# Search Engine for Wikipedia Articles

Search ranked by Page Rank and TF-IDF

Markus Pettersen*
University of Stavanger, Norway
mark.pettersen@stud.uis.no

Trond Tjersland
University of Stavanger, Norway
t.tjersland@stud.uis.no

## ABSTRACT

In information retrieval, TF-IDF and PageRank are two frequently used algorithms for ranking search results. Both algorithms become increasingly expensive to perform on a single machine as the size of the dataset increases. This paper introduces implementations of these algorithms in the Hadoop and Spark systems, which allows for using a cluster of machines to perform computations on massive datasets. This allows the computation of TF-IDF and PageRank on large datasets to become feasible. PageRank is implemented in Spark, while TF-IDF has both a Spark and Hadoop implementation. Preprocessing is implemented in Hadoop.

## KEYWORDS

PageRank, TF-IDF, MRJob, Hadoop, Spark, Search Engine

## 1 INTRODUCTION

In information retrieval, it is important to order the retrieved information relative to what is being searched for. For example, if we are retrieving documents using a search query, we want to return an ordered list of documents with the most relevant documents on top. To order or "rank" a collection of documents we can use the TF-IDF and PageRank algorithms. These algorithms become computationally expensive when performed on large datasets, which are quite common in the context of information retrieval. To scale such algorithms to work with large datasets required us to distribute both the data and the computation to clusters of computers. Apache Hadoop and Spark are two general-purpose frameworks that handle the distribution of the computations and data storage. By implementing ranking algorithms and their preprocessing step in these frameworks, we effectively scale our solutions to work with large datasets.

We want to implement ranking algorithms and their preprocessing step in Hadoop and Spark using a large dataset. To further specify; we want to implement TF-IDF, PageRank and their preprocessing in Hadoop and Spark using all articles from a dump of the English Wikipedia. We also want to make a search engine to test the validity of our solution. Implementing TF-IDF and PageRank in Hadoop and Spark shows that these algorithms can be scaled to large datasets using computational clusters. Hadoop and Spark use the MapReduce programming model. Programming using MapReduce can be difficult as it represents some unique and substantial challenges due to its distributed and parallel nature. Implementing our solution in these frameworks therefore requires more work than implementing the same solution using standard programming models.

In this paper we make the following key contributions:

- We use MRJob for preprocessing.
- We use a combination of TF-IDF and PageRank to evaluate and rank search results.
- We demonstrate the implementation of TF-IDF algorithm in two different ways using MRJob and Spark implementation.
- We demonstrate the implementation of the PageRank algorithm using Spark.
- We evaluate the calculational performance of PageRank with different clusters and cluster configurations.
- We demonstrate the implementation of a search engine based on the results from TF-IDF and PageRank.

## 2 BACKGROUND AND MOTIVATION

This section contains background information on the different aspects of our project. Here we will give you insight to what a search engine is, what TF-IDF and PageRank is, why we are using these algorithms and how they will work together. And information about the dataset we chose to work with.

### 2.1 Search Engine

Oxford Dictionary defines a search engine as:

> A program that searches for and identifies items in a database that correspond to keywords or characters specified by the user, used especially for finding particular sites on the World Wide Web. [2]

The first search engine WHOIS [10] was created in 1982, eight year before debut of the Web in 1990. This search engine was created to retrieve information from databases. Since then there have been many search engines, some of the most notable of these are; Google, Yahoo and Bing. These search engines were created to retrieve web pages on the Internet. Our implementation will search for documents within a database. We will be using the TF-IDF and PageRank algorithms to search through our dataset and rank the search results.

### 2.2 TF-IDF

TF-IDF is a numerical statistic often used in information retrieval that tries to reflect how important a term is in a document, when that document is a part of a set of documents [9]. TF-IDF is short for *term frequency - inverse document frequency*. A TF-IDF value for a given term in a given document is proportional to its frequency in the document, but this value is offset by the number of times that term appears in the complete set of documents. TF-IDF is a very popular statistic for term-weighting in a document. For example a

*Both authors contributed equally to this research.

Supervised by Thomasz Wiktorski.

*Project in Data Science (DAT500), IDE, UiS*
2019.

paper that went through 200 scientific articles about research-paper recommender systems (i.e information retrieval systems) [6] found that when a term-weighting scheme for a system was specified, 70% of these systems used TF-IDF.

There are different ways to calculate TF-IDF, but a common approach is to use a formula that is analog to what TF-IDF stands for.

```
for term t and document c:

(frequency of t in c) * log(sum of all documents c) / (sum
    of all documents c where t is a part of c))
```

This approach is however biased towards larger documents because larger documents naturally have bigger frequencies of terms. A slightly modified approach is therefore:

```
for term t and document c:

((frequency of t in c) / (length of document c)) * log(sum
    of all documents c) / (sum of all documents c where t
    is a part of c))
```

This takes into account the length of a document when calculating TF-IDF values for terms in that document.

## 2.3 PageRank

PageRank according to Google:

> PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. [5]

This algorithm can be applied to any data with incoming and outgoing links.

An underlying assumption of PageRank is that more important pages will get more links, and that a link from an important page weighs more than one from a less important page. Figure 1 is a good example of this. If you look at the figure you can see that page *B* has seven incoming links and has the highest PageRank score. What is important to note is that while page *C* has the second highest PageRank score, it only has one incoming link, but since that link is from page *B*, and page *B* only has one outgoing link, the contribution of that link weighs more than any other link in the example. What is also worth noting is that page *E* which has six incoming links has a relatively low score compared with page *B* and *C*, but this is because all the incoming links are from pages with a low PageRank score.

There are mainly two ways to calculate the PageRank score, a normalized and a non-normalized version. We decided to go with the normalized version, the equation of which as shown in figure 2. When added up, the scores should equal 1. Using the normalized score then represents the probability of landing on a page by clicking on a random links. The *d* from the PageRank equation is a damping factor. The theory is that there is a "random surfer" starting on a random page and keeps clicking on links until getting bored. And the damping factor is the probability that the surfer will get bored and start from a new random page. We decided to use 0.85 as a our damping factor, the same damping factor used in the original paper on PageRank [1].

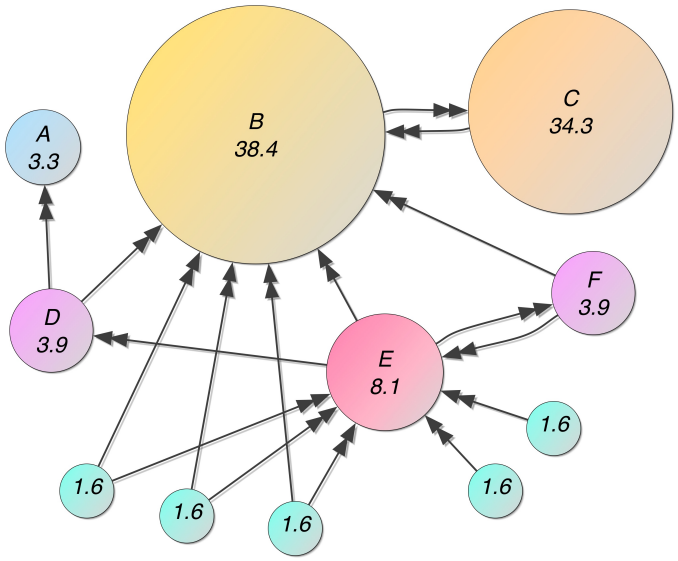


**Figure 1: PageRank example from Wikipedia**

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

**Figure 2: PageRank equation**

## 2.4 Use Case

As mentioned, a search engine is used to extract relevant information from a large database. Our search engine will extract relevant information from a large collection of documents. TF-IDF will be used to find relevant pages to the search query and also give these pages a rank according to the terms in the search query. The PageRank algorithm will then provide additional input to the ranking by evaluating the importance of the pages TF-IDF has deemed relevant.

