the time it takes to find the matches increases by a factor of 8 from 1 seconds to 8 seconds.

5.3.5.2 Solr index search time analysis

While the time for Solr to search and process also increasing with the number of indexed entries and the number of matches, the Solr core was affected by both much less than the database. While the increase of indexed documents increased by tenfold between core 2 and core 7 the time Solr took to process the search and result set only increased by a factor of 2 to 2.5. Showing the effectiveness of the file based indexes in dealing with volume.

5.3.5.3 Result set analysis

During the search evaluation the result count from the DB index and the result count from Solr did not correspond exactly. In most cases the DB had more matches than the Solr cores due to the default way of word matching. Solr by default implements whole word matches so for example the search term “Debug” is interpreted as different to “Debugging”. Whereas the DB index used partial word matching so that anything containing the term debug is returned via the search including “Debugging”.

To manage and implement governance in synonyms, Solr has a configuration property file that allows users to specify key word relationships. This functionality is not available in the DB search be default and would need an extra service layer to be added in order to achieve functionality matching Solr.

5.4 Analysis on adding additional cores and crawlers

5.4.1 Adding index cores in Solr

Adding additional cores would be the first step in an enterprise to expand a companywide search service. This can be achieved in solr relatively simply via the Solr core admin user interface (Grainger & Potter, 2014). In adding a core user is effectively duplicating all aspects of the core for example the schema of the core and the location of the indexes. Often than not (as during this experiment) cloning a core is desirable and can be done simple by copying an existing core folder into a new one and adding it via the core admin user interface.

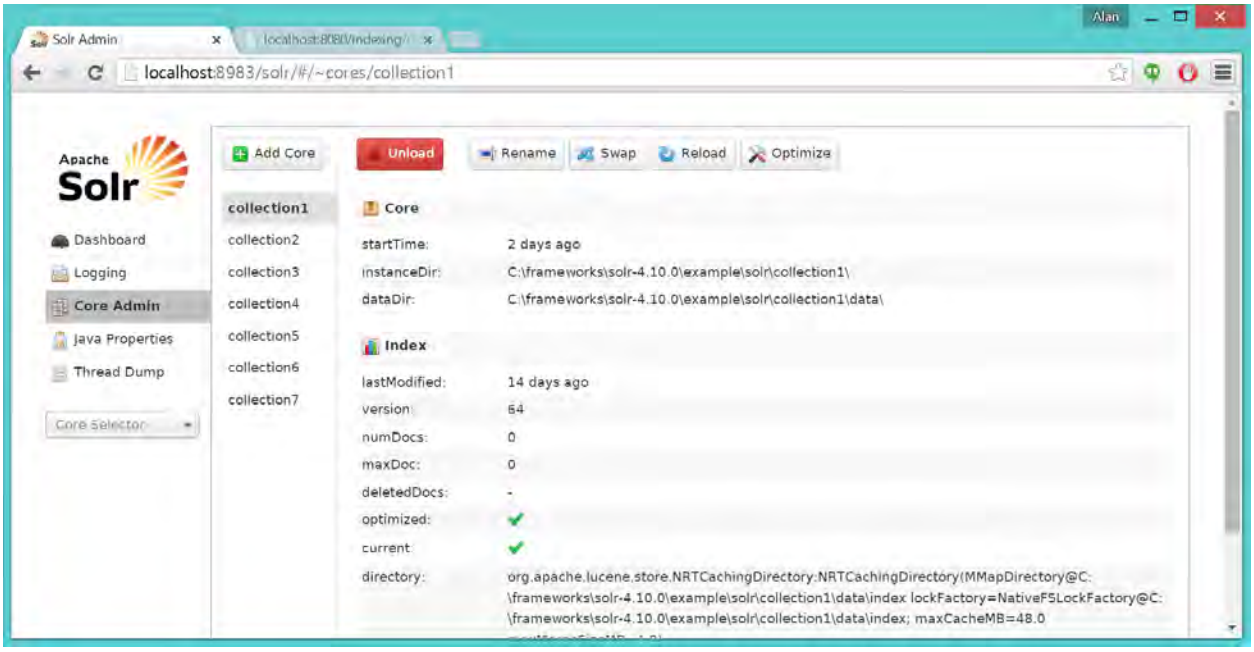


Figure 54 The Solr Core admin user interface

Once the core is added or cloned it can be used by SolrJ via a simple string trigger like such in the software layer:

```
public static QueryResponse getSolrQueryResponse(String searchString, String coreId)
        throws SolrServerException{
        String solrUrl = "http://localhost:8983/solr/collection" + coreId;
        SolrServer server = new HttpSolrServer(solrUrl);
        SolrQuery query = new SolrQuery();
        query.setQuery(searchString).setRows(9999); // everything
        return server.query(query);
```

```
}
```

To make the code reusable and easily manageable, the parameters to the SolrJ can include the core id so that a new core would not effect the underlying program logic for searching. A totally new core name can also be used if the "coreId" is externalized too.

