CloudSearch: A Custom Search Engine based on Apache Hadoop, Apache Nutch and Apache Solr

Lambros Charissis, Wahaj Ali
University of Crete
{lcharisis, ali}@csd.uoc.gr

Abstract. Implementing a performant and scalable search engine entails the need for infrastructure and specific knowledge. In order to bypass the challenge of implementing an own search engine, companies can use the services of custom search engine providers. CloudSearch provides a fully-managed search service and is based on the Apache open-source projects Hadoop, Nutch and Solr. It was designed to be scalable, easy to integrate and to provide high quality search results.

Keywords: Custom Search Engine, Hadoop, Nutch, Solr, Zookeeper

1 Introduction

Today most companies use their websites as the main information channel to the customers. On their websites, companies describe themselves, their services and their products. There are even cases where the website itself is the main service that is offered by a company. One example would be a newspaper, where the offered service is the information provided through the internet. One important aspect of this trend is the usefulness and usability of the information provided on a website. On the one hand companies need to offer the information that is demanded by customers, on the other hand, once the information is provided, it has to be found by the customer. Besides logical menu structures or the layout, search functionality is one contribution to help visitors finding what they are searching for. In this paper we will introduce the custom search engine CloudSearch, a fully managed and scalable search service that can be used by companies to make their website searchable without providing their own search functionality and infrastructure. First we will describe some related work. After that we will focus on the notion of a custom search engine and will illustrate some requirements that have to be met by a custom search engine. In the next two chapters the architecture and the implementation of CloudSearch will be specified in
detail. Finally we evaluate CloudSearch in respect to the ease of integration, quality of search and scalability.

2 Related Work

There are different commercial providers of custom search services. The most popular example of a custom search engine is Google Site Search. Google Site Search is an edition of Google Custom Search looking to create a highly customized and Google-hosted site search solution. Newly also Amazon offers a custom search engine called Amazon Cloud Search. Amazon Cloud Search is fully-managed search service in the cloud that allows customers to easily integrate fast and highly scalable search functionality into their applications. Amazon Cloud Search is currently in the Beta stage [1]. In April 2012 also Microsoft moved its Bing Search programming interface to the Windows Azure Marketplace and turning it into a paid subscription service [2]. Besides Google, Amazon and Microsoft there is a variety of smaller companies that offer custom search services. Two examples are [3] and [4].

3 Custom Search Engine

In this chapter we want to give a brief introduction to the idea of a custom search engine. One way to make information on a website better findable is adding search functionality to it. With an increasing amount of content and traffic the performance and quality of search can easily become a bottleneck. The company has to provide the infrastructure to firstly index the content periodically and secondly process and reply to all the search queries performed by the users of the website preferably with a low response time. As all this needs infrastructure and specific knowledge, many companies avoid dealing with their own implementation of a search engine by using the service of a custom search engine provider.

In the first step a customer has to register at a search service provider informing him about the websites that he wants to be searchable. After this the customer is provided
with some HTML and JavaScripts that let him integrate a search box and a search result page into his own website. After the registration, the search service provider starts crawling and indexing the websites specified by the customer. From this point in time all search requests that are performed by visitors of the customers’ website, are carried out by the search service provider.

4 Requirements

Before we address the architecture and implementation, we will give an overview of the requirements that we had in mind while developing CloudSearch.

4.1 Scalability

A custom search engine has to be scalable in two different dimensions. In respect to the size of the website as well as in respect to the traffic that will be created by the search requests.

![Fig. 1. Two dimensions of scalability [5]](image)

Ideally the two dimensions can be scaled independently. For a small website with a huge amount of traffic, more resources should be used in order to process and reply to search requests with low latency. Even if the traffic is high, the same resources should be used for the crawling part as the traffic is independent of the size of the website. For a big website with little amount of search requests and the configuration of an index that is updated on a daily base, more resources have to be used in order to crawl and index the website in the give time frame. The resources that are in use for the processing of the search queries should not change if the traffic is constant.
4.2 Quality of Search

A further requirement that we had in mind while developing CloudSearch is the quality of search. Whereas the scalability is a performance figure expressed in numbers, the quality of search is a qualitative requirement that addresses the search result page and comprises different aspects. One of them is the relevance to the query. CloudSearch should rank the results that are most relevant to a particular query higher than the results that are less relevant. Besides the plain search results, CloudSearch should have some functionality that supports the user while searching.

