Web Search Engines – Project Report
Search Overflow

Narasimman Sairam (ns3184), Manasa Kunaparaju (mk5376)

Introduction:

A Query Engine to leverage the body of knowledge created by the social question answering system, to recommend high quality, embeddable code.

Category:
Specialized search engine*: Implementing a search engine that gives high quality results for some particular class of queries*.

Q&A services like stack overflow are filling archives with millions of entries that contribute to the body of knowledge in software development" and they often become the substitute of the official product documentation.

For instance, developer is writing code and has to check online for syntax or API related information, the developer would:

- Open the browser
- Google for the particular question (Ex: how to iterate over a map in Java)
- From the list of responses, (usually stack overflow) open couple of tabs of stack overflow pages that answers the query.
- Look for the best answer (Top voted in stack overflow website)
- Close the other tabs
- Copy paste the top rated answer to the development environment and continue working

Our search engine will do all the hard work of querying for the best answer and suggest the developer with the top rated response from the stack Overflow website. The answer to the query will be exactly one response (top rated).

Searching for code examples is possible using Stack Overflow directly. However, using designated code search tools on top of Stack Overflow may provide better

results in terms of streamlining the various activities involved in example centric development (search, evaluation, and embedding).

Dataset Preprocessing

The data has been obtained from [Stack Exchange Data Dump] [1]. It contains 309 zip files which contain user contributed content on many different topics. Only 7 of those contain information regarding Stack Overflow. These are Badges, Comments, Post History, Posts, Post Links, Tags, Users, and Votes. The schema for this data is provided [here] [2]. After extracting these files and analyzing the data, we decided to use the Posts data, which was in XML format, as it contained all the relevant information required to follow up with our project. The XML file was converted into SQL database using a helper script db.py. Each XML entry is converted into an entry in the database.

Each entry in the database are divided into two types. The ones with PostTypeId = 1 are the questions and the ones with PostTypeId = 2 are all the answers in the system. Each post has its authors information like ID, reputation etc. It also contains various post particular information like upvotes, title, body, comment count etc, which are compulsorily present, however other information like acceptedAnswer, FavoriteCount etc. are nullable fields. We also explored using the [Stack Exchange API] [3]. This contains the real-time data of the whole website. Each page can be queried by providing several options like question ids, answer ids, tags and many more presented in the website.

Id
PostTypeId (Question=1, Answer=2)
AcceptedAnswerId (not null if PostTypeId=1)
ParentID (not null if PostTypeId=2)
ViewCount (nullable)
Body
Title (nullable)
Tags (nullable)
AnswerCount (nullable)
FavoriteCount

Table 1: Posts Table

Architecture of the System

The below is an overview of architecture of the system.

Indexer: Creates and inverted index from which data will be retrieved. Each entry in the database has been converted into a lucene document for further processing. For this we query the database and populate a "Post" class with the required information like ID, score and answer ID. That class is in turn used when building the index. Indexed fields are Title, Body and Score. The index is built using these fields. Stored fields are AcceptedAnswerId, ViewCount and FavoriteCount. These fields are required for later computation, however we do not want these fields to be considered for indexing.

Custom Scoring function: Instead of using the default lucene scoring function, we have implemented our own custom scoring function.

This takes two utilities into consideration.

1. The lucene score, which is a measure of the extent of input queries matching with the Post's title and body.
2. The Post score.

$$Score\ (S) = \ (0.75)\ *\ Q(V)\ +\ (0.25)\ *\ L$$

where;

Q(V) = 1 / (1 + e^{-X})

X = Post.score (Votes)

L = Lucene Score

As both the scores are not comparable, lucene score being in unit digits and post score being arbitrary, we decided to normalize both the parameters. We ran trials to normalize the values by converting the votes into the rate of upvotes (votes/views) and trying different function for the same.