If the core folder structure is copied directly including the index files under the structure then that core would be exactly like the source core with the same index content and schemas. During the life of this experiment many cores are duplicated in this way, with no program logic changes to use them as the core names form part of the Restful input parameters.

5.4.2 Adding additional indexing tables to MySQL and ORM

As mentioned in the proceeding chapters about the system architecture each of the index representing a core in Solr is represented in the DB as a Table. As such the manipulation of the index is far simpler to anyone with knowledge in SQL.

In order to create a new table users can simply clone the existing core tables and give it a new name. The slightly more complex part comes when this table is mapped into the java framework application via ORM. Once a table is created in the Database (which can be one of the many other supported ORM databases, MySQL is used here but ORM is vender independent) this table must be added to the ORM mapping as a Java file. Luckily for the purpose of cloning an existing table as in this experiment the Java file looks rather simple, it does not even need any attributes as it inherits it from a base class representing all the common shared attributes in the cores used in the experiment.

```
package domain;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "core7")
public class Core7 extends CoreBase{

	/**
	 *
```

```
     */
    private static final long serialVersionUID = 1L;

}
```

Core 7 shown here extends a CoreBase.java class that contains every field necessary to index the standard nutch crawl output.

In order to use this Core it is not necessary to add its reference to anywhere else in the logic that already processes the base table because the core name will be part of the Restful input and the HQL dealing with the table selection can dynamically use the table name in its queries (refer to source code for the full call stack).

5.5 Analysis on adding additional search/data sources to the architecture

If one is to assume operating from an enterprise perspective in order to add in an additional data source to be index the highlight of this process would be as follow with the insight gained from the experiment:

1. Deploy a crawler instance such as nutch. Multiple crawler can be operated independently and on different hosts if necessary. The configuration of each crawler can be unique and the output schema can be different. The key attributes for a federated searching service is the data source connection details (i.e. the database to push the index) and the Solr endpoint URL. Both of these attributes is via IP so the DB server and Solr servers can be different as long as the IP connections are reachable.
2. Deploy the Solr Core and/or the DB index table to the corresponding hosts (they can be on different IP addresses). The most important thing is that both schemas must match the output schema of the crawler.
3. Update the search federation service
    a. Adding the reference to the new DB table via ORM
    b. Adding the new Solr core reference in the Solr lookup process (This can be externalized into stored files for example XML files)
    c. Adding into the result processor any additional logic needed to process the result set form the new core.