4.3 Ease of Integration

From the perspective of a customer of CloudSearch the registration and integration process should be preferable easy. In order to integrate CloudSearch the customer shouldn’t need more than basic HTML knowledge. The customer should have a means to specify the URLs that have to be crawled and indexed by CloudSearch. One of the features is the search across multiple URLs. So the customer has the possibility to specify more than one URL in the configuration. The user will search across all these URLs over one interface. After the integration has succeeded the customer shouldn’t have any maintenance effort.

5 Architecture

The tasks of a custom search engine can be divided into three main categories: Crawling, indexing and processing search requests. As one of the requirements is to scale the crawler and search system independently from each other, we used Apache Nutch on top of Apache Hadoop for the crawling part and Apache Solr including Apache Lucene as the index and search system. Through the usage of these technologies we can assign a different number of VMs to Apache Hadoop and Apache Solr and can consequently scale the two systems independently. In the following diagram one can get an overview of the CloudSearch architecture.
For this project, we had a total of six virtual machines running on Amazon EC2. All the machines were the same in terms of specifications (micro-instance). Three of the VM’s composed the Hadoop cluster, two of them were running SolrCloud, while the last one was running our dashboard application. On all the VMs we used Ubuntu Server 13.04 as the operating system.

5.1 Hadoop Cluster

Hadoop provides us with a distributed platform for running Nutch jobs in a scalable manner. Apache Nutch is an open source crawler and parser. We have used three nodes for deploying Hadoop. One node, the master, runs the namenode and jobtracker, while the other two nodes, the slaves, run the task tracker and datanode. Hadoop is responsible for running the Nutch jobs in the MapReduce paradigm. The result of the Nutch job is the crawl database that is passed to SolrCloud for indexing afterwards.

5.2 Apache Solr

Apache Solr is responsible for indexing the document created by Apache Nutch and make them searchable. The SolrCloud component comprises two Solr instances and ZooKeeper. Zookeeper is responsible for distributed configuration and coordination between the different Solr nodes. Although it would be ideal to run Zookeeper in a separate machine, due to limited resources we run it in the same instance as the Solr master node. In order to improve the performance of indexing and to avoid that indexes are approaching the physical limitations of the machines, we distributed the index across two Solr servers. To distribute an index, you divide the index into partitions called shards, each of which runs on a separate machine. Solr then partitions
searches into sub-searches, which run on the individual shards, reporting results collectively. The architectural details underlying index sharding are invisible to end users, who simply experience faster performance on queries against very large indexes [6]. Since we have only two Solr instances, we also do not replicate them.

5.3 Apache Nutch

Apache Nutch provides us with web crawling capabilities. We use Nutch to crawl the domain that the user specifies. It provides us with an automated framework, carrying out tasks such as fetching, maintaining URL list, removing duplicate links and broken links. It is also closely tied to the Solr project, since both are maintained by the same foundation (Apache). Therefore, Nutch supports Solr out of the box and integration between the two is quite straightforward, to the extent that we just have to specify the URL of the Solr collection to which we want Nutch to pass the data to.

5.4 Application Dashboard

The application dashboard is the CloudSearch web application that enables interaction with customers. A customer that is already using CloudSearch can use the application dashboard as a control panel in order to retrieve his configuration, the crawl status and a health report covering the Hadoop cluster and SolrCloud. For new customers the application dashboard provides a registration service and a step by step instruction to configure and integrate CloudSearch.

![CloudSearch WebApp](image)

**Fig. 3. CloudSearch WebApp**

The CloudSearch web application is developed in PHP. The application is running on an Apache 2.4.4 web server and is using a MySQL 5.5.29 database. Both, the control panel and the registration service have to integrate with SolrCloud and Hadoop. For further details see the implementation chapter.
6 Implementation and Configurations

6.1 Hadoop

Hadoop is required to run Nutch jobs in a distributed environment. We setup up the following variables in Hadoop:

- **dfs.replication:** the number of nodes data should be replicated to. We set this to 2, that is replicate the dfs data on both the slaves.
- **mapred.map.task:** the number of map task. Set to 2 (the number of slaves)
- **mapred.reduce.task:** the number of reduce task. Also set to 2.
- **mapred.compress.map.output:** weather the output of the map task should be compressed before sending it over.