## 2.5 Wikipedia Dataset

To test our search engine we use a Wikipedia dataset. We chose to use Wikipedia as it is a finite, but complete collection of documents with their own internal wikilinks. We thought that by using a dataset with its own complete system of links, we would would avoid problematic behaviour for PageRank where we might have to follow a link outside the dataset. This turned out not to be true and we had several problems using PageRank on this dataset which you can read more about in section 5, *PageRank Algorithm and Implementation*

The Wikipedia dataset can be found at *https://dumps.wikimedia.org/*, here we can also choose the language of the dataset. For this project we initially decided to work with the English language Wikipedia dataset as this is one of the largest datasets sorted by language.

Choosing such a large dataset would allow us to test how well the implementations of our algorithms scaled. We also decided to have the Norwegian language Wikiepdia dataset as a backup. While the English dataset contains 11,132,071 pages and is 66 GB. The Norwegian dataset only contains 698,181 pages and is 2.5 GB. The file structure of a page from the wiki dump is shown in figure 3.

```
<page>\n
    <title>Det norske Arbeiderparti</title>\n
    <ns>0</ns>\n
    <id>1</id>\n
    <redirect title="Arbeiderpartiet" />\n
    <revision>\n
        <id>15529121</id>\n
        <parentid>9343288</parentid>\n
        <timestamp>2016-01-09T14:07:08Z</timestamp>\n
        <contributor>\n
            <username>StigBot</username>\n
            <id>27430</id>\n
        </contributor>\n
        <minor />\n
        <comment>Standardisering av
             omdirigeringer</comment>\n
        <model>wikitext</model>\n
        <format>text/x-wiki</format>\n
        <text xml:space="preserve">#OMDIRIGERING
             [[Arbeiderpartiet]]\n
            [[Kategori:Omdirigeringer fra eldre
                 skriveform]]</text>\n
        <sha1>ljql4j8oluonmqkzqq17l5d8ylf1uo0</sha1>\n
    </revision>\n
 </page>\n
```

**Figure 3: File structure, first article in Norwegian wiki dump**

## 3 PREPROCESSING

This section show the step-by-step implementation of the preprocessing in MRJob and then in pure Python. Here we go through how to extract the necessary data from each Wikipedia page in such a way that we have a clean dataset to further work with for both the TF-IDF and the PageRank algorithms.

*3.0.1 MWParserFromHell [3].* The Python package *MWParserFromHell* is designed to parse files with MediaWiki [7] code such as Wikipedia articles. This tool allows us to easily strip the article text of coding and formatting, leaving us with clean text to process. It is also able to extract wikilinks and external links from the articles. The extraction of wikilinks is particularly useful to us since these are the links we will be using for our PageRank algorithm.

### 3.1 MRJob

The MRJob implementation of preprocessing is a map-only job as you can see in Figure 4 and Figure 6. The *mapper_init* function is used to hold information over multiple lines. A line is processed by the *handle_line* function shown in Figure 5 This function uses *MWParserFromHell* to extract links from a line and to strip code and formatting from that line. The stripped line is then further processed by removing stopwords using a stopword-list. The links and the processed line is sent back to the main mapper function.

The main mapper function, *mapper*, shown in part in Figure 6 is quite long and has a complex control structure with several if-else statements. It has a complex control structure out of necessity as it has to check and set a number of flags in order to keep track of where the current line is in the context of the article.

The *mapper* function only yields a result when the end of an article is found. When the end of an article is found, the mapper yields the information kept in its class structure and resets all variables for the next article as you can see in line 14-24 in Figure 6. Unfortunately this means that if the end of input is reached before the end of an article, no partial information is yielded. This implementation of preprocessing in MRJob is therefore susceptible to splits in the data caused by mapping and some articles will inevitability be lost. Due to the resource constraints of running MRJob locally without a cluster and the size of our dataset, improving this was not prioritized.

*3.1.1 Experimental setup and results.* We ran the MRJob preprocessing code on a local cluster with 10 GB RAM and a 4-core 4.6 GHZ Intel i7 processor. Running time for the English Wikipedia dataset was 23 hours, 22 minutes. Running time for the Nowegian Wikipedia dataset was 1 hour, 13 minutes.

```
def mapper_init(self):
    self.in_article = False
    self.article_id = ''
    self.Title = ""
    self.in_text = False
    self.text = []
    self.in_links = False
    self.links = []
    self.titles_to_ignore = ['Wikipedia:', 'File:',
         'Template:']
```

**Figure 4: MRJob: mapper_init**

```
def handle_line (self,s):
    s = mwp.parse(s)
    links = s.filter_wikilinks()
    links = self.return_links(links)
    s = s.strip_code()
    s = self.return_tokens_wo_stopwords(s)
    return s,links
```

**Figure 5: MRJob: handle_line**

### 3.2 Pure Python Implementation

Even though our MRJob implementation was able to process the dataset on a local cluster, the results were not as well filtered as we wanted. The output file was also hard to work with if it was not

```
def mapper(self, _, line):
    line = line.strip()
    if self.in_article:
        if self.in_text:
            if line.find('/text') != -1:
                self.in_text = False
                s = line[0:line.find('<')]
                s,l = self.handle_line(s)
            else:
                s,l = self.handle_line(line)
            self.text.extend(s)
            self.links.extend(l)
        elif line.find('</page>') != -1 and self.in_article:
            self.in_article = False
            article_info = []
            article_info.append(self.title)
            article_info.append(self.text)
            article_info.append(self.links)
            self.title = ""
            temp_id = self.article_id
            self.article_id = ""
            self.text = []
            self.links = []
            yield temp_id, article_info
        elif line.find('<text') != -1:
            if line.find('#REDIRECT') != -1:
                    self.in_article = False
                    self.title = ""
                    self.id = ""
            else:
                ...
```

**Figure 6: MRJob: mapper_snippet**

```
def startElement(self, name, attrs):
  """Opening tag of element"""
  if name in ('id', 'title', 'text'):
    self._previous_tag = self._current_tag
    self._current_tag = name
    self._buffer = []
def endElement(self, name):
  """Closing tag of element"""
  if name == self._current_tag:
    if name == 'text':
      if self._redirect():
        self._skip_page = True
        pass
      else:
        self._skip_page = False
        self._process_page()
    elif name == 'id' and self._previous_tag == 'id':
      pass
    else:
      self._values[name] = ' '.join(self._buffer)
  if name == 'page':
    if not self._skip_page:
      self._pages.append((self._values['id'],
                          self._values['title'],
                          self._values['text'],
                          self._values['wikilinks']))
      self._page_count = len(self._pages)
```

**Figure 7: Code for dealing with XML tags**

| ID | Title | idx txt | Nlinks | word1 | ... | wordN | link1 | ... | linkN |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Ex | 4 | 787 | word1 | ... | wordN | link1 | ... | link787 |
| 2 | Test | 4 | 5415 | word1 | ... | wordN | link1 | ... | link5415 |

**Table 1: csv format of cleaned data**

used for another MRJob implementation. We therefore decided to implement our preprocessing in pure python as well, so we could move on to the next steps of the project.

*3.2.1 XML Parser.* A very useful Python package is *xml.sax* [4]. This is a package designed for working with XML documents. To use this to parse the XML all that is needed it to define the content handler.

The content handler defines what to do with the data that is parsed. By implementing the functions seen in Fig 7 we decide which tags from the XML to process, and the rest we ignore. When a new tag is encountered the buffer as seen in line 6 is set to empty. Until the endtag is encountered the content is saved to the buffer. When encountering the endtag the *endElement* function decides what to do with what is stored in the buffer. And if the endtag is **</page>** indicating the end of the Wikipedia article then it saves the articles; id, title, text and wikilinks before moving on to the next article.

Once the parser is done with the file it saves the cleaned articles to a csv-file with the format as shown in table 1.

*3.2.2 Multiprocessing.* To optimize the preprocessing we implemented multiprocessing. The python preprocessor required the entire data to be loaded into memory and this was a problem since we did not have a computer with enough memory to process a 66 GB file. Luckily Wikipedia provides the dumps in portions, allowing us to work with 56 smaller files instead of a single 66 GB file. Because each file contained only complete articles, there was a need to run them sequentially. That meant that we could run multiple files at a time without having to worry about inconsistent result or the split documents we would get by running it in MRJob.