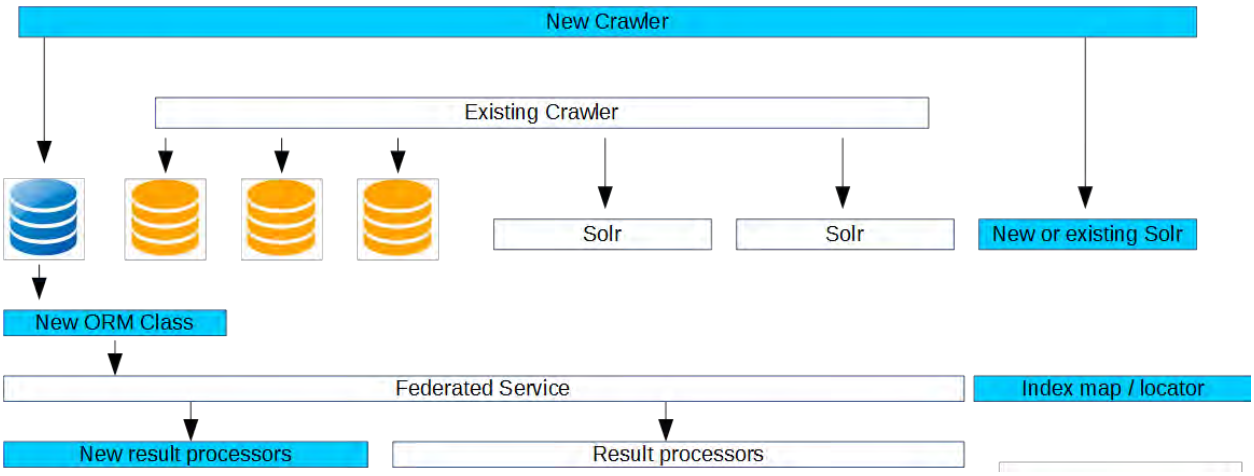


Figure 55 The new components added for a new search source (new components highlighted in blue)

6 CONCLUSION AND FUTURE WORK

6.1 Introduction

The initial concept of this thesis started as a very broad statement on how to have a crawling and indexing system that is flexible, which can be quickly deployed and work with a target source dataset. The dataset its self can range from intranet, internet to local resources. When the literature review investigations started, this idea then quickly expanded into seeing would it be possible to use only open source frameworks to achieve this goal.

Further investigation into how indexes operate brought up the question of how the traditional databases can be used to index data along with file based solutions. Their performance and integration possibilities also warranted investigation.

This experiment demonstrated that it is possible to use open source software frameworks to solve the problem of creating a federated search service to serve the needs of an enterprise as discussed in the challenges of enterprise search (Hawking, 2004). While the out of the box implementations might need adjustment and further development, the capabilities of the full search stack used in this experiment contains the foundation necessary to facilitate such modifications. The design and development of the software framework took up a large portion of experimental time due to the technical nature of integrating and configuring the software stack. Without an effective federated searching framework it would not be possible to directly compare and contrast the performance of the file index and the database index and to explore and evaluate the modular nature of the indexing system components.

Using the architecture design in the experiment, it can be seen that the modular nature of the components makes distributed communications with each component possible thereby allowing organizations with different offices spread across geo locations to achieve the goal of having a centralized search service (Regli, 2010). The modular nature also easily facilitates scaling where additional search cores/indexes and crawlers can be added without major architecture changes. This solves some key challenges in the existing literature such as integrating old/new systems, generating indexes from both exposed rich content and databases and serving up efficient and speedy searches (Singh, et al., 2007).

6.2 Key finding and discovery

Belong are the key finding and discoveries made during the experimental setup and execution stages.

6.2.1 Overall system architecture

The key strengths of the proposed system can be identified as two primary findings. The first is the loosely coupled modularity of the components and the second being the language independent interface between the components using the common HTTP (hypertext transport protocol) standard.

By having a loosely coupled modules representing the crawler, indexer and service all running on different IP (internet protocol) addresses each module can co exists without a direct dependency on the physical location of any of the other modules. This lends particularly well to scaling and deployment of modules across different geographic zones.

An example of this is proven during the experiment where it was possible to deploy multiple crawlers independently at the same time oblivious to each other to crawl Wikipedia pushing to different database tables since they each had their own database connection configurations. The database connections are driven by IP addresses which meant that as long as the IP address is visible to the host running the crawlers it was possible to access the database irrespective of the location of the databases.

The Solr storage is also exposed via IP addresses and therefore present the same location independence as the databases. This made the crawlers highly flexible by only needing to know the target URLs IP address, the database IP address and the Solr IP address to function.

The federated service itself is also flexible in the same context. It only needs access to the same database IP addresses and the Solr IP addresses to operate.

6.2.2 The technologies

Core evaluations that did not fall in line with initial estimation for the technology started with the speed it took Nutch to crawl webpages. Referring back to the figures in the evaluation section for the time it took to crawl more and more layers, it is noted that by default Nutch respects the target URLs and enforced a friendly wait time in order not to over load the target website. This can be adjusted accordingly in the configuration. The ideal scenario would be to provide a staging site which many companies do have before publishing and allow the crawler to access this site instead of a live site for the indexing during non-use hours. In such cases the crawler would be configured to operate at max speed.

The speeds that it took ORM (object relational mapping) framework (Hibernate) to do a query against the databases as the number of the dataset increased also increased almost linearly showing that optimization in this space is necessary depending on the volume of the expected results being indexed.