Increasing mapred.reduce.task and mapred.map.task to a value greater than 2 causes Hadoop to start throwing more errors and more worker VMs are instantiated. Hence we found that in our case 2 was the most optimal. We set the last variable mentioned above to increase the data movement between the different nodes to a minimum.

6.2 Nutch

We use Apache Nutch to fetch documents from the internet, parse them, extract the required data from HTML pages and then pass them on to SolrCloud for further indexing. For every crawl we use the following variables:

- **url:** A list of seed urls. Usually we just specify one. These are specified in a file which is passed to the crawler.
- **url-filter:** Determines by means of a regular expression which urls to fetch. We use this to ensure the crawl does not go outside the current domain.
- **index.parse.md:** Specifies the metatags that we want to extract from the webpage.
- **fetcher.server.delay:** Delay between each successive request to the same server. In our case we have set this to 0.
- **fetcher.threads.fetch:** Number of concurrent fetches, set to 20.
- **depth:** Indicates the link depth from the root page that should be crawled. For the final demo, we set this to 15.
- **topN:** Determines the maximum number of pages that will be retrieved at each level up to the depth. For the final demo, this was set to the same value as depth.
- **crawldb:** Is a folder that contains the information from previous crawls, such as the urls that were fetched, the time, outgoing links etc. For every crawl we need to ensure that we are storing the last crawled data so that we can utilize it in later crawls to the same website.

Other than the variables url, depth, topN and crawldb the rest are all specified at build time.

To run Nutch on Hadoop, we have to build the file using Ant, and then to subsequently use the job that the build generates. One challenge here was that we wanted to
crawl different urls with different configurations. Although we don’t have to define the urls at built time, we do have to configure Nutch for metatags before we do the actual build. Since in our system, each user has the liberty to define unique metatags for different urls, this requires Nutch to have different configuration for crawling each url. The solution we came up was to build Nutch once for each url. What this means is that each Nutch url has a corresponding job file. The directory structure we chose looks like this:

![Nutch directory structure](image)

**Fig. 4. Nutch directory structure**

According to our system each user can have multiple urls. Therefore, each customer has his own directory, which has a subdirectory each for the urls he wants to index. This subdirectory is responsible for storing the Nutch job corresponding to the url, which is specified in seed.txt. Both the folder containing the seed file and the Nutch job file are to be passed to Hadoop when the crawl is to be run. Therefore created a cron job that runs once every day, goes into each subdirectory to fetch the job file and corresponding seed file and then runs them on Hadoop.

Some of the important variables that we configure in Hadoop are:

- **mapred.map.tasks**: This is the number of map task, which in our case is configured to the number of slaves.
- **mapred.reduce.tasks**: This is the number of reduce task. Similar to mapred.map.tasks this is also set to the number of slaves.
- **dfs.replication**: The number of file replications. We have set this to 2, since we have two data nodes. This means each file we add gets replicated to both the slaves.

To run Nutch we need to pass the Nutch job that we built earlier. We also have to specify variables such as depth and topN as command line arguments to Nutch. As
mentioned before the cron jobs are responsible for running these commands. Lastly, we tried to increase the number of map.task and reduce.task, however we found that this causes a large number of errors due to lack of heap space.

6.3 Index and Search System

For indexing and querying the system we deploy two instances of Solr, running on two separate EC2 micro instances. Solr offers us a REST like API to query, monitor, and manage the servers and is ideal as it takes care of the internals of indexing.