The code for realizing the multiprocessing can be seen in figure 8. When using multiprocessing there are two main concerns to take into consideration; how much of the CPU each process uses at full capacity, and how much memory each process uses. At full capacity each process used about 12.5% of our CPU, if nothing else is running on the computer then using eight processes would be the optimal use of the CPU, but given that most computer relies on systems running in the background, we found that using seven processes was more efficient since it allowed each process to run at full capacity. On average each process used 3-5 GB of memory, which if we used seven processes would lead to a memory use of 21-35 GB, and since our computer only had 32 GB of RAM this caused our computer to crash. We then tried with six processes,

```
from multiprocessing import Pool
def main():
    # Create a pool of workers to execute processes
    # IMPORTANT!!!
    #   It is vital to choose the number of processes carefully. Each process
    #   can use in excess of 5GB RAM. Use these estimates if you are unsure:
    #      4GB available: DO NOT RUN
    #      8GB available: 1 process
    #      16GB available: 2 processes
    #      32GB available: 5 processes
    pool = Pool(processes = 5)
    # Map (service, task), applies function to each partition
    pool.map(preprocesser, wiki_partitions)
    pool.close()
    pool.join()
```

**Figure 8: Code for multiprocessing**

but then discovered that while the general memory consumption was 3-5 GB it could sometimes use up to 6 GB. Because of this we decided to use five processes when running the preprocessor.

Running the English wiki dump (56 files totalling 66 GB) through the pure Python preprocessor took 19 hours, 37 minutes and 59 seconds. And running the Norwegian wiki dump (1 file of 2.5 GB) took 3 hours, 36 minutes and 24 seconds. When running multiple files through the preprocessor with multiprocessing it processed 3.4 GB per hour versus 0.7 GB per hour when only running one file. This is approximately five times faster which is what we expected given that the multiprocessor ran five processes in parallel.

# 4 TF-IDF ALGORITHM AND IMPLEMENTATION

This section show the step-by-step implementation of TF-IDF in MRJob and then in Spark. For our implementation of TF-IDF, we need four parameters to perform the calculations necessary to find a TF-IDF value. For a term t and document c, these parameters are:

**n** = frequency of t in c
**N** = length of c
**d** = sum of all documents c where t is a part of c
**D** = sum of all documents c

## 4.1 MRJob

The MRJob implementation uses *JSONProtocol* as the input protocol. This means that the implementation is dependent on input from another MRJob. In practice, the code will only work on the output of our preprocessing also done in MRJob.

The code has two MRSteps. The first step has a mapper with the primary objective of counting terms in an article. The functions of this mapper is shown in Figure 9. The counting of terms is done in the yield step in line 11. As a secondary objective, *mapper_get_n* also finds **N** for each article, shown in line 8-9. *mapper_get_n* yields *((article_id, term, N), 1)* for each term in an article, where the first tuple is the key and the 1 is the value. We also implement a partial count of **D** in the mapper that we can later aggregate in the reducer shown in Figure 10. This partial count is done by implementing an _init function, *mapper_get_n_init* and a _final function, *mapper_get_n_final* which keeps track of the number of articles processed and yields this with a special key, *'article_count'* that we can aggregate on, as shown in line 14 in Figure 9.

In the reducer, shown in Figure 10 we aggregate on our tuple key and sum up the count to get **n**. We yield *((article_id, term, N, D), n)* for every term in every article. Again the key is a complex tuple. We cannot yield anything in *reducer_get_n* until we are sure that the key 'article_count' has been processed and **D** has been found. To get around this problem, key-value pairs are stored in the reducers class structure using the *reducer_get_n_init* function. These key-value pairs are only yielded in the *reducer_get_n_final* function with **D** appended.

The second MRstep has a simple mapper that maps *((article_id, term, N, D), n)* to *(term, (article_id, n, N, D))*. Here term is the key. We then reduce on each term in the final reducer. By counting the number of value tuples, i.e *(article_id, n, N, D)* for each term we are able to find **d**. Since we now have the four parameters necessary, the final reducer can yield *(term, (article_id, TF-IDF value))*

*4.1.1 Experimental setup and results.* We ran the MRJob TF-IDF code on the same local cluster we used for our MRJob preprocessing code. This local cluster has 10 GB RAM and a 4-core 4.6 GHZ Intel i7 processor.

The code did not complete on the Norwegian Wikipedia dataset due to running out of memory. We did not attempt the English Wikipedia dataset.

*4.1.2 Analysis.* The implementation of TF-IDF in MRJob was a challenge. TF-IDF involves finding parameters from a set of documents and using these to calculate a TF-IDF value. These parameters once found in MRJob, must be passed on from mapper to reducer and to the next step and so on because they cannot be held in memory for longer than a single map or reduce step. Our solution to this was to embed these parameters into keys, using complex tuples as the keys we pass on. This worked because the part of these complex tuples that were really keys and not just embedded values, had the same parameters and so the parameter embedding did not change the keys in a meaningful way. For example; in our first reducer we reduced by the key *(article_id, term, N)*. Here *(article_id, term)* is our real key, but **N** is fortunately the same for all such keys and embedding **N** in such a way is therefore acceptable.

Another challenge was finding some of the parameters. While a simple parameter like **D** is easy enough to find if you have access to entire dataset in one location, it is not so easy when you are using MapReduce, where the dataset is distributed amongst different nodes. In our implementation we had to partially calculate D for each distributed part of the dataset and later aggregate them in a reducer. Compare this to an approach in python where one could simply call *len(dataset)*.

As is evident from our results on our local cluster, this implementation of TF-IDF requires a lot of Memory. This is because a TF-IDF matrix, even though sparse, quickly becomes larger in size than the

input data. We will see the same problem in the Spark implementation and in an effort to not repeat ourselves, further discussion on this matter will continue in the Spark section of TF-IDF.

```
def mapper_get_n_init (self):
    self.article_count = 0

# article_id, article_info -> (article_id,term,N), count
def mapper_get_n(self, article_id, article_info):
    self.article_count += 1
    article_info = list(article_info)
    terms = list(article_info[1])
    N = len(terms)
    for term in terms:
        yield (article_id,term.lower(),N), 1

def mapper_get_n_final (self):
    yield 'article_count', self.article_count
```

**Figure 9: First mapper in TF-IDF MRJob implementation**

```
def reducer_get_n_init (self):
    self.keys = []
    self.value = []
    self.article_count = 0

# (article_id,term,N), count -> (article_id,term,N,D),
    sum(count) = n
def reducer_get_n (self, term_info, count):
    if term_info == 'article_count':
        self.article_count += sum(count)
    else:
        self.keys.append((term_info[0],term_info[1],
            term_info[2]))
        self.value.append(sum(count))

def reducer_get_n_final (self):
    for i in range (len(self.value)):
        yield (self.keys[i][0],self.keys[i][1],\
        self.keys[i][2],self.article_count), self.value[i]
```

**Figure 10: First reducer in TF-IDF MRJob implementation**

## 4.2 Spark

The spark implementation starts with importing preprocessed data from files into an RDD, *id_and_terms*, with a key-value pair consisting of *(article_id, (term_1, ..., term_n))*. This RDD is then processed into *TF*, with the key-value pair*(term, (article_id, TF-score))* by the code shown in Figure 11. The code starts with mapping *it_and_terms* into another RDD called *id_and_N (line 2-3)*, where N is **N**. TF is then created out of a RDD pipeline, *line 4-11*. First every occurrence of a unique *(id, term)* tuple is counted and these counts are summed up in a *reduceByKey* to find **n**, *(line 5-6)*. It is very important to use flatMap in this step as we map *((id), (terms))* into *((id,term), 1)*. There are more keys in the output than in the input so this mapping is not one-to-one. The next step of the RDD pipeline is a simple join operation where *((id,term), count)* is joined by *(id, N)* on id, *(line 7-9)*. This requires some extra mapping, i.e making id the single key of both RDDs in order to make the join work. A join can only be performed on two RDDs if they have the the same key, we cannot join on anything in the value tuple or specific values in the key. The last mapper simply calculates TF from **n** and **N**, *(line 9-11)*.

```
# Create TF
id_and_N = id_and_terms \
    .map(lambda key_value: (key_value[0],
        len(key_value[1])))
TF = id_and_terms \
    .flatMap(lambda key_value :
        get_IdTerm_tuple(key_value[0], key_value[1])) \
    .reduceByKey(add) \
    .map(lambda key_value: (key_value[0][0],
        (key_value[0][1],key_value[1]))) \
    .join(id_and_N, npart) \
    .map(lambda key_value:
        (key_value[1][0][0],
         (key_value[0], key_value[1][0][1] /
             key_value[1][1])))
# Create IDF
D = id_and_terms.count()
IDF = TF \
    .map(lambda key_value: (key_value[0], 1)) \
    .reduceByKey(add) \
    .map(lambda key_value: (key_value[0], log(D /
        key_value[1])))
```

**Figure 11: TF calculation in spark**

The IDF part is easier to understand and requires less steps to perform. First we find **D** by simply counting the number of articles in the RDD *id_and_terms*. **d** is then found by counting every occurrence of a unique term in TF and summing the counts in a reduceByKey, *(line 15-16)*. The last mapper simply calculates IDF from **d** and **D**, *(line 17)*.

