Search Engine for Structured Data

Ksenia Saenko
MS Data Science
ks4841@nyu.edu

Stella Sun
MS Data Science
ss8955@nyu.edu

Zhiwei Yu
MS Computer Science
zy1129@nyu.edu

ABSTRACT

The search engines are essential tools used to locate locate information for many people. For this paper, we are using NYCOpenData website (https://opendata.cityofnewyork.us/) as an example to show how to implement inverted index, ranking, and querying to make searching more productive and efficient.

The technologies we used were Hadoop Mapreduce and PySpark, and the goal of the project was to enable not only table name searches, but also column search, content search, and topic search, with customized filters.

CCS CONCEPTS

• **Database management system engines** • **Database query processing**

KEYWORDS

Search Engine, Data Cleaning, Database Design, Inverted Index, Jaccard Similarity, TF-IDF

ACM Reference format:

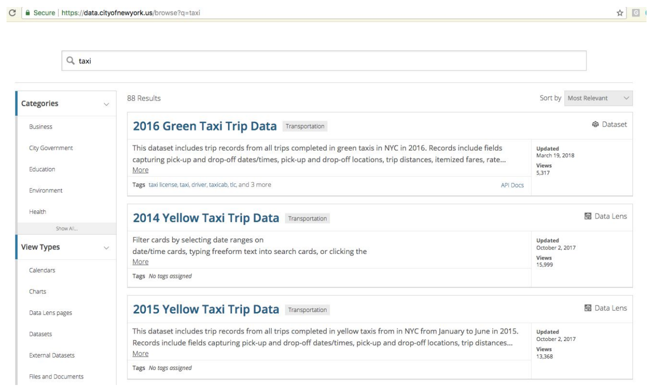
G. Gubbiotti, P. Malagò, S. Fin, S. Tacchi, L. Giovannini, D. Bisero, M. Madami, and G. Carlotti. 1997. SIG Proceedings Paper in word Format. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97)*, 4 pages. https://doi.org/10.1145/123 4

1 INTRODUCTION

Problem: Efficiency and quality of any data project depends on the ability to accurately and timely obtain the necessary information. The current search capabilities of Open Data NYC portal are limited to search over table titles (See Fig. 1). A user cannot search contents of tables, such as specifying a column name or keyword to retrieve from the rows. A user also cannot filter the search by table properties, such as a number of rows in a table. The current interface is inconvenient for the user and results in slower workflow and inability to find all relevant tables.

The goal of this project is to build a search engine for structured data with improved query capabilities for the NYC Open Data portal https://opendata.cityofnewyork.us/

Our search engine will allow users to query the contents of a large collection of data sets using structured queries.
Users will be able to specify their search criteria. For example, they could specify custom query output, such as 1. table name contains keyword "new york"; 2. table with columns contain "address"; 3. table content that contain "1988"; 4. a more general search topic, like, table that is related to "traffic". In addition, we will provide filters on the final output to take table size and other properties of table instances into consideration for our search engine. To give the user a more complete result, we are not only showing the table name, but also the table category, and table description.



2016 Green Taxi Trip Data Transportation
This dataset includes trip records from all trips completed in green taxis in NYC in 2016. Records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate...
More
Tags taxi license, taxi, driver, taxicab, tlc, and 3 more API Docs
Updated March 19, 2018
Views 5,317

2014 Yellow Taxi Trip Data Transportation
Filter cards by selecting date ranges on date/time cards, typing freeform text into search cards, or clicking the
More
Tags *No tags assigned*
Updated October 2, 2017
Views 15,999

2015 Yellow Taxi Trip Data Transportation
This dataset includes trip records from all trips completed in yellow taxis from in NYC from January to June in 2015. Records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances...
More
Tags *No tags assigned*
Updated October 2, 2017
Views 13,368

Fig.1: Search result in NYC Open Data. The platform only searches for keywords across titles.

2 ARCHITECTURE AND DESIGN

2.1 Method Design

A general approach to the problem is shown in Figure 2.