Expecting a heavy overhead in the Java logic for post processing of the result set proved to be incorrect as the timing in the evaluation section shows the time it took to post process the result and formatting them into the standard JSON (Java Script Object Notation) was surprisingly fast with almost no increase as the number of results increased.

6.2.3 The Cores

The most significant finding of using the Solr cores and the database cores are that despite the overlap in the role of the Core table and the Solr cores, it is not expensive computationally to create both indexing resources. This means that the DB solution and the File solution can be complimentary to each other especially if the aforementioned method of linking external database sources as index cores is utilized.

In a framework such as Nutch, it is deployed by default with a self contains data storage mechanism that acts as database while it is in the process of running the crawls. This data storage is necessary for storing information in relation to the crawling session and the data of URLs before the indexing can be created to be pushed into Solr. The overhead of moving this store

from the internal storage into a database core is minimal within the new Nutch framework thanks to the inclusion of Gora connectors (Apache Software Foundation, 2015), with this in mind it is even more beneficial to have both index created at the same time during a crawl to complement each other in the search process.

The advantages discovered while integrating the database index into the federated search service was the ease of management using SQL to manipulate the indexes directly unlike the Restful API of Solr. The nature of using a relational database for indexing also opens the exciting possibility of additional integration option for searching. One example would be using an index table that is not created from crawling but actual database table (or logic view(s)) representing an external system. In such cases the system can be integrated into search via the use of ORM mapping and negates the need for the index creation and crawling completely. The overhead is of course with multi-dimensional tables it is not so simple to identify the main tables that should be included for the search at the first place and even then it would be necessary to create and maintain additional logic to ensure the right information is served when the results are found.

6.2.4 Crawling and indexing the data

Accessing the data was straight forward in the experiment due to the nature of the data presented via the HTTP protocol. Any data that can be made visible via the HTTP protocol can be access easily by a crawler. Any potential sub URLs that cannot be accessed by the crawler would be skipped.

Due to the nature of the IP oriented addressing of content Crawlers can target any resource that can be exposed via IP addresses.

The default crawling strategy of Nutch (Breadth first crawl) (Hafri & Djeraba, 2004) worked out well for the crawling of a site such as Wikipedia. With the max links per depth set to -1 it would have been possible to crawl each page at all depths. For a site such as Wikipedia however this was not possible due to the recursive nature of the site pages (high number of bidirectional referencing) and the time it takes to do a large crawl (refer to the evaluation section for URL indexed vs time).

The default behavior of the crawling unit accessing or indexing images is that it would be indexed as a foreign reference. This meant that while it is accessing the page with the image it would identify that the image is foreign resource and be pushed into its own index as a URL to the resource. This is the expected behavior from the start.

6.2.5 Searching and Performance

The experiment also provides an indication of the baseline performance that is achievable using the frameworks in areas such as crawling, index creation and searching. The numeric measurements indicates that a dedicated file based system is still superior to DB lookup for indexing when it comes to creating the index and searching, as supported by the design goal of Solr (Grainger & Potter, 2014).

6.3 Key contributions

This section describes the key contribution this thesis should add to the existing body of knowledge around indexing architectures and performance evaluations.

- The modular architecture presented in using the Nutch Crawler (Apache Foundation, 2015), MySQL (Oracle Corp, 2015) Database and Solr (Apache Software Foundation, 2015) file based index. How the proposed IP interfaced services solves some of the problems of system location integration.
- The framework for a federated search service presented using Java technologies such as Hibernate (JBoss, Redhat, 2015), Spring (Pivotal Software, 2015), Apache Gora (Apache Software Foundation, 2015) and Jersey API (application programming interface) (Oracle Corp, 2015).
- The programming examples for the configuration of each of the technologies working together.
- An analysis of the configuration and performance of the Apache Nutch open source crawler.
- An Analysis of the configuration and performance of the Apache Solr File indexers.

- A comparison of using MySQL schema as a comparative Nutch schema index versus the equivalent Solr schema index.

6.4 Limitations and Future work

Below is an analysis of the key unexplored limitations of the experiment within the thesis and the future work that can be done in order to extend the proposed framework and to answer additional outstanding queries about the existing framework.