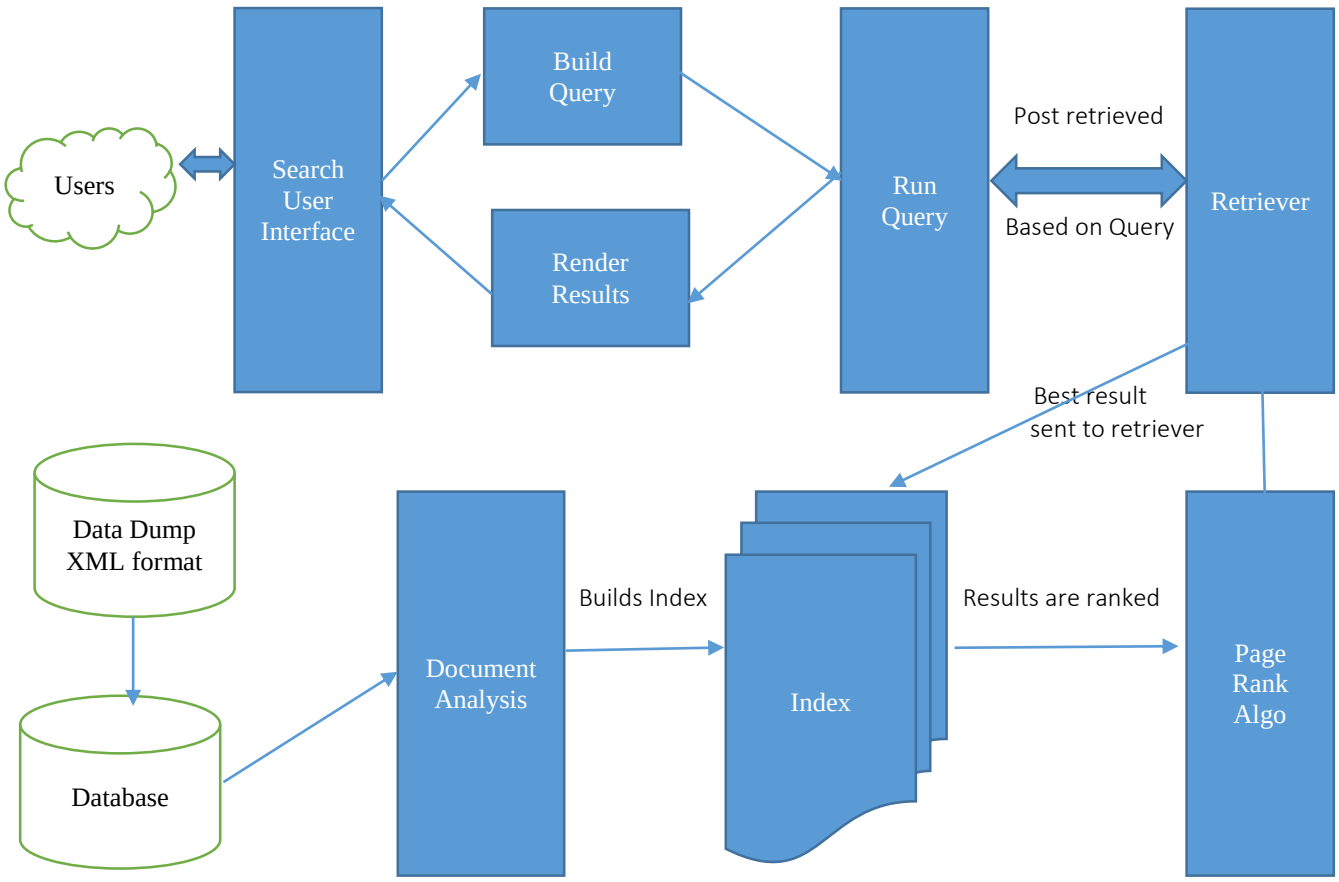
After experimenting we decided to use the sigmoid function, which is a unity-based normalization, to perform feature scaling, as it converts both the scores to the range [0, 1]. Both the normalized scores were given weights, which add up to a total of 1, with the lucene score given more weight as we primarily require a good textual match. The index has been created using this scoring function.

Retriever: Retrieves the best 100 documents based on the custom score, using the input query, from the index. These documents are processed by extracting the indexed and the stored values which are added to newly created post classes which are defined by the id number. The documents have earlier been processed irrespective of whether they are of PostTypeId = 1 or 2. Since we require only the best matched questions we add the documents with PostTypeId =1 and further filter down to only the questions with an AcceptedAnswerId. They are then added to a hashmap, with the key being the id and the value being the object, for ease of access.

The next part is retrieving the answers for these Posts. With our code, this can be done in two ways. As answers are already a part of the database, we can send a query to extract them based on the AcceptedAnswerId information already available. The other way is to use the StackExchange API to send a batch request to retrieve all the answers required. Presently we opted to using the local database over using the API due to a few issues which will be discussed later. The retrieved information is used to populate an Answer class based on the ids provided.