To process raw json data stored in HDFS, we processed both metadata and data tables. Metadata was used to generate **Master_Index** table which contained Doc_ID, Table_Name, Table_Length, **Table_Descr** contained Doc_ID, Category and Table_Description, and **Tag_Table** was used to generate tag_inverted_index; the data tables were used to generate four inverted index tables: **Title_Index, Column_Index, Content_Index and Tag_Index.**

A clear explanation of how data was processed is discussed in Section 3. **DATA PROCESSING**

In the python application, we will prompt users for their inputs, and then all files and user input will be passed into Spark.

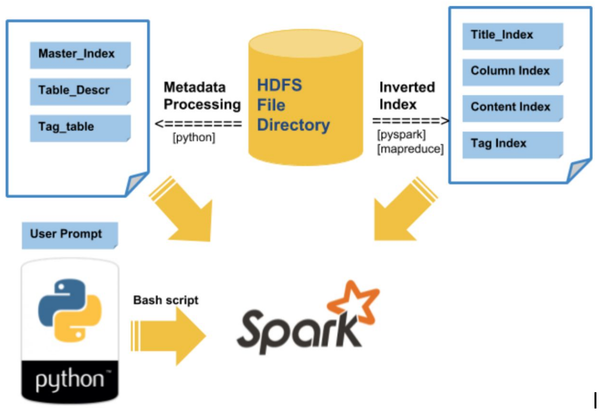


Fig.2: Method Design

2.2 Application Design

A general process of how the application is working is shown in Fig.3.

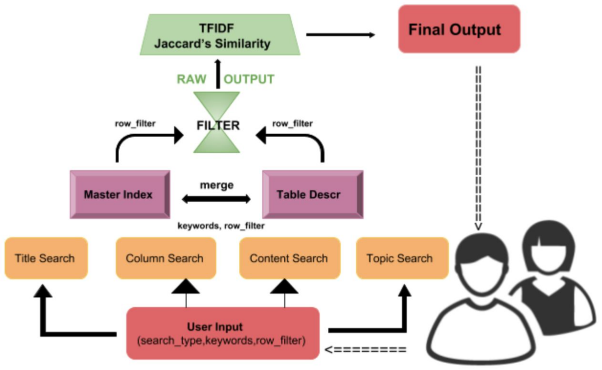
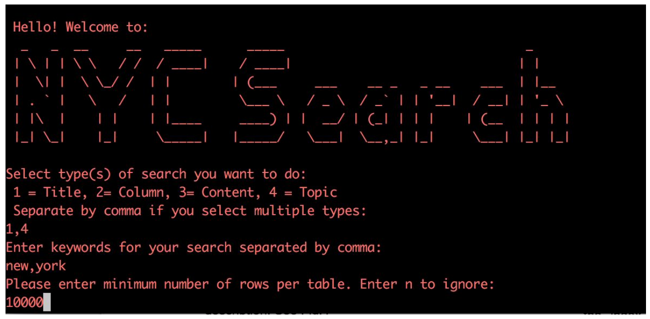


Figure 3: Architecture of the search engine.

User will be asked to enter three inputs: *search_type, keyword, row_filter*, an example of user prompt in python is the following:



```
Hello! Welcome to:
 _   _ __   __ _____   _____                        _
| \ | |\ \ / // ____| / ____|                      | |
|  \| | \ V / | |     | (___    ___   __ _  _ __   ___ | |__
| . ` |  \ /  | |      \___ \  / _ \ / _` || '__| / __|| '_ \
| |\  |  | |  | |____  ____) ||  __/| (_| || |   | (__ | | | |
|_| \_|  |_|   \_____||_____/  \___| \__,_||_|    \___||_| |_|

Select type(s) of search you want to do:
 1 = Title, 2= Column, 3= Content, 4 = Topic
 Separate by comma if you select multiple types:
1,4
Enter keywords for your search separated by comma:
new,york
Please enter minimum number of rows per table. Enter n to ignore:
10000
```

Fig.4: Sample user prompt

With each search_type, the corresponding search function will be called in Spark. Key_words parameter will go to corresponding inverted_index table to get a list of Doc_IDs and then the Master_Index table will take the Doc_ID and return the table name. In addition, with Doc_ID as primary key, and foreign key, Master_Index table is then merged with Table_Descr table to get the table category and table description.