Cores

Solr comes with new capabilities which allow distributed indexing, replication and search capabilities. To achieve this it works in tandem with Zookeeper for cluster configuration and coordination, but requires little configuration on our part. To describe how we take advantage of the distributed capabilities of Solr, we require that we understand the concept of Solr Cores. From the Nutch website Solr Cores are described as follows:

“On a single instance, Solr has something called a SolrCore that is essentially a single index. If you want multiple indexes, you create multiple SolrCores. With SolrCloud, a single index can span multiple Solr instances. This means that a single index can be made up of multiple SolrCore’s on different machines. We call all of these SolrCores that make up one logical index a collection. A collection is an essentially a single index that spans many SolrCore’s, both for index scaling as well as redundancy.”

http://wiki.apache.org/Solr/SolrCloud

In our system, we employ an approach where each Solr core is distributed between the two instances of Solr. Each Solr instance contains part of the index. For each customer we create a new Solr Core, i.e a new index collection. With the latest version of Solr this can be done dynamically at runtime without having to restart the server. Such an architecture ensure that if one of the shard fails, we are still able to access half of the indexes, i.e some results will still be returned. To add fault tolerance it is also possible to replicate the shards (i.e parts of the index), however due to the limited number of VM’s we did not go forward with this. One short coming of the current approach is that Zookeeper is now a single point of failure. If Zookeeper fails, our show system becomes inaccessible. However it is also possible to replicate the Zookeeper instances, between different machines.

Schema

To index within Solr we need to define a schema file, to tell Solr what fields need to be index and stored, and their data type. Some of the important fields that we store are
URL, content, title, domain and author. There are other fields, but these are just some of the important ones.

Since we allow customers to define their metatags to index for each url, we need a way to be dynamically add these metatags to the schema. One way is to add them to the schema files and restart the server, but that is not the approach we are looking for. The dynamic field functionality allows to do this without having to actually change the schema or restarting the server. Using dynamic fields we are able to tell Solr to store all fields which match a certain name. In our case we use the following declaration:

\[
\text{<dynamic Field name="metatag.*" type="text_general" indexed="true" stored="true"/>}
\]

Therefore, any field whose name has a prefix “metatag” will get indexed and stored in the system.

6.4 Application Dashboard

In order to make CloudSearch useful to customers it has to offer an interface where the customer can register, specify his configurations and get instructions how to integrate CloudSearch into his own website. The application dashboard is implemented in PHP and uses a MySQL 5.5.29 database to store user information persistently. For the look and formatting of the web application we used the Bootstrap framework [7].

Registration

Every new user undergoes the following 4 steps.

- Registration
- Add URLs
- Add Metatags
- Receive Code

![Fig. 5. Registration steps](image)

In order to use the CloudSearch service the customer has to create an account. In this step the new user specifies his name, his email and a password. After creating the account the user gets directly redirected to the page where he has to add the URLs that have to be taken into account by CloudSearch. The user can add an arbitrary number of URLs in this step. When the customer added all the URLs he is redirected to the page where he can add metatags for each URL. Again for each URL an arbitrary number of metatags can be specified by each customer. All the information is stored in the MySQL database. The following model shows the structure of the CloudSearch user database.
After completing this step the user gets presented the instructions of how to integrate CloudSearch into his own website.

**Integration**

After the user registered and specified all necessary information a new Solr core is created for this specific user. In order to integrate CloudSearch to his own website the user has to consider three steps:

- Add the search box
- Add the result page
- Add the JavaScripts and the CSS-File to the result page

For the first steps a simple HTML-form is provided that points to the result page:

```
<form id="cloudsearch" action="search.html" method="GET">
  <input type="text" name="q" size="31" />
  <input type="submit" value="Search" />
</form>
```

By default the result page is set to search.html. However the user is informed that he has to edit this value in case the result page differs from the default URL.

Next the customer has to add the HTML that will show the search results provided by CloudSearch. This HTML has to be added to page that has that has been specified in the action-field in the first step. Whereas the search box code looks the same for every user, the search result HTML is created dynamically depending on the metatags specified by the user.

![Fig. 6. User database ERM](image-url)
Finally the JavaScripts have to be imported at the search result page. All the JavaScript that are needed for the integration are hosted on a web server on the same machine the application dashboard is running on.

```html
<script src="http://cloudsearch.com/cloudsearch/Solrcore_12/cloudsearch.js"></script>
<script src="http://cloudsearch.com/cloudsearch/core/Core.js"></script>
<script src="http://cloudsearch.com/cloudsearch/core/AbstractManager.js"></script>
...
```

Except for the cloudsearch.js the JavaScripts are the same for each customer. Similar to the search result HTML-Code the cloudsearch.js is created dynamically depending on the users input. Again it takes into consideration the specified metatags in order to create the appropriate queries. Secondly it points to the Solr core that has been created for this particular user. Through this it is guaranteed that the visitors of the customers’ website get presented the results from the right Solr index.