6.4.1 NoSQL indexing evaluation

During the experiment it is discovered that the support of the Gora connector (Apache Software Foundation, 2015) between the crawler and the underlying database also supported non-relational database such as Cassandra (Apache Software Foundation, 2015) NoSQL databases. It would be interesting to evaluate the performance of a NoSQL database versus the Solr file based index instead of the relational MySQL. Due to the nature of file based data store of NoSQL databases it would be assumed that the performance lead of Solr over NoSQL would be less than that of a relational database.

In order to integrate a NoSQL database it is necessary explore the configuration within the Nutch crawler instance to use the Gora connector (which is modular and interface agnostic) by setting the IP address to the NoSQL instance. As long as the NoSQL service is visible via an IP Address the crawler will be able to use it swapping the MySQL functions used in the experiment.

6.4.2 Security framework integration

The current existing framework provided provide no component to introduce security applications. This should be addressed in the future by introducing a security framework into the overall architecture allowing security to be implemented while preserving the modular design of the framework itself.

Ideally the security framework should work within the Java service layer by filtering results via meta-data contains within any one index. The challenge posed by this is the way which such

security meta-data is generated from the crawling of the original content. Experiments will be needed to determine the best way to generate security meta-data which will differ depending on the content type and also the storage structure within an enterprise of organization.

6.4.3 Optimizing and redesign of the database indexes

The default schema for the construction of the table in MySQL for storing the indexed data from Solr is not optimized via indexes. This has a significant performance penalty when comparing to databases that has optimal indexes. It would be interesting to see how much the performance of the database index can be improved upon if the table structure are to be enhanced with more than one table per index and the introduction of indexes on the content (McHugh, et al., 1998).

The default schema used in the experiment did not support indexing on the key content column due to the column definition being a blob. An experiment can be carried out to split the content into a separate table (or possibly several other tables) in order to reduce the word count in the default blob so that a quicker search would be possible. Be splitting the content up into smaller character field it should also be able to index the field within the database schema itself.

6.4.4 Content parser integration for proprietary file formats

The site Wikipedia used in the experiment did not contain any proprietary binary text files such as Microsoft word or adobe pdf. Typical larger organizations would contain large quantities of both file types. In order for a crawler to read within these files experiment should be conducted to evaluate the apache manifold connectors framework (Apache Software Foundation, 2015) which are designed to work with other apache technologies for parsing special proprietary files such as Microsoft word and Microsoft excel and adobe pdfs.

6.5 Summary

While it is easy to evaluate in terms of speed the response times from a file based index system against a database base index system this experiment has shown that they both have their strengths which can be used in combination for a better search service using an overhead framework to help with the result consolidation and post processing. The processing overhead of

having both indexes created is minimal during the experiment which further encourage using both index types within one service.

Another interesting conclusion on building the federated search framework showed that existing Apache family packages has great designs in terms of modularity which is evident in the way that each technology works independently but sharing a common interface transport standard (IP). Using this a robust and scalable federated indexing service can be deployed without software licensing costs.

7 Bibliography

Apache Foundation, 2014. *Apache Tika Formats.* [Online]
Available at: http://tika.apache.org/1.4/formats.html

Apache Foundation, 2015. *Apache Nutch Downloads.* [Online]
Available at: http://nutch.apache.org/downloads.html

Apache Software Foundation, 2015. *Apache Cassandra.* [Online]
Available at: http://cassandra.apache.org/

Apache Software Foundation, 2015. *Apache Gora.* [Online]
Available at: http://gora.apache.org/

Apache Software Foundation, 2015. *Apache Solr.* [Online]
Available at: http://lucene.apache.org/solr/

Apache Software Foundation, 2015. *Apache Solr Resources and Tutorials.* [Online]
Available at: http://lucene.apache.org/solr/resources.html

Apache Software Foundation, 2015. *Apache Tomcat.* [Online]
Available at: http://tomcat.apache.org/

Apache Software Foundation, 2015. *What is Apache Manifold CF.* [Online]
Available at:
http://manifoldcf.apache.org/en_US/index.html#What+Is+Apache+ManifoldCF%3F

Baeza-Yates, R., Castillo, C., Marin, M. & Rodriguez, A., 2005. *Crawling a country: better strategies than breadth-first for web page ordering.* New York, WWW '05 Special interest tracks and posters of the 14th international conference on World Wide Web.

Bishop, T., 2008. *Bill Gates e-mail re Movie Maker.* [Online]
Available at: http://blog.seattlepi.com/microsoft/2008/06/24/full-text-an-epic-bill-gates-e-mail-rant/

Chernov, S. et al., 2006. *Enabling Federated Search with Heterogeneous Search Engines -- Combining FAST Data Search and Lucene.,* Hannover: Vascoda Report.