A simple join and map operation calculates TF-IDF from TF and IDF RDDs. The final output is a key value pair; *((term,id), TF-IDF value)*

*4.2.1 Experimental setup and results.* When trying to calculate the TF-IDF of the English wiki dumps we had a cluster with 6 worker node each with 2 CPU cores and 7 GB of RAM. Even with this setup we were not able to calculate the TF-IDF on the full English wiki dumps. We only had that cluster for a short period of time, before the cluster was downgraded to 2 worker nodes with same configuration per node. Because of this calculating the TF-IDF for the English dumps were no longer feasible. We then moved on to work with the Norwegian wiki dumps and with this dump we were able to find the TF-IDF.

*4.2.2 Analysis.* The implementation was made significantly easier by using Pyspark instead of MRJob. Finding **D** for example, which required multiple lines of code in MRJob, was reduced to

a one-liner in PySpark. There were however some challenges involved in joining two RDDs. Because join operation between RDDS requires keys to match exactly and has no join by parameter, some unecessary map operations had to be done in order to make our join operations work.

As you can see in our results, we again ran into the problem of our TF-IDF algorithm requiring too much memory. A subsection in Section 9, *Further Work* is dedicated to potential optimizations to our TF-IDF algorithm.

## 5 PAGERANK ALGORITHM AND IMPLEMENTATION

In this section we show the step-by-step implementation of PageRank, the problems we faced and how they were solved. The flow of the PageRank algorithm is shown in figure 12.

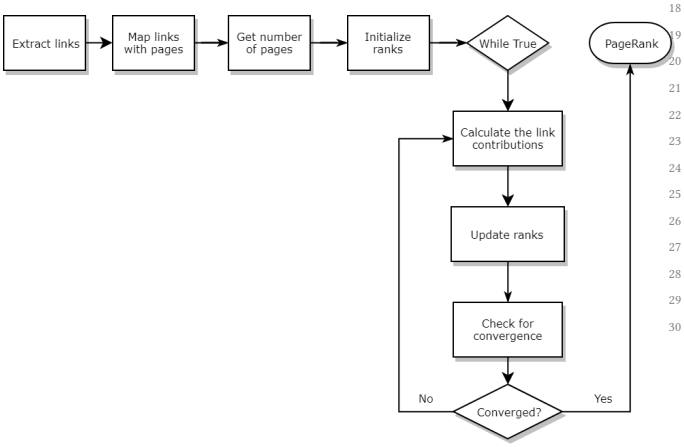


**Figure 12: PageRank Flow**

The PySpark code for implementing PageRank is shown in figure 13. As we can see from the PageRank flow, the first step in the PageRank algorithm was to take the output from the preprocessor and map the links to the page, the code for which can be found in line 11-13, hereby creating what we called *page pairs* (pairs of page titles and the links on the page). The function *getPages* in line 11 simply checks if the path is valid and returns the data from the file or a set of files as an RDD. The next step was to initialize the ranks, line 15, this sets the initial rank to 1 divided by number of documents in dataset. We then start the calculations for finding the PageRank, this runs until a convergence has been reached. The code for the iterations is the while-loop in line 16-28. If a convergence has been reached, the iterations stop and the final PageRank is calculated in line 29-30 by joining the pages from the *page pairs* with the *ranks*.

```
1  def getLinks(page):
2    num = int(page.strip().split('\t')[3])
3    if num == 0: return []
4    else: return page.strip().split('\t')[-num:]
5  def computeContribs(links, rank):
6    if len(links) > 0:
7      for link in links:
8        yield (link, rank / len(links))
9  if __name__ == '__main__':
10   eps = 1e-2
11   pages = getPages(nowiki_path)
12   page_pairs = pages.map(lambda line:
13     (line.strip().split('\t')[1], getLinks(line)))
14   N = page_pairs.count()
15   ranks = page_pairs.map(lambda pair: (pair[0], 1.0/N))
16   while True:
17     contribs = page_pairs.join(ranks).flatMap(
18       lambda page_links_rank:
19         computeContribs(page_links_rank[1][0],
20                         page_links_rank[1][1]))
21     ranks_previous = ranks
22     ranks = (ranks.leftOuterJoin(contribs.reduceByKey(add))
23       .mapValues(lambda rank:
24                  rank[1] * .85 + .15/N if rank[1]
25                  else (rank[0]/N) * .85 + .15/N))
26     convergence = (ranks_previous.join(ranks)
27         .map(lambda rank: abs(rank[1][0]-rank[1][1])).sum())
28     if convergence < eps: break
29   page_ranks = page_pairs.join(ranks).map(
30     lambda page_rank: (page_rank[0], page_rank[1][1]))
```

**Figure 13: PageRank PySpark code**

### 5.1 Analysis

To test whether or not our code was functioning, we used a small subset from the English wiki dumps. When running the algorithm on this subset we found some interesting results. We initially expected that the total PageRank score would add up to 1 as it theoretically should, but this was not the result that we encountered. On our first successful run of the algorithm the total PageRank score we got was less than 0.1. On seeing this result we first thought that we had made an error in our code, and decided to make a test file with perfect conditions. By perfect conditions we mean that all the outgoing links in the dataset links to pages within the dataset and that all pages within the dataset have incoming links. The dataset we made is shown in the first two coulumns of table 2

| PAGE | LINKS | PageRank |
|---|---|---|
| A | B C D | 0.30233 |
| B | A C | 0.16400 |
| C | A E | 0.23371 |
| D | A B C | 0.17063 |
| E | A D | 0.12933 |
| TOTAL RANK | | 1.00000 |

**Table 2: Perfect Test File**

When running this file through the PageRank algorithm we got the PageRank scores that can be seen in the third column of table 2. This time, when the conditions were perfect we got the result we expected and the PageRank scores summed up to 1.

Now that we had shown that our implementation of the algorithm worked as expected under perfect conditions, we now wanted

to test how the algorithm reacts to different scenarios that we may encounter in the real world. We consider three different scenarios:

- *dead pages*
  - Pages with no incoming links
- *dead links*
  - Links that link to pages outside the dataset
- *sink pages*
  - Pages without outgoing links

To test these scenarios we again created some simple test files so that analyzing the results would be easier. Table 3 shows the files and results for testing the three scenarios. As we can see from these results each scenario have an effect on the outcome.

| *dead pages* | | | *dead links* | | |
|---|---|---|---|---|---|
| PAGE | LINKS | PageRank | PAGE | LINKS | PageRank |
| A | B C D | 0.39515 | A | B C D | 0.11792 |
| B | A C | 0.18654 | B | A C H J | 0.08772 |
| C | A | 0.26581 | C | A E K L | 0.10636 |
| D | A B C | 0.15732 | D | A B C | 0.08577 |
| E | A D | 0.03614 | E | A D | 0.05261 |
| TOTAL RANK | | 1.04096 | TOTAL RANK | | 0.45038 |

| *sink pages* | | |
|---|---|---|
| PAGE | LINKS | PageRank |
| A | B C D | 0.07067 |
| B | A C | 0.14352 |
| C | A E | 0.12925 |
| D | A B C | 0.08494 |
| E | | 0.09070 |
| TOTAL RANK | | 0.51908 |

**Table 3: Test of scenarios**

To conclude, the case of the perfect dataset seem to be the case in most examples, online and in textbooks, but is rarely the case when working with real datasets, especially when working with a subset. What we found working with a subset of the Wikipedia articles was that there were a lot of *dead pages* and *dead links*. So dealing with these problems became an important task for us. Next we will go through how we dealt with these problem.