The integration instruction and all the HTML-Code and JavaScript can be accessed at any time when the customer logs in to the CloudSearch application dashboard.

**Control Panel**

All users of CloudSearch have access to their individual control panel. The control panel is an overview that included the customers’ configuration, the crawl status and a health report.

**Configuration**

The configuration lists all the urls and metatags specified by the customer during the registration process. The customer can access the integration code at any time using the control panel.

**Crawl Status**

The crawl status informs the customer about the next crawl. We configured Nutch to crawl the urls once a day. The crawl status is generated by a PHP script that sends a request to the Nutch master node to retrieve the time of the next crawl.

**Health Report**

The health report informs the customer whether the Hadoop cluster and SolrCloud are up and running. The health report is generated by a PHP script that checks periodically if Hadoop and SolrCloud are still running.
6.5 Search Result Page

After the registration, configuration and integration part, the search results of the customer’s website are provided by CloudSearch. In the following image one can see the structure of the search result page.

The search result page is separated into two main parts. On the right side, the search results with the time of indexing, a preview text snippet and a link to the url of the title are presented. The left side consists of the current selection, the search box and the faceted search. The current selection section acts like a filter. If a user clicks on one of the tags of the faceted search section or searches for one or more keywords, the current query will be adapted taking the new keywords or tags into account. All keywords and tags that are used in the current query are listed in the current selection. The user has the option to remove one or more item from the list. The search box is placed below the current selection section. The search can be executed by typing enter or by clicking on one of the suggested keywords that are generated by the auto suggest component. Finally we decided to implement the faceted search feature. Depending on the specified metatags during the registration, the search result page has one or more faceted search sections.

Fig. 7. Search result page
The whole search result page is built using AJAX technology in order to send requests to Solr asynchronously. For the implementation we used “AJAX Solr”, a JavaScript library for creating user interfaces to Apache Solr [8]. The following JavaScript files were included in the search result page:

- cloudsearch.js
- Core.js
- AbstractManager.js
- Manager.jquery.js
- Parameter.js
- ParameterStore.js
- AbstractWidget.js
- ResultWidget.js
- PagerWidget.js
- CurrentSearchWidget.js
- AbstractTextWidget.js
- AutocompleteWidget.js
- AbstractFacetWidget.js
- TagcloudWidget.js

The Manager is responsible for sending request to the Solr core of the specific customer. The cloudsearch.js initialises the Manager and contains the address to the Solr core. The responses are received in JSON-format by the manager and are passed to each widget for handling and presenting them at the corresponding HTML section.

6.6 Integration of Components

![Component integration diagram]

**Fig. 8.** Component integration
The above diagram specifies a typical workflow of our system. On registration, the new user is required to enter a list of urls that he wants to be crawled and indexed. He also specifies metatags, that may be found on the urls. These metatags can be different for the different urls that he has entered. On successful registration a new Solr core with two shards is created for this user. In case we had more virtual machines, it would also be possible to replicate these shards for the purpose of fault tolerance. At the same time, the user is provided with some HTML code and a list of JavaScript files that he is required to add to the pages where he wants the search functionality. The next step is to create a Nutch job that would crawl the websites on the internet taking into account the configurations that have been specified by the user. For this, we first go and configure the Nutch instance. Here we will setup, for example, the metatags that need to be extracted. Once that is done, Ant is used to build Nutch. This results in a job file, which is then stored in a separate directory, along with the seed file for its corresponding url. This process will repeat for all the urls that the user has specified. Now that we have the Nutch jobs and the urls setup, our next task is to run them on Hadoop. For this we have setup a script that goes into each folder gets its seed file, adds it to HDFS, and then runs the Hadoop command with the corresponding job. It will do this for every url in our system. We use a cron job on a daily basis to run this script.

7 Evaluation

For the evaluation of CloudSearch we refer to the requirements we specified in the corresponding chapter. We will evaluate our implementation is respect of scalability, quality of search and ease of integration. Scalability will be evaluated using quantitative measures whereas the quality of search and the ease of integration are evaluated qualitatively.