A simple filter function will be applied to filter out the results based on row_filter input from users, and then raw output is handled by ranking algorithms, TF-IDF and Jaccard Similarity, to generate final output and pass back to the user.

How ranking algorithms are working in the application will be discussed in Section **4.4 Similarity Score and Ranking.**

3 DATA PROCESSING

All datasets from NYC Open Data were stored on HDFS in JSON format. In order to work with the data efficiently in PySpark, we first extracted the necessary fields from JSON files. The challenging part was that the JSON files were large in size (the largest file being 17 GB) and all datasets combined were around 650 GB large. Due to the unstructured format and special encoding, it was difficult to process files in PySpark directly. Therefore, we created a program in Python to read one JSON file at a time, extract metadata and data, and save them in CSV format which was then more efficient to process with PySpark. In this fashion we processed a sample of 50 files (~15 GB in total), with the largest file of ~3GB in size. After we saved all files in CSV, it became very efficient to load them into PySpark for index development.

3.1 Metadata

For each file we extracted the following metadata:
table_title, column_names (schema), table_description, tags associated with the table. We also calculate the number of rows per table in pyspark. This information was later used to build the Master_Index file. Metadata was also used in building column_index and tag_index.

3.2 Raw tables

We extracted raw table data from JSON files and saved each table as a CSV file. That information was later used for creating content index.

4 METHODS

4.1 Sample Fabrication

To simulate the problem for concept testing and development we obtained a sample dataset from NYC Open Data website using Socrata API. We downloaded tables from five categories: business, education, government, environment, and health in order to have a representative sample. We also queried metadata about each table, including the number of columns and title for later processing. We used Python to create a table with table ID's matched to metadata to allow for efficient querying and filtering in later stages of the project. We uploaded fabricated data to Hadoop. Once we were satisfied with the performance of methods on the fabricated sample, we then applied them on a larger scale sample of ~15GB size.

4.2 Inverted Index

An inverted index is a data structure common to nearly all information retrieval systems. In our experiment, we created inverted index for three types of search: **title search** where user can search by table names; **column search** where user could search by individual columns from all tables in the system; **content search** that the search goes through all table contents, and **topic search** will allow user to search by a general topic.

Figure 6 below describes a general approach for creating inverted index using key-value pairs for MapReduce concept.

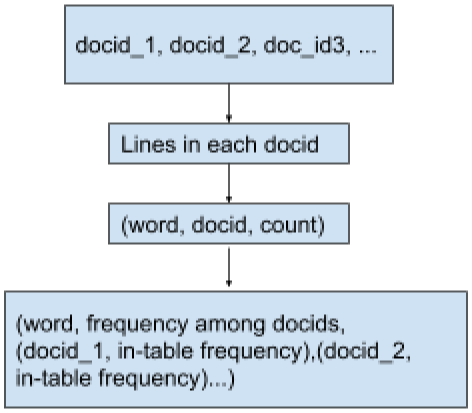

docid_1, docid_2, doc_id3, ...
Lines in each docid
(word, docid, count)
(word, frequency among docids, (docid_1, in-table frequency),(docid_2, in-table frequency)...)

Fig.6 Process to generate inverted indices

4.2.1 Title Index.
Our first goal was to build an inverted index for title search. After initial research we decided on the following design of the index table: key will be the keyword, first column represents a number of documents containing the keyword, third column contains a list of document ID's containing the word, sorted by the frequency of the occurrence. This format is efficient for later search and ranking as it allows to sort the index by the keyword frequency. Hence searching for a word in the index is faster for more frequent words.

We implemented two version of creating the title index - one with PySpark and one with Hadoop. This way we could determine the most efficient method. We eventually decided to proceed with the PySpark method because it was more efficient and easier to implement. PySpark also has built-in functionality for querying that we used later for processing user input. It also has a machine learning library that we leveraged for ranking.

4.2.2 Column Index.
We then built an inverted index for the column names of each table: the keyword is the specific word we will be looking for in the column names sorted by the frequency in

the dataset and the corresponding index is the document ID along with its column name. This process is implemented by MapReduce, made parallel and thus efficient as the preprocessing of the user query search.