Ranker: Once the answer list is populated we require the one best answer that is being provided by the Ranker utility class. This class takes as input the map Posts, iterates through it and updates the score of the class. This score also takes into consideration the up votes of a particular answer. Although we contemplated considering using the user reputation, we decided it bears no weight on the score of the post, moreover our experiments supported our conclusion. The votes have been normalized using the previously mentioned sigmoid function and provided a weight. These classes have now been added to a *max heap based Priority queue* with the comparisons centered on the newly computed final score.

UI: The webpage where the query can be typed and the retrieved answer is shown to the user. The user interface is a simple HTTP Servlet and that calls the retriever interface and passes back the result for a given query.



List of external software used:

1. Apache Lucene – for indexing the pages
2. SQLite – to store the database of Posts
3. Apache Tomcat – hosting the web page
4. JSONParser – parse the JSON object obtained from StackExchange API.

Systematic Evaluation of the Project

This project has been undertaken with the idea of providing a programmer a perfect piece of code that is relevant to his query. Our ideal case is when the programmer asks for a certain algorithm in a particular language, we would provide a working implementation of the program in the required language.

We gauged the success of this search engine by comparing output to the results provided by Google. Once we typed in a query in Google, we noted down the results in the 1st page. We then queried in our search engine using the same terms. Listing our results, we noted that the top google results are present in our output with a very high lucene score.

Performance of the System:

- The size of Posts.xml is 42GB and was dumped into a database of the same size. After we ran the lucene Indexer on the database schema, the size of the index is 4GB.
- It contains 30 million entries and took on an average of 30mins to build the index.
- When retrieving the total time took from 4s to 15s depending on the number of documents that matched the query. Also, the retrieval from the indexer depends on the power of the hardware as well.
- When the application was deployed on high performance computing cluster of NYU (HPC nodes), the system performs several order of magnitude better.

Limitations of the System:

1. There are conditions where the system is could be improved. While retrieving the answer, we filter the posts based on the presence of an accepted answer. We have observed situations where there are good answers with a high number of upvotes and views but no accepted answer. Our search engine does not capture this answer at the moment.

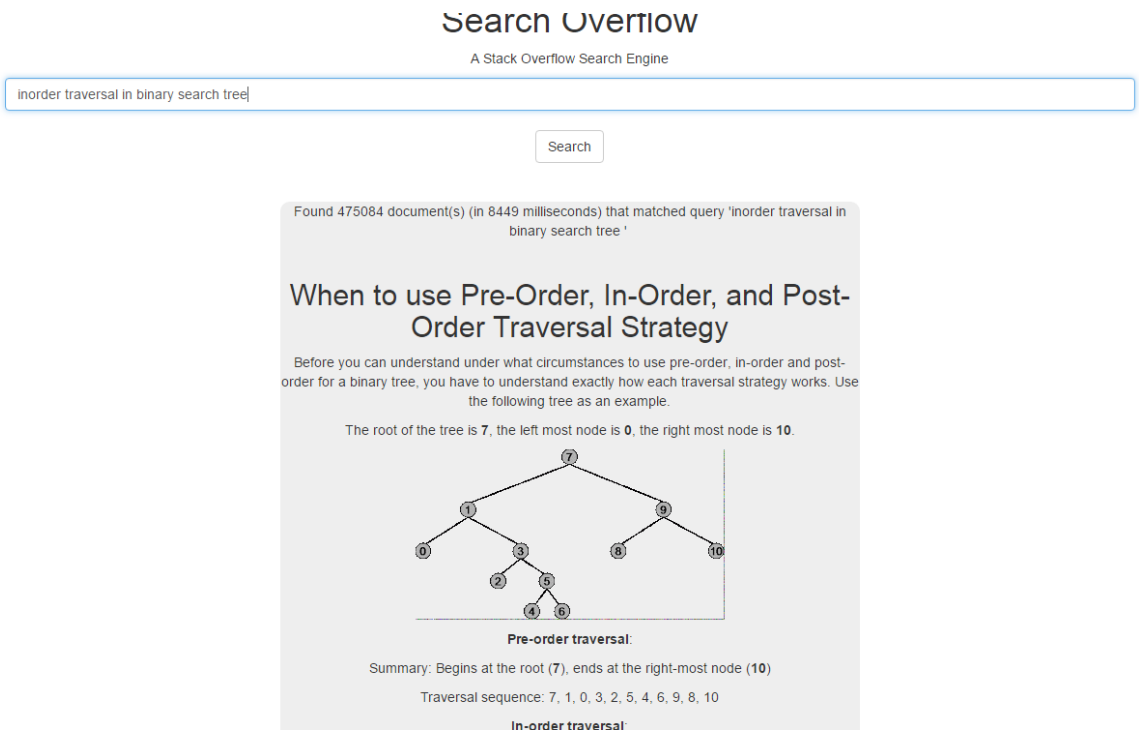
2. As we use the Stack Exchange data dump, our data set is not up to date. The data we use is last updated in March 2016. For this reason we initially opted to use the API. Ideally we would batch all our queries into one single query by appending all the answers together to the API. This would send us back a JSONobject and a JSONParser will be used to read the body of the answer. We do have this option available via a Boolean variable in the code for the function "populateAnswer" that can be used to turn this requirement on and off.

