Information Retrieval and Web Search - Project Phase 2

Lukas Pfahler Tejas Umakanth

March 12, 2014

1 Collaboration Details

Lukas implemented the web interface, the Lucene search engine backend and the Hadoop backend including the ranking algorithm used for Hadoop search as well as a MapReduce Job for reading aggregated JSON Input, creating a inverted index and outputting it in JSON files. Tejas worked on a MapReduce job to create a inverted index and used his implementation to compare the performance with Lucene indexing.

2 Architecture

2.1 Indexing

Our crawler stored each document in a separate JSON file named `{id}.json`. These files are roughly 8kb each, so we had to merge files into bigger files in order to process them with Hadoop efficiently. We create new, bigger JSON files by concatenating the objects from the small JSON files in a list, until we reach a file size of more than 10mb.

We wrote a custom `FileInputFormat` subclass for loading and parsing these aggregated JSON files. Like with our Lucene indexing strategy, we for each image extract the caption as well as all the comments and use this text data for indexing. We remove some stopwords as well as all punctuation (including hashtags). We noticed, that many captions are merely a list of hashtags and not a real text in the traditional IR sense. Thus our Hadoop Index doesn't index hashtags separately. Our FileInputFormat outputs key value pairs `(Text =>Text)`, where the key is the document id and the text is all extracted data.

Our MapReduce Job for creating an inverted index is pretty basic: The Mapper takes a pair $(id \mapsto doc)$ and for each $word \in doc$ maps $(word \mapsto id)$.

The Reducer then gets a list of document ids for one word and combines these document ids to a map from document ids to counts:

$$\{(word \mapsto id_i)\} \xrightarrow{reduce} \{(word \mapsto \{(id_i, \#wordCount)\})\}$$

The maps the Reducer produces are then written to JSON index files using our custom `FileOutputFormat` subclass. We considered writing all words to a single JSON index, which was inefficient for looking up a single word. Then we considered writing each word to a separate JSON index file, which is very inefficient in the Hadoop Job, because of the small file sizes. We decided on a solution in the middle; we create 1024 index files and decide which word to write to which file using a hash function. We understand that using a fixed number of files is a bad idea in terms of scalability, but decided for this technique because of its simplicity and because for our fixed-sized dataset it is very efficient, since we don't need additional index structures.

The index file has the following form:

```
{
    "word1" : {
        "documentCount" : 2,
        "documents": [
            {
```

```
            "document" : "id1",
            "score" : 1.0
            },
            {
            "document" : "id2",
                "score" : 2.0
            },
            ...
        ]
    },
    "word2" : {...},
    ...
}
```

2.2 Webinterface and Search Engine

The actual search engine and the web interface are two separate components; the web server is written based on node.js and the search engine is written in Java, they exchange information using a TCP connection, as seen in figure 1.

The search engine listens for connections on port 8999. To search, a client connects and writes 'x query', where x is either 'l' for Lucene or 'h' for Hadoop and query is the search query. Our search engine then performs the search using either our index created by Hadoop or our index created by Lucene. The results are returned in JSON format.

The Lucene search uses the default Lucene ranking algorithm. Our Lucene index indexed hashtags separately, thus each term #`term` in a query is replaced with `hashtags:term` to tell Lucene to look it up in the right index.