7.1 Scalability

Environment
For this project, we had a total of 6 virtual machines running on Amazon EC2. All machines were of the same type (micro instance) with the following specifications:

- Single Core
- 615 MB of Memory
- 18 GB Hard Disk

Note that according to the AWS documentation, micro instances usage profile is targeted towards applications that have a spiky usage profile. The following graph is taken from their documentation:
For the evaluation of scalability we have to consider both dimensions, scalability is respect of traffic and scalability in respect of the size of the website. Firstly we will generate search requests synthetically in order to simulate different degrees of traffic.

**Scalability in Respect of Traffic**

In order to generate traffic synthetically we used “ab”, a tool for benchmarking Apache installations [10].

In the first scenario we start with 1000 requests and increase the number of requests in each round by 1000 until we stop at a total number of requests per round of 10000. In this example we set the concurrency level to 1. This means that there are no simultaneous requests. First we look at the percentage of failed requests.

![Percentage of failed requests](image)

**Fig. 10. Percentage of failed request without concurrency**
Without simultaneous requests we increase the number of requests in each round without noticing a significant change in the percentage of failed requests. A second measure that was taken into consideration is the 99th percentile of the response time.

![Response time (99th Percentile)](image)

**Fig. 11.** Response time without concurrency

Again the response time did not change significantly when increasing the number of requests. In the next evaluation scenario we started at 100 requests and increased the number of requests in each round by 200. In this scenario we assumed that 5% of the requests are performed simultaneously. So if the number of requests is 200, the concurrency level is set to 10. Consequently we will generate 20 times 10 simultaneous requests. We decided to make the level of concurrency dependent on the number of total requests as this is the most realistic scenario. Taking concurrency into consideration the results differ dramatically from the ones in the first scenario. The following diagram shows the results for the percentage of failed requests.
The number of failed requests increases disproportionately high when increasing the total number of requests. At a total number of 800 requests and a concurrency level of 40 we can see that 99% of the requests failed. Consequently the SolrCloud components of CloudSearch do not scale for the more realistic scenario including concurrent requests. Next we look at the 99th percentile of the response time.

As we can see, for the subset of requests where a response was received, the 99th percentile of the response time increases proportional to the number of requests. An application that scales should keep the response time constant for an increasing number of requests. So again this shows us that CloudSearch does not scale for an increasing number of simultaneous requests.
Scalability in Respect of Website Size

To scale the fetching and indexing of pages, we had two approaches: either to run Nutch on barebone machines on several (in our case 3) servers or to run it on top of a 3 node Hadoop cluster. Since we have multiple clients, the first approach would mean that one particular client gets all his fetching and indexing done on a single server. To really distribute it, we went ahead with the second approach: we wanted to make that we are even able to do a distributed crawl, so that if a website has a large number of urls to fetch, it doesn’t take much time. Hadoop therefore allowed us to distribute the workload of fetching and indexing over multiple machines for each seed url. One real shortcoming of this was the fact that we had to contend with the added load that Hadoop adds on each machines, including the cost of running multiple JVM’s on a single node. Since we used micro instances for running our Hadoop cluster, this ended up being quite a bottleneck. However, this is not to say that the approach we took was wanting, but that micro instances are not up to the mark of handling such load (specifically their limited memory of around 600 MB).

To evaluate the maximum crawl sizes that we could successfully run, before the server threw enough memory exceptions for it to crash, we ran several crawls on http://www.edition.cnn.com with progressively increasing values for topN and depth. We measure the time it takes for crawls with the configured topN and depth values and then map it onto a graph. We map, for each value of depth, progressively increasing values of topN. Since we are trying to gouge the capacity of the Hadoop cluster, we don’t limit the results based on any domain filtering (so we get most number of URLs per run). Therefore, Nutch will fetch any URL in any domain.