## 5.2 Dealing with *dead pages*

The first scenario, the *dead pages*, was the first problem we addressed. We saw two ways of dealing with this, the first to simply remove dead pages and the second to set the rank of these pages to zero. The problem we discovered with the first solution is that it will also remove the contribution the outgoing links makes to the rank of other pages. We concluded that this was not desirable and moved on to the second solution.

When we tested the second solution. We found that if we initialized the rank of the dead pages to zero the sum of all the ranks would keep increasing with each iteration. The reason for this was that initializing the ranks to zero, was the same as not counting the contribution of that page, but still keeping it among the dataset. A way to solve this was to set the rank of the dead pages to zero at the end after a convergence had been reached. The results of implementing this can be seen in table 4, and show that the sum of the scores is not quite 1, but a lot closer than before.

| PAGE | LINKS | PageRank |
|---|---|---|
| A | B C D | 0.39515 |
| B | A C | 0.18654 |
| C | A | 0.26581 |
| D | A B C | 0.15732 |
| E | A D | 0 |
| TOTAL RANK | | 1.00482 |

**Table 4: *dead pages* fix**

## 5.3 Dealing with *dead links*

The second scenario was that of the *dead links*. One solution would be to remove all links linking outside the dataset. But doing this would require us to check every link of every page and see if the the link linked to a page in the dataset, and if not remove it. This would be an expensive process, and may not scale as well with larger datasets. A side effect of doing this is that it for some pages it may yield a higher PageRank score than it should. The total amount of links on a page along with the rank of the page decides the contribution the links make to their respective pages. Because of this removing the links would lower the link count on pages with dead links and this would give the remaining links a higher contribution than they actually should receive. Resulting in, as previously stated, that some pages would get a higher PageRank score than is realistic.

But given that our objective is to use PageRank to help rank search results, and that the searches will only be done on the data within our dataset, links linking to pages outside our dataset are of no interest to us. We will therefore be removing all *dead links* from our dataset.

The result of this implementation can is shown in table 5. And as you can see the total PageRank score now sums up to 1, proving the validity of our implementation.

| PAGE | LINKS input | LINKS output | PageRank |
|---|---|---|---|
| A | B C D | B C D | 0.30233 |
| B | A C H J | A C | 0.16400 |
| C | A E K L | A E | 0.23371 |
| D | A B C | A B C | 0.17063 |
| E | A D | A D | 0.12933 |
| TOTAL RANK | | | 1.00000 |

**Table 5: *dead links* fix**

## 5.4 Dealing with *sink pages*

The third and final scenario that caused us problems were *sink pages*. One way of solving this would be to assume that a *sink page* links

to all other pages [8]. Another way would be to simply remove all *sink pages* from the dataset.

We decided to go with the latter and remove all the *sink pages*. The reason for this being that this would only be one extra step in the initializing part of the PageRank algorithm, before starting the PageRank calculations.

Another downside to removing *dead links* is that if a page only has *dead links* then by removing them we create more *sink pages*.

When implementing this there are two aspects that are important to note. First, if the *dead links* have been removed, this may result in more *sink pages* if a page only had *dead links*. Because of this removing the *sink pages* has to be done after the *dead links* have been removed. Secondly, removing *sink pages* will result in the links that link to these pages becoming *dead links*. So, to properly remove both *dead links* and *sink pages* completely, the removal process has to be done in three steps.

(1) Remove all initial *dead links*
(2) Remove all *sink pages*
(3) Remove all *dead links* resulting from the removal of *sink pages*

And now the total PageRank score adds up to 1, as it should. The result of this implementation can be seen in table 7.

| PAGE in | LINKS in | PAGE out | LINKS out | PageRank |
|---|---|---|---|---|
| A | B C D | A | B C D | 0.39066 |
| B | A C | B | A C | 0.19017 |
| C | A E | C | A | 0.27099 |
| D | A B C | D | A B C | 0.14818 |
| E | A D | | | |
| TOTAL RANK | | | | 1.00000 |

**Table 6: *sink pages* fix**

## 5.5 Optimization

Once the PageRank algorithm was implemented properly and was able to handle each of the three scenarios mentioned in the previous sections, we moved on to test the algorithm on a larger subset of the data. Our initial subset consisted of only one 500 MB file. Now to see if our implementation would scale properly as the data-size grew, we tried with half the data (15 GB of files).

When running this with the initial cluster we had been assigned we encountered the error *no space left on device*. Since we went from processing a 500 MB file to 28 files totalling 15 GB and our cluster consisted of two worker nodes each with 2 cores and 1.2 GB of RAM, we simply thought that the problem was lack of resources. We were then provided with a new cluster consisting of three worker nodes each with 2 cores and 7 GB of RAM. When this too failed we decided to run some more test to see if there could be something else at fault. We tested with the *perfect* test file we had used to validate our algorithm. We lowered the convergence threshold to ensure that it would run for more iterations. When this, an 80 B file, also failed we came to the conclusion that the *no space left on device* error we got was not necessarily related to the size of the data.

When analyzing the information on the Spark cluster in real-time we discovered something we thought was a little weird. For every iteration of the PageRank calculations the number of tasks per job and the execution time doubled. Since the PageRank calculations did the same thing in every iteration and the size of the data created and stored in every iteration should stay the same, it didn't make sense that the execution time and amount of jobs would double every time. This did however explain why we were having *no space left on device* errors. The problem was not that there were not enough resources to perform the calculations, but that the number of partitions doubled for every iteration, meaning that even though it was more space left on the device there had been created so many partition (reserved space) that there was not enough available space to create new partitions.

We found that the root cause of this problem was that when performing join operations in an iterative function the number of partitions would increase to infinity [11], and that when working with low level Spark it is necessary to manually control the partitioning. The join function in spark takes in two parameters, the RDD to be joined and the number of partitions. Unless the latter is defined the number of new partitions used will increase with every iteration. When defining this parameter it ensures the number of new partitions increases linearly, which significantly lowers the chance of getting an *no space left on device* error.

Having gained control over the partitioning of the join operations we were able to run the subset of our data through the PageRank algorithm. We also found a function, see figure 14, that helps with the general partitioning of the RDDs and if *cache* is set to *True* then the RDD is cached in memory making the join operations a lot faster.

```
1 def procRDD(rdd, cache=True, part=True, hashp=True,
      npart=12):
2     rdd = rdd if not part else rdd.repartition(npart)
3     rdd = rdd if not hashp else rdd.partitionBy(npart)
4     return rdd if not cache else rdd.cache()
```

**Figure 14: Function from stackoverflow [11]**

The implementation of the *procRDD* function created another problem for us, which was that the memory reserved for cached data was quite limited. This meant that if the RDDs that were cached stayed stored in memory, the cluster would eventually crash when trying to cache a new RDD if there wasn't enough memory left of the reserved space. We dealt with this by setting the Spark parameter *cleaner.ttl* so that RDDs and other variables would only be stored in memory for a limited time. By default this parameter is set to infinite which means that nothing is ever removed, we found that by setting this parameter to 1.5 times the execution time of one iteration allowed the variables to be stored for only as long as they were needed. This means that the value of this parameter is relative to the size of the data, and the cluster that Spark is running on. It's a parameter that needs to be tuned to fit its environment.

# 6 SEARCH ENGINE IMPLEMENTATION

The goal of this project was to create a search engine for finding Wikipedia articles relevant to a search query. In this section we will go through how we used TF-IDF and PageRank to implement a search engine. We implemented two versions of the search engine, the first in Spark using SparkSQL and the second in pure Python using the output from TF-IDF and PageRank.