4.2.3 Content Search.

To create inverted index for content search, we used MapReduce to go through all tables and took each word as a key, with a (table_name, count) as value. The key, value pair was reduced as (key, frequency among tables, table_name) with table_name sorted by in-table frequency. The result was then sorted and printed by frequency in the descending order. We processed all contents in the following manner: first, all table contents were converted to strings. We tokenized all sentences, such that the sentence “New search engine” would become a list of words “new”, “search”, “engine”. We removed all stop words (like “a”, “the”, “an”, etc) and punctuation marks (such as commas). We then applied Porter Stemmer algorithm to all words such that the words with the same root will be treated as the same vocabulary term. For example, the words “search” , “searchable”, and “searched” would be converted to the same word “search”. Although numeric data was converted to strings, we assumed most users would not be interested in searching for a particular number. At the same time, we wanted to allow users a functionality to look up a numeric value if necessary. Hence, we applied a shrinkage factor to all numeric keywords so that they would appear at the bottom of the content index.

4.2.4 Topic Search.

A tag table was generated when processing the metadata. Each table was tagged with a few words, and then the tag table was used to generate an inverted index table in spark in the same way as other inverted index tables.

We implement the topic search to allow the user to search in a more general way, and since the tags are pre-generated for each table, the tags were not just 1-gram as other inverted indices, but n-grams, such as “new york taxi”,”traffic problems 2017”, etc., which provided a more efficient search for users.

4.3 Query Design

A simple Python application was built to take user input, and a spark job was submitted through bash script, and passing all required parameters into spark application.

In spark, user queries was converted into spark.sql queries to execute some table join and filter, and then returned the raw output.

4.4 Similarity Score and Ranking

We ranked the raw results by relevance based on two metrics, namely term frequency inverse document frequency (TF-IDF) with vector space model, and Jaccard similarity.

4.4.1 TF-IDF

The term frequency is the number of occurrences of a term in a document. It describes how frequent a term is in a document.

$$tf_{t,d} = N_{t,d}$$

Term frequency considers all terms equally important. However, some terms occur more rarely and they are more discriminative than others. For example, the word “taxi” should have more importance than the word “year” in the document because it contains more information about what a document is about. As an extreme case, the stop words can be eliminated in the preprocessing steps but still a metric to distinguish the discriminative terms among others is needed. This problem can be solved by introducing the concept of document frequency.

Document frequency is the number of documents containing a term.

$$df_t = N_t$$

The opposite concept, inverse document frequency, defined as below, describes how discriminative a term is in a corpus.

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1}$$

After we applied this algorithm, we calculated the TF-IDF score, which was simply the product of term frequency and inverse document frequency.

4.4.2 Vector Space Model

We used the vector space model to represent documents as vectors and made them lie in the same vector space. For each document, the entries corresponding to the terms that appeared in the current document would be their TF-IDF scores. The entries corresponding to the terms that did not appear in the current document would be zero because their term frequencies would be zero since they do not occur.

We also represented the query as a vector in the same vector space as the documents. Since now both the query and the documents were represented as vectors in a common vector space, we took advantage of this by computing the cosine similarity between the query vector and all the document vectors, which could be a good measure to rank these documents based on how relevant each document is to the query.

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D).$$

Since the dimensions of the documents and the query would usually be much smaller than the dimension of the vector space generated from them, in the end we would have a lot of space vectors with most of the entries being zero. This made saving these vectors a relief after we turn to the spare vector data structure.

We calculated the TF-IDF scores from the inverted indices. Both the term frequencies and the document frequencies can be easily obtained from the inverted indices. We constructed four TF-IDF models, corresponding to four different types of search from four inverted index files, and saved them into files.

When we got the input query, we parse it in the same way as we parse the documents. Then we read the corresponding model files and convert the query into a vector in the same vector space as the documents. Then we calculated the cosine similarities, ranked the results by scores in the descending order, and passed back the output to the user.