Ding, B. et al., 2012. *Optimizing index for taxonomy keyword search.* s.l., SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.

Duda, C., Frey, G., Kossmann, D. & Zhou, C., 2008. *AJAXSearch: Crawling, Indexing and Searching Web 2.0.* s.l., Proceedings of the VLDB Endowment.

EMC, 2013. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things.* [Online]
Available at: http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

Fagin, R. et al., 2003. *Searching the Workplace Web,* s.l.: IBM Almaden Research Center.

Forbes, 2014. *The most valuable brands in the world.* [Online]
Available at: http://www.forbes.com/powerful-brands/

Giusti, M. R. D., Lira, A. J. & Oviedo, N. F., 2010. Service Cloud for information retrieval from multiple origins. *International Conference on Engineering Education ICEE.*

Giusti, M. R. D., Lira, A. J. & Oviedo, N. F., 2010. Service Cloud for information retrieval from multiple origins. *International Conference on Engineering Education ICEE-2010 July 18.*

Google inc., 2015. *Google features help.* [Online]
Available at: https://support.google.com/websearch/?hl=en#topic=3378866

Google inc., 2015. *How google Search Ranking Algorithm Work.* [Online]
Available at: http://www.google.ie/insidesearch/howsearchworks/algorithms.html

Grainger, T. & Potter, T., 2014. *SOLR in Action.* 1st ed. s.l.:Manning.

Hafri, Y. & Djeraba, C., 2004. *High performance crawling system.* New York, MIR04 Proceedings of the 6th ACM SIGMM international workshop on Multimedia information retrieval, pp. 299-306.

Hawking, D., 2004. *Challanges in Enterprise Search.* Australia, ADC '04 Proceedings of the 15th Australasian database conference.

JBoss, Redhat, 2015. *Hibernate Technologies.* [Online]
Available at: http://hibernate.org/

Kennedy, N., 2014. *Google processes over 20 petabytes of data per day.* [Online]
Available at: http://www.niallkennedy.com/blog/2008/01/google-mapreduce-stats.html

Livne, O. E., Schultz, N. D. & Narus, S. P., 2010. *Federated querying architecture for clinical & translational health IT.* New York, IHI10: Proceedings of the 1st ACM International Health Informatics Symposium.

McHugh, J. et al., 1998. *Indexing Semistructured Data,* s.l.: Standford.

Najork, M., 2009. *Encyclopedia of Database Systems.* California: Springer US.

Nancy Blachman, J. P., 2014. *How google Works.* [Online]
Available at: http://www.googleguide.com/google_works.html

Olston, C. & Najork, M., 2010. Web Crawling. *Foundations and Trends in Information Retrieval,* 4(3), pp. 175-246.

Oracle Corp, 2015. *Jersey Framework.* [Online]
Available at: https://jersey.java.net/

Oracle Corp, 2015. *MySQL.* [Online]
Available at: http://www.mysql.com/

Palma, F., Farzin, H., Guéhéneu, Y.-G. & Moha, N., 2012. *Recommendation System for Design Patterns in Software Development: A DPR Overview.* NJ, RSSE '12 Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering.

Pivotal Software, 2015. *Spring Frameworks.* [Online]
Available at: https://spring.io/

Rafalovitch, A., 2013. *Instant Apache Solr for Indexing Data How-to.* 1st ed. s.l.:PACKT.

Ranjan, J., 2009. Business Intelligence - Concepts, components, techniques and benefits. *Journal of Theoretical and Applied Information Technology.*

Regli, T., 2010. *Enterprise Search vs Federated Search.* [Online]
Available at: http://www.realstorygroup.com/Blog/1854-Enterprise-search-versus-federated-search

Singh, A., Srivatsa, M. & Liu, L., 2007. *Efficient and Secure Search of Enterprise File Systems.* Utah, Web Services, 2007. ICWS 2007. IEEE International Conference on, pp. 18 - 25.

Woods, W. A., 1997. *Conceptual Indexing - A Better Way to Organize Knowledge,* California: Sun Microsystems.

Yasser Ganjisaffar, 2014. *Github - crawler4j.* [Online]
Available at: https://github.com/yasserg/crawler4j