## 6.1 Spark implementation

We decided to implement a search engine in Spark using SparkSQL because it we wanted to utilize the computational power of the cluster and because SparkSQL allows us to use a SQL query to retrieve data from our dataset. Figure 15 shows the code for the search engine we implemented in Spark. The function *search* takes in a search query represented as a string, and then finds documents that it deems relevant to the query. The function returns the top ten result with names of the documents and their respective search scores. If the title of a document matches the search query, then this document will get a score of infinite ensuring it ends at the top of the list, this is shown in line 8.

When a user calls the *search* function with a search query, it first checks if there is a title that matches. Then for each word in the query it checks if the TF-IDF contains the words and if so returns the id of every document containing the word and the corresponding TF-IDF score. If a document contains several words from the search query, then TF-IDF score of each word is added together and mapped with the document id. It then for every document id found in the TF-IDF process finds the corresponding PageRank score, and multiplies the TF-IDF score with the PageRank score, creating the *search score*. Finally it sorts the all the results by the search score in descending order and returns the top ten results.

```
1  def search(query):
2    start = time()
3    search_res = []
4    title_search = (sqlContext
5      .sql("SELECT title, rank FROM PR WHERE
           title='{}'".format(query))
6      .collect())
7    if len(title_search) > 0:
8      search_res.append((title_search[0]['title'], np.inf))
9    for word in query.split():
10     results = (sqlContext
11       .sql("SELECT list from TF_IDF WHERE
           id='{}'".format(word))
12       .collect())
13     docs = {}
14     for result in results:
15       for row in result:
16         for item in row:
17           if item[0] not in docs:
18             docs[item[0]] = dict()
19             docs[item[0]]['TF_IDF'] = item[1]
20             res = (sqlContext
21               .sql('SELECT title, rank FROM PR WHERE
                   id={}'.format(item[0]))
22               .collect())
23             if len(res) > 0:
24               docs[item[0]]['title'] = res[0]['title']
25               docs[item[0]]['rank'] = res[0]['rank']
26             else:
27               docs[item[0]]['title'] = None
28               docs[item[0]]['rank'] = 0
29           else:
30             docs[item[0]]['TF_IDF'] += item[1]
31     for _, doc in docs.items():
32         search_res.append((doc['title'],
               doc['TF_IDF']*doc['rank']))
33   search_res.sort(key=lambda tup: tup[1], reverse=True)
34   end = time()
35   print('Found', len(docs), 'matches in', round(end-start),
         'seconds')
36   return search_res[:10]
```

**Figure 15: Search Engine using SparkSQL**

## 6.2 Python implementation

We were not happy with the performance of the SparkSQL implementation so we wanted to see if we could make a faster version in pure Python. This version uses the Python object dictionaries as the core of the search engine. The reason for this is that Python dictionaries are key-value mappings, making them easy to search through by key. The data we used to do this was the output from the TF-IDF and PageRank algorithms implemented in Spark, and the initial output of the preprocessor from the pure Python implementation.

The output of the PageRank algorithm consists of key-value pairs where the key is the title of the page and the value is the PageRank score. This format is analog to the Python dictionary format we wanted, and we simply read the file as is into a dictionary. With the output from the TF-IDF algorithm we created a dictionary with the key as a word, and the value as a list of tuples with page-ID and TF-IDF score. This way we could easily access all pages that contains a given word. With the output from the preprocessor we created a dictionary with the page-ID as the key and page title as the value.

Having these three dictionaries we could easily search for a word, find all the pages that contained the word with the corresponding PageRank and TF-IDF score. The program structure of this implementation was more or less the same as the SparkSQL implementation. The code for the *search* function from this implementation is shown in figure 16.

## 6.3 Results and Comparison

The essential aspect of our search engine was the ability to find pages relevant to the search query, and not just the the page with the corresponding title. To see if we were able to achieved this we tested with a few queries which gave us the results you see in tables [11, 12, 13, 14] in section 11. Most of the results we get here seem quite relevant to the query while there also seems to be some outliers. For example when searching for 'potet' (potato) and the 5th result is 'Fylkesvei 918 (Nordland)' (a road). If you look at the results from searching for 'USA', you will see 'USA' as an entry twice. This happens because the first time, when the search rank is

```
def search(query):
    start = time()
    search_res = []
    if query in PR:
        search_res.append((query, np.inf))
    for word in query.split():
        docs = dict()
        if word in TFIDF:
            for doc, score in TFIDF[word]:
                if doc not in docs:
                    docs[doc] = dict()
                    docs[doc]['TF-IDF'] = score
                    page = pages[doc]
                    if page in PR:
                        docs[doc]['title'] = page
                        docs[doc]['rank'] = PR[page]
                    else:
                        docs[doc]['title'] = None
                        docs[doc]['rank'] = 0
                else:
                    docs[doc]['TF-IDF'] += score
        for _, doc in docs.items():
            search_res.append((doc['title'],
                 doc['TF-IDF']*doc['rank']))
    search_res.sort(key=lambda tup: tup[1], reverse=True)
    end = time()
    print('Found', len(docs), 'matches in', end-start,
          'seconds')
    return search_res[:10]
```

**Figure 16: Search Engine using SparkSQL**

infinite, the match was with the title. The second time it appears is because of the information on the page, and the combined score of the PageRank and TF-IDF.

As we mentioned, the reason for creating a pure Python implementation of the search engine was because we were not satisfied the performance of the SparkSQL search engine. Both of the implementations returned the same results, with the same search score. So to test which implementation was the better of the two, we tested by using the same search queries for both and comparing the search time.

The SparkSQL search engine was tested on a cluster with two worker nodes each with 2 CPU cores and 7 GB of RAM. The pure Python search engine was tested locally on a computer with an Intel i7-6700K, quad-core CPU and 32 GB of RAM.

| | | **SparkSQL** | **vanilla Python** |
|---|---|---|---|
| Query | num result | search time | search time |
| hund | 1,294 | 326s | 0.003947s |
| Akita | 53 | 8s | 0.000000s |
| Vesle hund | 1,294 | 351s | 0.002988s |

**Table 7: Comparison of SparkSQL and pure Python**

From these results it is very clear that the pure Python implementation is vastly superior to SparkSQL in this case. When trying to query a sentence the SparkSQL version used several hours without being able to finish the query, while for the pure Python version searching for *'La oss prÃÿve med en lang setning som inneholder ord som, Kina hund Norge Sverige USA'* resulted in 60170 matches and was processed in 2.19 seconds.

So to conclude, the search engine we implemented in pure Python was by far the best version, and the combination of TF-IDF and PageRank yielded relevant results.

## 7 ANALYSIS AND PERFORMANCE TUNING

In this section we will analyse the performance of the PageRank implementation on the Spark cluster. We will systematically go through the different cluster configurations and the code optimizations we did to make the algorithm more efficient. Because of the time limitations when working with the more powerful clusters, we did most of the performance analysis on a subset of the data, approximately half of the English wiki dump, about 15 GB.

### 7.1 Optimization of PageRank implementation

As mentioned in the section concerning the PageRank optimization we had some problems with *no space left on device* errors. Here we will go more into detail of why this happened and visually show the improvements made.

With our initial implemented as can be seen in figure 13 the amount of partitions doubled with each iteration, so did the execution time, even though the number of jobs stayed the same. The reason for this is that when performing a join operation the data is partitioned, and unless the number of partitions is defined, then it will choose what it thinks it needs. This is not a problem that reveals itself if the join operations are done once and only once, however if the are used iteratively like in a loop as done in the PageRank calculations then for every iteration the partitioned data is partitioned again. This causes the number of partitions to double with each iteration.

A simple way to solve this is to define the number of partitions every time a join operation is called. When we did this the number of new partitions created were the same for each iteration. Another optimization we found as we mentioned was to use the function *procRDD*. This function handles the partitioning and caching of RDDs for us.

To demonstrate the difference in performance between these three we ran the algorithm multiple times with the *perfect* test file. We used this file instead of wiki files to be able to run the algorithm multiple times in not to long a time, and to ensure that the algorithm would actually be able run all the way through. The figure 17 shows the total number of partitions at each iteration, this clearly shows why the using partitioning managing and caching is useful. The figure 18 shows the execution time of the iteration and the figure 19 show the total execution so far at each iteration. As can be seen from these figures the best one with respect to number of partitions is partitioning without caching, while the best in execution time was the one with both partitioning and caching. And since they all return the same result we want to be able to use the fastest one.