4.4.3 Jaccard Similarity

We implemented ranking using Jaccard similarity for title search. Before applying Jaccard similarity, the words were stemmed and lowercased. Jaccard similarity was calculated based on the proportion of shared words between two sets. So for example, if a keyword search is "green taxi" and the table title is "green taxi NYC", we can see that two out of three unique words are shared. Hence Jaccard similarity is 0.75. The higher the score, the more similar are the sets. Titles will then be ranked by their Jaccard similarity score before printing out the output. The formula for calculating Jaccard similarity is below.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

One of the benefits of Jaccard similarity is that it is efficient to calculate and does not require a lot of resources. In case of search over title, the length of titles is relatively small - and hence the scores are fast to calculate.

5. TESTING AND VALIDATION

We tested accuracy and breadth of results compared to searching in NYC Open Data website. We timed our queries to ensure efficient performance. We also compared our Hadoop vs. Spark methods and determined that Spark resulted in the best performance in terms of time and difficulty in implementation.

6 RESULTS

6.1 Final Results

With 47 files processed, we created four indices files: title index, column index, content index and tag index; in addition, we also extracted *Category* and *Description* for each table from metadata. We have two methods for creating inverted index: MapReduce in Hadoop and Spark; comparing the time performance and difficulty level to implement, we decided to go for Spark. We determined it to be more efficient than Hadoop.

We also built a simple Python application to take user input and pass all parameters into Spark, and we created four search functions, as well as ranking models using TF-IDF and Jaccard Similarity score. We used SQL functionality in PySpark to perform queries and return a final result to user.

Our search engine resulted in a number of improvements upon the NYC Open Data search engine. In particular, we provided flexibility to the user to perform custom search by title, columns, content and topic. We also let the user filter out results by size of the table.

Because our inverted indices were implemented in PySpark, the same code could handle more data allowing for scalability. Additional applications, such as a website, could be implemented on top of the code.

6.2 Github Repository

https://github.com/stella0316/Search_Engine

6.3 Demo Video

https://youtu.be/Mko2MV8vg0o

7 CONCLUSIONS

In summary, we have built a simple but complete search engine application in Spark, which innovated upon the current capabilities of NYC Open Data website - in particular, it allows a user to search not only by table names, but also by column names, table content, and topics, in addition, user can limit the table size by customized filters -- which is currently unavailable at the website.

By implementing ranking algorithm, all results are sorted by scores, and our TF-IDF model and Jaccard similarity function both worked well in calculating and ranking the results.

8 DISCUSSION

This project allowed us to build a deeper understanding of search engine fundamentals. One of the biggest challenges was obtaining and processing large amounts of data from JSON to CSV format. We recognized that while powerful, PySpark can perform much more efficiently on text and csv files compared to JSON.

While experimenting with PySpark and Hadoop implementation of inverted indices, we appreciated the efficiency and higher functionality of PySpark compared to Hadoop. Furthermore, implementing the queries and transforming the data into a vector format for TF-IDF ranking would be more time consuming and challenging in Hadoop.

We implemented two methods for ranking - using PySpark machine learning library and TF-IDF and using Jaccard similarity, which is calculated directly. One of the methods could be chosen based on applications of the final product.

A.1 Introduction

A.2 Architecture and Design

A.2.1 Method Design

A.2.2 Application Design

A.3 Data Processing

A.3.1 Metadata

A.3.2 Raw tables

A.4 Methods

A.4.1 Sample Fabrication

A.4.2 Inverted Index

A.4.3 Query Design

A.4.4 Similarity Score and Ranking

A.5 Testing and Validation

A.6 Results and Discussion

A.6.1 Preliminary Results

A.6.2 Github Repository

A.6.3 Demo Video

A.7 Conclusion

A.8 Discussion

A.9 References

REFERENCES

[1] Cafarella, M. J. et al. WebTables:Exploring the Power of Tables on the Web.
[2] Dasu, T. et al. Mining Database Structure; Or, How to Build a Data Quality Browser.
[3] Agrawal, S. et al. DBXplorer: A System for Keyword-Based Search over Relational Database.
[4] Luo, Y. (n.d.). Spark:A Keyword Search Engine on Relational Databases.
[5] Rejaraman, A., & Ullman, J. (n.d.). *Mining of Massive Datasets*.
[6] Cloud9. (n.d.). Retrieved April 13, 2018, from https://lintool.github.io/Cloud9/docs/exercises/indexing.html