The following graph shows our result.
Each line corresponds to a particular depth value, with increasing topN with jumps of five. As we can see there is a shift in the graph from one depth value to another. This means that the running time for each run of Nutch jumps with an increase in depth. Likewise there is a near consistent increase in running time for crawls with increasing topN (depth being constant). This is more prominent in the lines of depth-25 and depth-30. It can be observed that in the first four lines (depth-5 to depth-20) the graph flattens out at towards the end. For the lines representing depth-5 and depth-10 this trend is visible from the beginning. This can be explained by the fact that total number of URLs being fetched in each instance is similar. This is most likely due to duplicate links occurring in different pages. For low values of topN and depth, this is quite a common occurrence. Since topN return the first N URLs in a page, it is very likely that in a domain like CNN, we get the same set of URLs over and over again (say for example, due to the top menu bar). Hence when the running time stays about the same from topN 35 to topN 40, it means that the number of URLs fetched remains the same between the two runs. This is not observed in the graphs representing depth-25 and depth-30, since we have more chances of diverse URLs being generated. Lastly, the graphs for depth-25 and depth-30 terminate abruptly. This is due to memory issues we have with Hadoop on micro instance. At this point we start getting a lot of “out of memory” exceptions and the process crashes.

In conclusion, running such a setup on the given hardware was not the best decision. Although CPU power was never a problem, memory was significant bottleneck. In
retrospect, to truly gauge the scalability of our system, we would need to start off with servers that meet some minimum requirements.

7.2 Quality of Search

Relevance to Query

In order to evaluate the relevance to the query we assumed that we have given a page of a website that is known to be the best result for a certain search query. Given this page we used the CloudSearch results for that query and analyzed on which rank the best result appears. If the page that is known to be the best result is ranked at the top, the relevance to the query is excellent. If the page cannot be found, the relevance to the query is bad. For the experiment we crawled the websites of CNN and Wall Street Journal. In the following table one can see five queries and the corresponding rank at the CloudSearch result page.

<table>
<thead>
<tr>
<th>Query</th>
<th>android vine</th>
<th>samsung mini</th>
<th>stock scan</th>
<th>turkey protest</th>
<th>Google Waze</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>2</td>
<td>2</td>
<td>not in top 10</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

In total we used the described procedure to determine the rank of the results for 25 queries. The average rank over all queries is 2.04. This means that in average the page with the content searched by the visitor is ranked on the second place in CloudSearch. Consequently the default page rank that is used by Apache Nutch is not sufficient and there is still room for optimisation.

Assisting the User

Besides the relevance of the query, CloudSearch had the requirement to provide some functionality to support the user while searching. In order to meet this requirement CloudSearch implements the following features that were already described in previous chapters:

- Filter
- Auto Suggest
- Faceted Search

7.3 Ease of Integration

To evaluate the ease of integration, we assume customers that have only basic HTML knowledge. So the requirement was that CloudSearch should be capable of being integrated in the customers’ website without any advanced web development skills. Furthermore after the integration the customer should not have any maintenance ef-
In order to evaluate these criteria we look at the steps that have to be performed for the integration of CloudSearch:

1. Copy the HTML-Code for search box to the website
2. Copy the HTML-Code for the search results to the page specified in the action field in 1.
3. Copy the JavaScript and CSS imports to the head-part of search result page.

The only things a customer has to know are:

- The difference between body and head part of a HTML document
- The HTML structure of his own website in order to place the search box and the search result HTML-Code at the right place.

One can see that the integration is possible with only basic HTML knowledge.

After the integration of CloudSearch the customer can login and access his control panel. From the customers’ side no actions have to be taken in order to keep CloudSearch running.

8 Conclusion

Implementing a performant and scalable search engine entails the need for infrastructure and specific knowledge. In order to bypass the challenge of implementing a own search engine, companies can use the services of custom search engine providers. CloudSearch is a custom search engine based on the Apache open-source projects Hadoop, Nutch and Solr. It was designed to be scalable, easy to integrate and to provide high quality search results. The architecture of CloudSearch enables the scaling of both, the crawler and the search service, independently from each other.

The evaluation of CloudSearch reveals that the current implementation lacks significantly in the ability to scale. The search service as well as the crawler are not able to cope with an increase in traffic or the website size. In the context of this project CloudSearch was running on Amazon EC2 micro-instances. Future work should analyze the scalability in a production environment.

CloudSearch is using a distributed search index which provides fault tolerance only to a certain degree. If one server crashes, the other is able to serve the pages that are stored in its file system, i.e. to say that half of the indexes are still available. In order to make CloudSearch scalable as well as highly available future work should consider the introduction of Solr index replication.

9 References