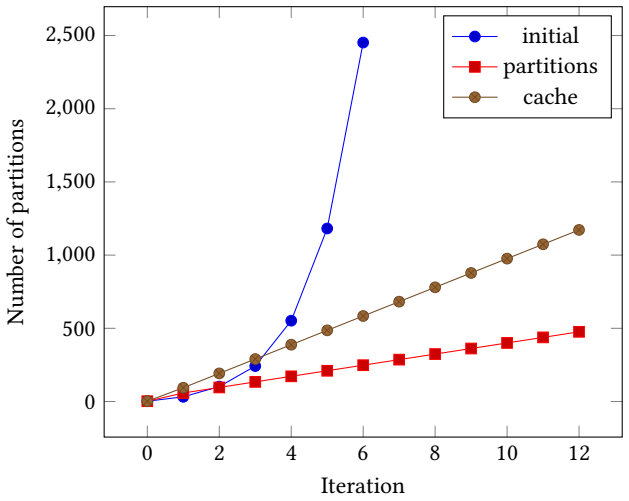


**Figure 17: The graph shows the total number of partitions created so far.**

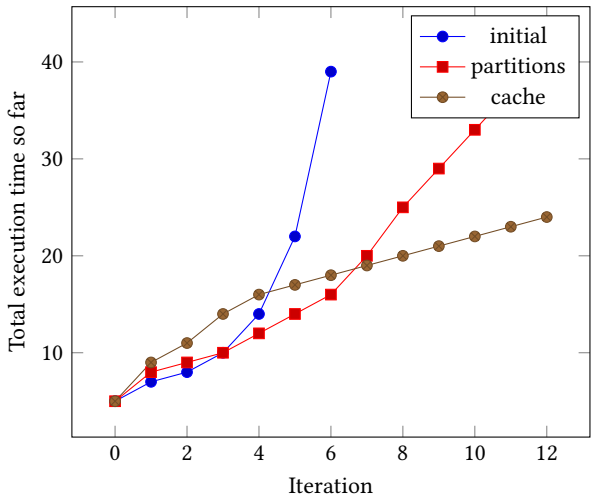


**Figure 19: The graph show the total execution time (in seconds) so far.**

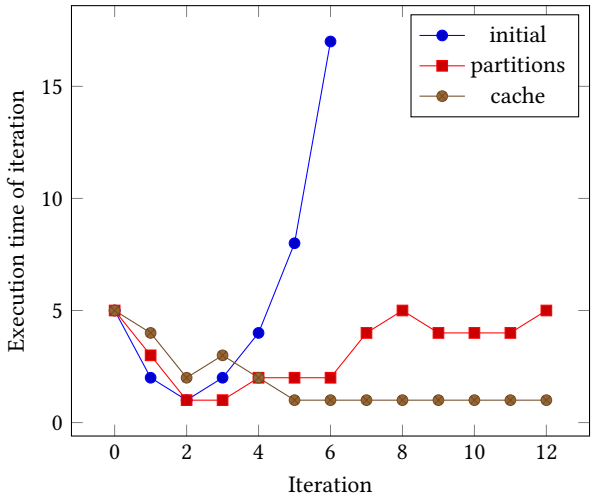


**Figure 18: The graph shows the execution time (in seconds) per iteration.**

## 7.2 Adjustments of objects time to live

As you see in figure 17 even though the number of partitions when using cache were a lot lower than than the initial implementation, the total number of partitions were still increasing. We could run the PageRank algorithm for longer, and would be able to process more of the full dataset, but we would still get *no space left on device* errors. The reason being that by default a partition created will never be deleted, even if it is not needed anymore. The solution for this was to set the the Spark system parameter *cleaner.ttl*, which defines the time to live of any object on the Spark cluster. By setting this parameter to a value that equals at least the execution time of one iteration we ensure that the variables stay in memory for only as long as they are needed. To better demonstrate the improvement of this optimization take a look at figure 20 that shows the total amount of partitions at each iteration. Note that the graph for the initial implementation still increases rapidly and will still cause errors.

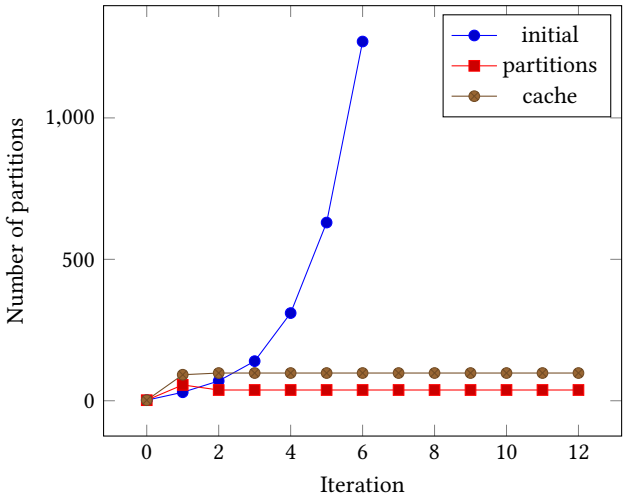


**Figure 20: The graph shows the number of new partitions created in each iterations.**

## 7.3 Number of partitions and TTL

When we now had an implementation capable of processing the wiki dumps it was now time to find the optimal combination of number of partitions and how long variables should be kept alive, the TTL. At first we simply didn't have enough resources to process the entire dataset so we worked with half the dataset, a dataset of 15 GB. Because running the algorithm on the dataset multiple times for each configuration is very time consuming and our time with

the cluster was limited we only got to run the algorithm a couple times per configuration, sometimes only once.

Initially we had a cluster consisting of 3 worker nodes each with 2 CPU cores and 7 GB of RAM. The test for parameters under these conditions are shown in table 8

| num partitions | TTL (in secs) | Execution time |
|---|---|---|
| 12 | 1200 | 2h52m42s |
| 12 | 900 | 2h51m30s |

**Table 8: Test of Subset (15GB) with 3 nodes**

Before being able to run more test we got a more powerful cluster, so we went on to finding more optimal setting for this cluster while we had it. Our new cluster consisted of 1 driver node and 6 worker nodes each with 2 cores and 7 GB of RAM. We first ran the test on the subset to see if the execution time improved without changing the parameters, indicating that our algorithm scales well to bigger clusters. The result of which can be seen in table 9.

| num partitions | TTL (in secs) | Execution time |
|---|---|---|
| 12 | 900 | 55m18s |
| 24 | 900 | 34m57s |
| 24 | 240 | 46m26s |
| 24 | 500 | 34m36s |
| 24 | 600 | 34m24s |
| 24 | 1800 | 46m35s |
| 48 | 500 | 27m25s |
| 96 | 500 | 33m50s |

**Table 9: Subset(15GB), 6 nodes, 12 cores, 42GB RAM**

From these results we can see that with the same parameters as before the execution time is now only one third of what it was. Intuitively you might think that it should only be halved, but when Spark doesn't have enough memory to load the entire dataset into memory it temporarily store some of the data to disk. When increasing the amount of memory of the cluster we allow for more of the data to loaded into memory instead of to disk, this would account for the level of improvement we see here.

We found that for our cluster the best parameters values were 48 partitions and TTL as 500 seconds. The number of partitions, 48, is four times the amount of worker cores. And a TTL of 500 is approximately twice the execution time of an iteration.

### 7.4 Test with full dataset

Having found the optimal parameters, it was time to test with the full dataset. Because of some unexpected server downtime, we were only able to run the full English wiki dump successfully through the PageRank algorithm once. When we were not able to successfully run the TF-IDF algorithm with this cluster, and when our time with the cluster ran out we transitioned over to the Norwegian wiki dumps. A much smaller dump of size 2.4 GB. The result from running the both datasets are shown in table 10.

| dataset | nodes | iterations | Execution time |
|---|---|---|---|
| enwiki | 6 | 10 | 56m23s |
| nowiki | 2 | TBD | TBD |

**Table 10: Final trial**

## 8 LIMITATIONS

This section details the various limitations we faced when working on this project.