3. We use the lucene indexing functionality for the creation of our index. The query terms "hashmap" and "hash map" yield considerably different result.

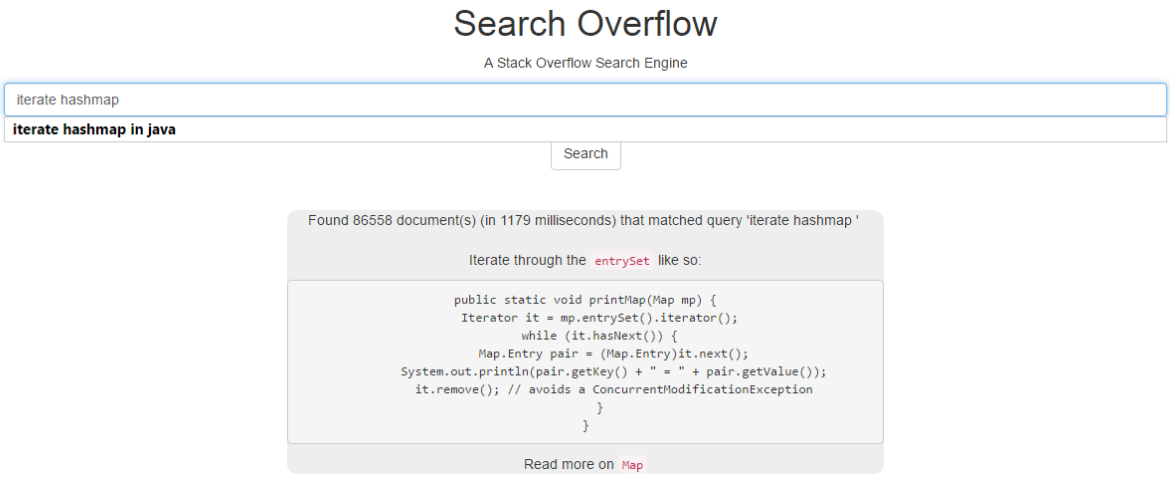
There needs to be a better keyword match in terms of language. The technical language needs to be better analyzed to eliminate these kinds of differences.

4. Another situation that needs to be addressed is the case where there may be a high number of votes for an answer and that could be the presented answer. However, it need not be the best answer.

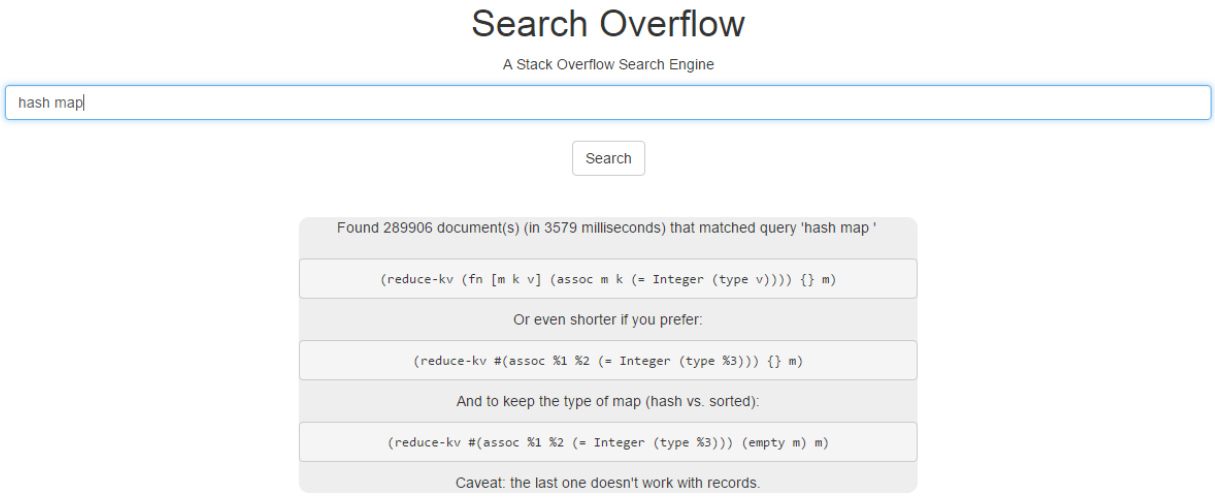
Examples of Project Success and Failure:



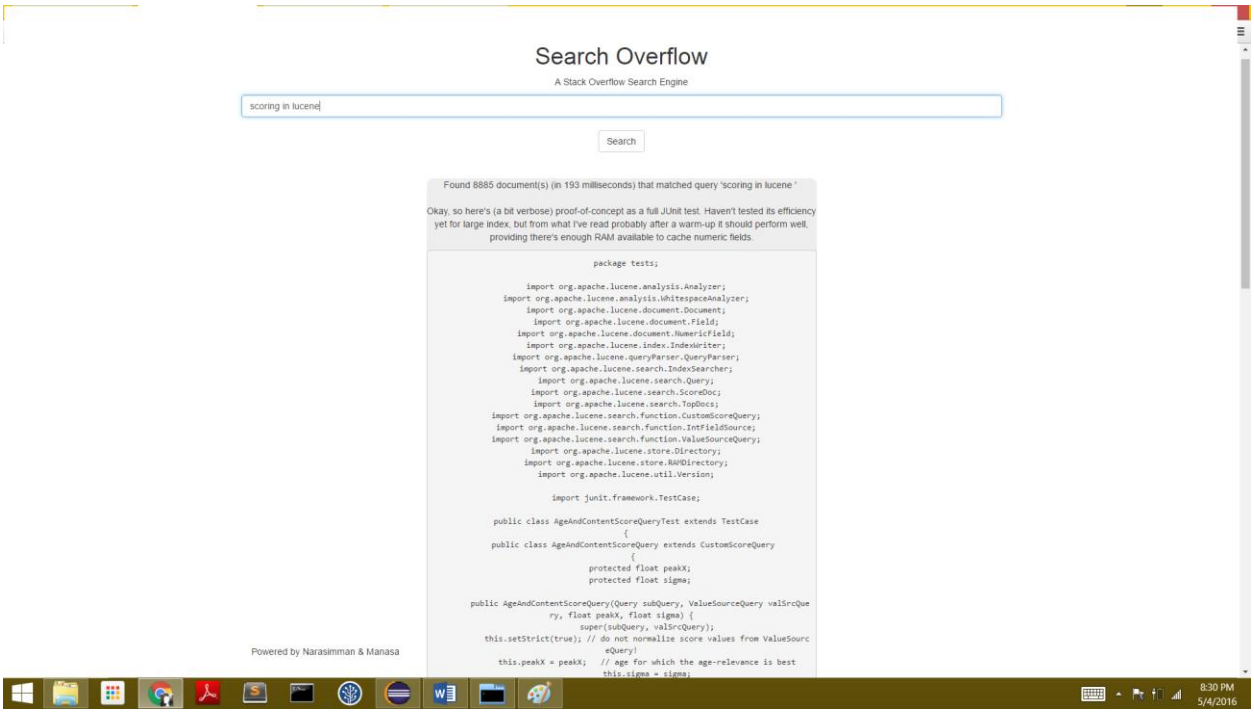
This query provides the perfect result
It shows the theory as well as the implementation



Another perfect result. The exact code required for the query



This is not the result expected
The query needs to be more explanatory to provide the desired result



Scoring in Lucene Query provides the perfect result

Next Steps

The customized scoring can be experimented on more. With experimentation, we could come up with the ideal weights that could be given to make a better match with the results that Google returns. We would like to incorporate more parameters in scoring the documents. For example we have the user ranking data, the favorite count and views. We are presently not taking them into consideration for the lack of time to experiment and obtain a good formula to use. To further provide a perfect answer, we need to explore with creating a custom query matching function to eliminate the differences in keywords mentioned previously. To make it more commercialized, the process to retrieve the answer needs to be optimized. The time taken requires to be lowered.

References

[1] Stack Exchange. Stack Exchange Data Dump, March 1, 2016
[2] Stack Exchange. Database schema documentation for the public data dump and SEDE, April 16, 2016
[3] Stack Exchange. Stack Exchange API V2.2, 2016