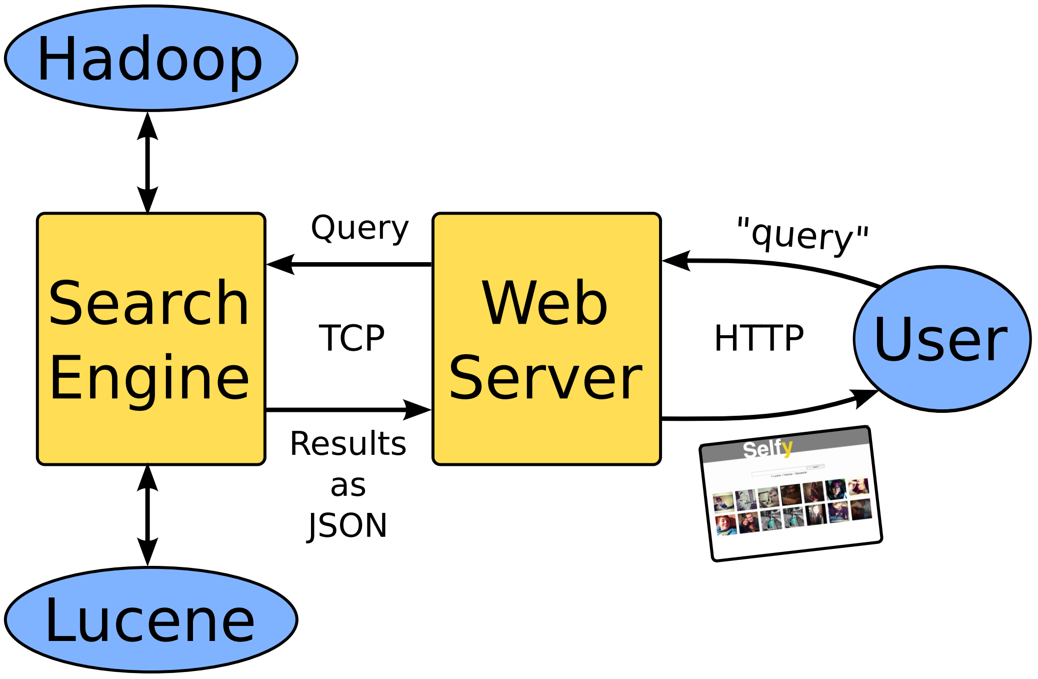


Figure 1: An overview of our search engine architecture.

For Hadoop-based search, we lookup each term of the query in the right index file by applying the hash function and parsing the right JSON file. This way we can retrieve a list of documents with word counts. For each query we maintain a map from document to score, containing all the documents retrieved for any term of the query. For the scoring function we use a simplified BM25.

For the web interface we decided to go with node.js again. We start a http server on port 8080 and wait for requests. Once a user enters a query, our webserver opens a TCP connection to our search engine backend, specifying whether to use Lucene or Hadoop and relaying the query. The backend executes the query and returns the results as JSON. The results are then output to the user by the web server as a grid of thumbnail images. Additional information like the caption and the comments for each image is loaded in an asynchronous fashion using AJAX and JSON when the user requests it by clicking on an image.

3 Usage

3.1 Indexer

To run the indexer execute the following command:

```
sh indexer.sh {dataDir}
```

where `{dataDir}` contains the aggregated JSON files. This will create a directory `HadoopIndex` with 1024 JSON index files.

3.2 Web Server

To start the web server on port 8080 execute the following command:

```
sh server.sh {dataDir} {hadoopDir} {luceneDir}
```

where `{dataDir}` contains the raw crawled data, `{hadoopDir}` contains the hadoop index and `{luceneDir}` contains the lucene index.

To shutdown the server properly, search for 'killtheserver'.

4 Performance

Tejas Part...