Our first major limitation was a lack of a true cluster for running MRJob code. This directly resulted in MRJob being a second priority compared to Spark and that our preprocessing was also implemented in Python. While a local MRJob cluster did not perform too badly compared to python for preprocessing, (even beating python for preprocessing a single file, timewise), it was frankly not worth it to use it over python. The MRjob implementation was more logically complex and produced worse results. It produced worse results because it is limited to processing one line at a time. This is especially bad for a XML document where a tag usually stretches over multiple lines. Unless the MRJob has a very complex control structure, a python implementation will give better results. A MRJob implementaton is worth it if it can save you a large amount of processing time. A lack of a MRJob cluster also prevented us from running our MRJob implementation of TF-IDF on a full dataset as a local cluster was not sufficient to run either the English Wikipedia dataset or the Norwegian Wikipedia dataset, only samples. Though as we saw from the results of our spark implementation of TF-IDF, this might have more to do with optimizing our TF-IDF algorithm.

To tune the performance, we have to take into considerations many different combinations of Spark configurations. Number of cores, memory per executor, variables time to live. Another aspect is that testing each set of parameters can be very time consuming, and running the test multiple times to get stable and reliable results might not be feasible when we have a time limit with the different clusters.

## 9 FURTHER WORK

This section will discuss further work than can be done to optimize our algorithms and get better and faster search results. There were a few things we wanted to do to improve our project, but because of limitations with time and resources we were not able to.

If we had the time, we would try to optimize our TF-IDF algorithm. The main problem of our implementation was that it required too much memory from our clusters. To alleviate this, we could split up TF-IDF into one implementation for TF and one for IDF. This would reduce the memory cost of a single running implementation, but it would however require more space on the hard drive, as duplicate information would have to be stored for TF and IDF. We could also do partial calculations of TF and IDF on smaller samples of the data set and later try to stitch the results together. This would require starting with one result file and growing that file larger, file by file, never holding too much in memory at once. **d** and **D** would have to be recalculated for each new file. Implementing these changes would increase processing time and code complexity for our TF-IDF algorithm. In the case of TF and IDF being stored

separately, it would also increase the processing time for our search engine as it would have to look for information needed together in two separate locations.

One thing that can be done that would most likely improve the accuracy of the search results would be to implement n-gram-range into TF-IDF. That way we could search on not just individual words, but also consecutive words which would help get a better picture of the relevance of the page. The theory being that if the search query is a sentence, then the placement and order of the word matter, and n-grams would help with determining this. As we mentioned earlier the output of the TF-IDF algorithm was twice the size of the input causing us to be unable to run the TF-IDF algorithm on the English wiki dumps. Using n-grams would increase the size of the TF-IDF manifolds, and there was no way we could implement this with the resources we had available.

A second thing could be when calculating PageRank to instead of removing the *sink pages* to count them as linking to all other pages. This is something that we didn't have time to find a way to implement. And since it would require a higher level of overhead throughout the iterations necessary for calculating the PageRank, it may not even be feasible to do with the resources we had available.

## 10 CONCLUSION

A search engine is essential when retrieving information from a large dataset. Using the TF-IDF and PageRank algorithms was very useful when trying to retrieve and rank information. Implementing TF-IDF and PageRank in Spark allowed for the algorithms to be effectively scaled to work with computational clusters. TF-IDF requires a lot of memory and can be hard to implement. The preprocessing is a key step in making the algorithms more accurate.

We created this search engine for searching through Wikipedia articles, but the algorithms can be applied to any linked dataset with text content.

## REFERENCES


[1] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30, 1-7 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[2] Oxford Dictionary. 2019. Search Engine definition. https://en.oxforddictionaries.com/definition/search_engine

[3] Legoktm Earwig with contributions from Îč et al. 2019. mwparserfromhell. https://github.com/earwig/mwparserfromhell

[4] Python Software Foundation. 2019. Support for SAX2 parsers. https://docs.python.org/2/library/xml.sax.html

[5] Google. 2019. Facts about Google and Competition. https://web.archive.org/web/20111104131332/https://www.google.com/competition/howgooglesearchworks.html

[6] Stefan Langer Corinna Breitinger Joeran Beel, Bela Gipp. 2016. Research-paper recommender systems: a literature survey. https://kops.uni-konstanz.de/bitstream/handle/123456789/32348/Beel_0-311312.pdf?sequence=1&isAllowed=y

[7] Wikipedia. 2019. MediaWiki. https://www.mediawiki.org/wiki/MediaWiki

[8] Wikipedia. 2019. PageRank. https://en.wikipedia.org/wiki/PageRank

[9] Wikipedia. 2019. tf-idf. https://en.wikipedia.org/wiki/Tf%E2%80%93idf

[10] Wikipedia. 2019. WHOIS. https://en.wikipedia.org/wiki/WHOIS

[11] zero323. 2019. Spark iteration time increasing exponentially when using join. https://stackoverflow.com/questions/31659404/spark-iteration-time-increasing-exponentially-when-using-join

## 11 RESULT FROM SEARCH QUERIES

| hund | | |
|---|---|---|
| | page | search rank |
| 1 | Chihuahua | 5.88e-07 |
| 2 | Store hund | 2.72e-07 |
| 3 | Den lille hund | 2.65e-07 |
| 4 | Kategori:Store hund | 2.30e-07 |
| 5 | Kategori:Den lille hund | 1.99e-07 |
| 6 | Hofteleddsdysplasi | 1.75e-07 |
| 7 | Vesle hund | 1.54e-07 |
| 8 | Akita | 1.45e-07 |
| 9 | Den lille hunden | 1.38e-07 |
| 10 | Kina | 1.38e-07 |

**Table 11: Top ten search results for 'hund' out of 1,294 found in 0.00299s**

| potet | | |
|---|---|---|
| | page | search rank |
| 1 | Sverige | 9.30e-07 |
| 2 | India | 1.89e-07 |
| 3 | Amandine (potet) | 1.03e-07 |
| 4 | ÃŸstre Toten | 9.69e-08 |
| 5 | Fylkesvei 918 (Nordland) | 7.18e-08 |
| 6 | Kerrs Pink | 7.07e-08 |
| 7 | Hasselbackpoteter | 6.30e-08 |
| 8 | HjortetrÃŸffel | 5.25e-08 |
| 9 | Rotrasper | 5.04e-08 |
| 10 | SjÃŸmannsbiff | 4.71e-08 |

**Table 12: Top ten search results for 'potet' out of 164 found in 0.0000s**

| Norge | | |
|---|---|---|
| | page | search rank |
| 1 | Norge | inf |
| 2 | Kategori:Norges fylker | 0.00034 |
| 3 | Kategori:Byggverk i Norge | 7.30e-05 |
| 4 | Kategori:Veier i Norge | 6.79e-05 |
| 5 | Kategori:Undernasjonale omrÃĕder i Norge | 4.98e-05 |
| 6 | Kategori:Kommuner i Norge | 4.82e-05 |
| 7 | Kategori:Landformer i Norge | 4.59e-05 |
| 8 | Kategori:Sport i Norge | 3.74e-05 |
| 9 | Kategori:Fylkesveier | 3.73e-05 |
| 10 | Kategori:Transportbyggverk i Norge | 3.73e-05 |

**Table 13: Top ten search results for 'Norge' out of 73320 found in 0.13168s**

| USA | | |
|---|---|---|
| | page | search rank |
| 1 | USA | inf |
| 2 | Kategori:USAs delstater | 6.74e-05 |
| 3 | Kategori:Personer fra USA | 4.41e-05 |
| 4 | Kategori:Personer fra USA etter beskjeftigelse | 2.95e-05 |
| 5 | Kategori:Skuespillere fra USA | 2.91e-05 |
| 6 | USA | 2.89e-05 |
| 7 | Kategori:Undernasjonale omrÃĕder i USA | 2.39e-05 |
| 8 | Kategori:Personer etter fÃŸde- eller oppvekststed i USA | 2.27e-05 |
| 9 | Kategori:OL-deltakere for USA | 2.25e-05 |
| 10 | Kategori:SportsutÃŸvere fra USA | 2.04e-05 |

**Table 14: Top ten search results for 'USA' out of 60170 found in 0.1047s**