5 Screenshots

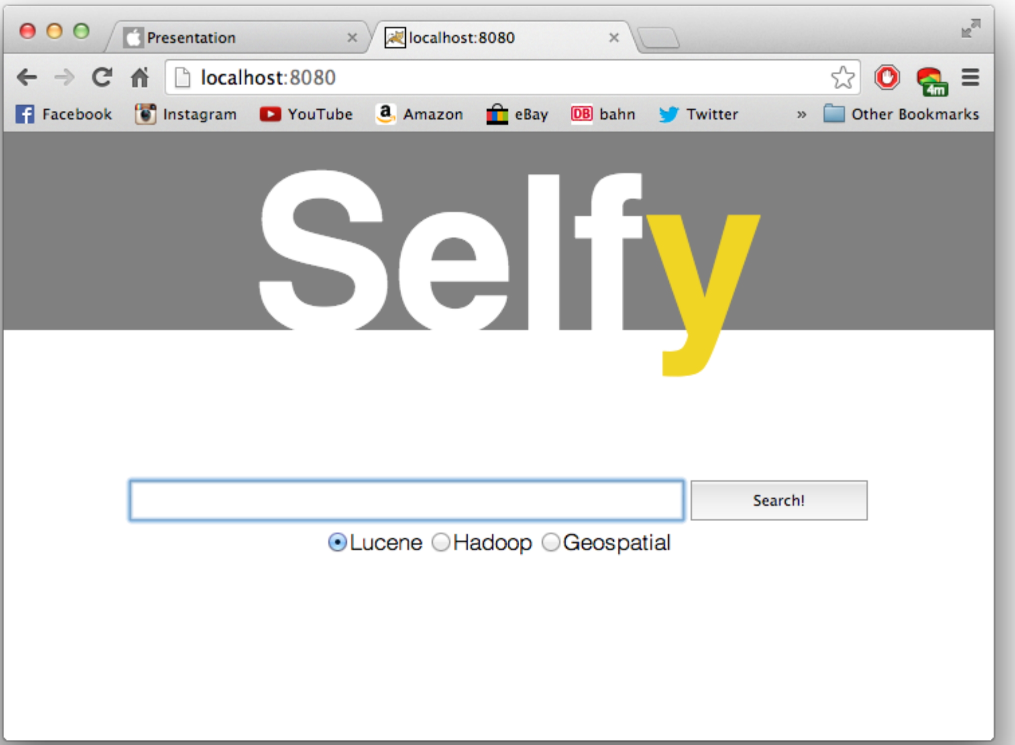


Figure 2: The search front page

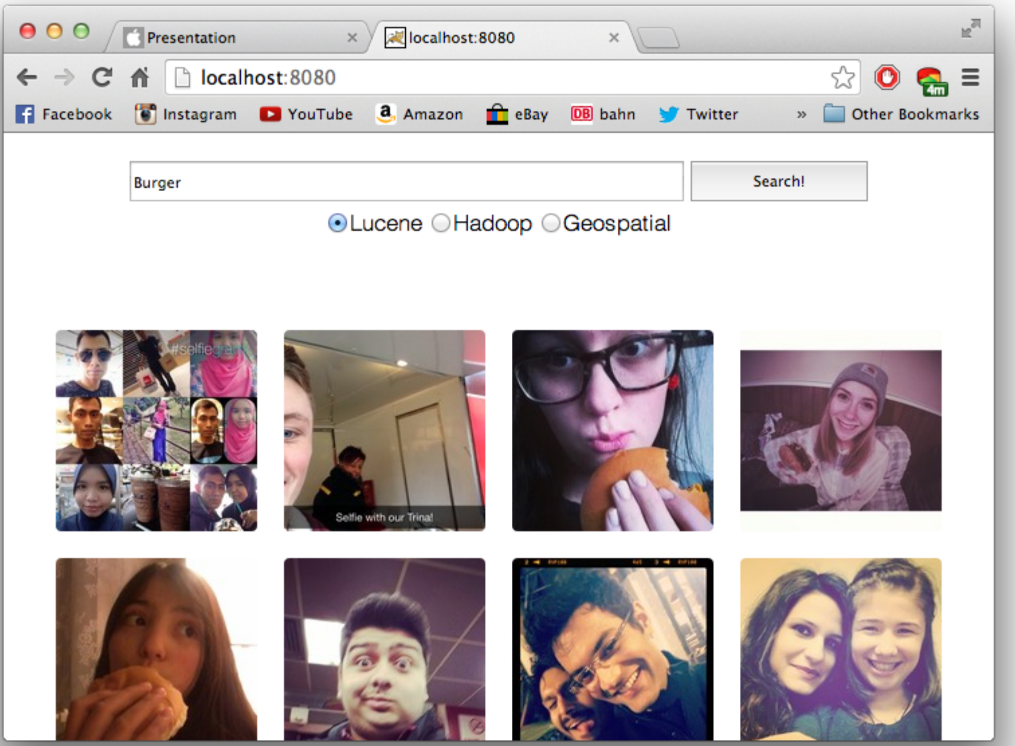
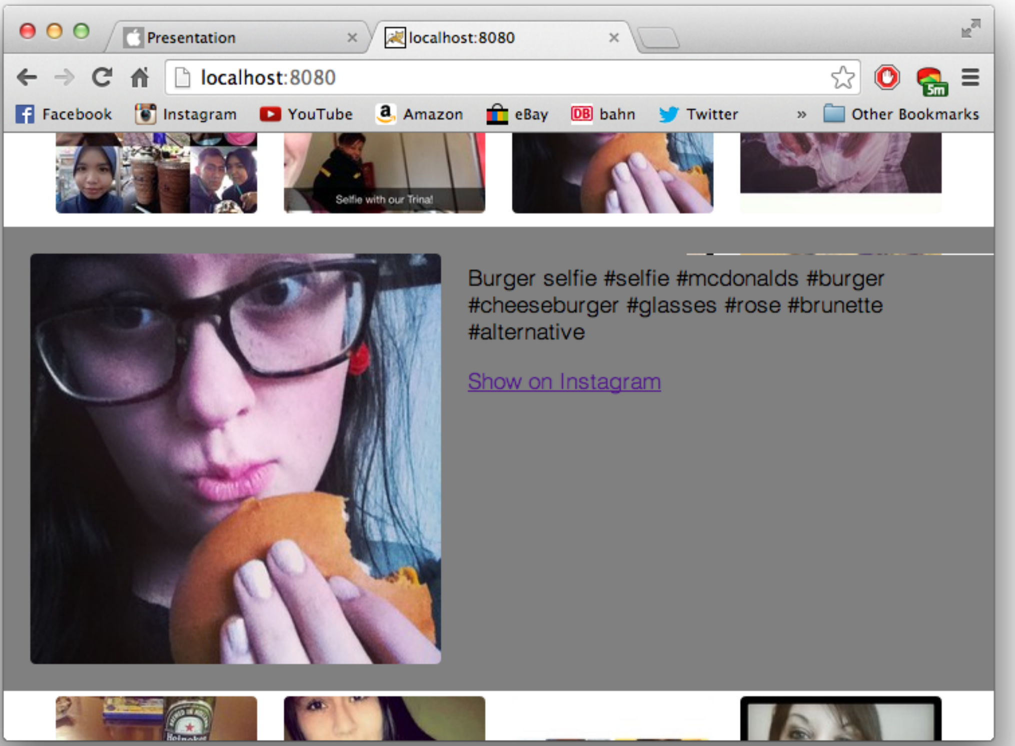


Figure 3: The results



Burger selfie #selfie #mcdonalds #burger #cheeseburger #glasses #rose #brunette #alternative

Show on Instagram

Figure 4: More details on one document