EECS 767

INFORMATION RETRIEVAL

InProgress Report

By: FiniteLoop Squad

Ron Andrews, Nidhi Midha, Blake Bryant

Table of Contents

[1. Introduction 3](#_Toc512167432)

[2. Programming Platform and Version Control Selection 4](#_Toc512167433)

[3. Data Structures 5](#_Toc512167434)

[3.1. Niche Web Crawler Data Structure 5](#_Toc512167435)

[3.2. Pre-Processing to Processing Interface Data Structures 5](#_Toc512167436)

[3.3. Processing to Query Processing Interfaces Data Structures 7](#_Toc512167437)

[3.4. HTML to Query Processing Interfaces Data Structures 10](#_Toc512167438)

[3.5. Query Processing to HMI Data Structures 10](#_Toc512167439)

[4. Ingest 11](#_Toc512167440)

[5. Preprocessing 11](#_Toc512167441)

[5.1. Tokenization 11](#_Toc512167442)

[5.2. Indexing 11](#_Toc512167443)

[6. Processing 12](#_Toc512167444)

[7. Query 13](#_Toc512167445)

[8. Appendix 14](#_Toc512167446)

[8.1. Manifest & Installation 14](#_Toc512167447)

[8.2. GitHub Contributions 15](#_Toc512167448)

# Introduction

The FiniteLoop Search Engine is a simple Information Retrieval System for a relatively static web page (document) repository, or corpus, using the Vector Space Model via an inverted index. In order to optimize the results and user experience, we have added techniques such as term proximity and Relevance Feedback for ranking of the results and the ability for a user to refine their search query based on the search results. The FiniteLoop Search Engine utilizes a multi-threaded niche web crawler to collect data from a specific domain and caches the documents locally for ingest and processing. In developing the search engine, we created a control group of documents, based on a quiz provided in class[[1]](#footnote-1), in order to test the ingest, processing, and query capability of the system. This report provides an explanation of the code structure, the data structures employed by the various search engine modules, and our results. Figure 1, FiniteLoop Search Engine Functional Flow, provides a conceptual functional (or process) flow of the search engine, post-retrieval of the documents via the niche web crawler.

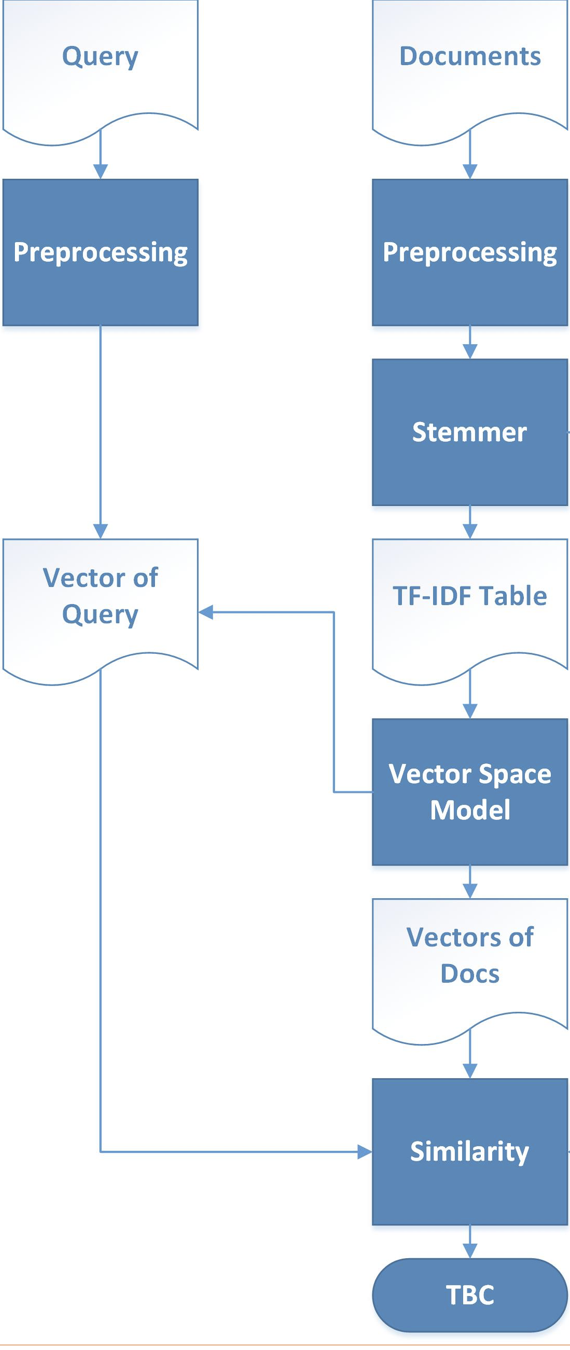


Figure , FiniteLoop Search Engine Functional Flow

# Programming Platform and Version Control Selection

In review of the various programming options available, we focused on those languages which were most capable, natively for this project. Specifically, looking at those languages which supported complex functions such as cosine similarity, web compatibility (Common Gateway Interface, CGI, or apache server module based), and of course, familiarity. After consideration of various options, such as *R*, *Perl*, *C++*, and *Python*, we selected *Python* as our language of choice.

With the current versions available for *Python*, we initially selected to go with version 3.6, being the latest available. As we worked through the various modules of our search engine (pre-processing, processing, query, and human machine interface (HMI), we ran into a few challenges. The Natural Language Toolkit (NLTK) that we selected to facilitate the stop list and lemmer was compatible with *Python* 3.5, not 3.6. Additionally, the Electrical Engineering and Computer Science (EECS) student web server currently provides access to *Python* 2.7 and 3.5. Our conclusion was to go forward with Python 2.7 as it was common to our individual environments as well as the web server. Additionally, we elected to use the EECS web server CGI capability for hosting our search engine.

For our collaboration environment, we set up a GitHub repository specifically for our **FiniteLoop** **Squad** to work and share. In the environment, we are able to coordinate our code development efforts as well as documentation.

# Data Structures

In order to pass the data structures between our modules, we are leveraging a *Python* module called *shelve*. This native module enables us to pass the raw data structures by way of a binary file stored on the file server. The following sub-sections provide the data structures passed between the modules. As a test we utilized a control group of data files, based on the class quiz covering VSM. The text data files are as follows:

test1.txt: Shipment of gold damaged in a fire

test2.txt: Delivery of silver arrived in a silver truck.

test3.txt: Shipment of gold arrived in a truck.

test4.txt: Truck arrive damaged.

# Niche Web Crawler Data Structure

The niche web crawler creates an index of the documents as it crawls and caches the web sites from the frontier. The crawler provides the following data structures to the Ingest module:

1. Download Manifest

The Download Manisfest database provides ingest a dictionary of the filenames that were crawled with a value of the URL that the file was retrieved from.

The generalized data structure look like the following, in *Python* terms:

‘’ = {

|  |  |
| --- | --- |
| DocName1: | DocURL1], |
| DocName2: | DocURL2], |
| …, |  |
| DocNamen: | DocURLn] |

}

The control group does not provide a download\_manifest, as these files are made locally available and not run through the crawler.

# Pre-Processing to Processing Interface Data Structures

Pre-Processing provides the following data structures to the Processing module:

1. Document Key Matrix
2. Term Incidence Matrix, with Frequency
3. Term Proximity Matrix
4. Title mapping
5. Number of documents

The Document Key provides the details for each document as a *Python* dictionary where the document name is the key and the details is a list of values. The details for each document include the document identifier, current location on the local filesystem (cache), and the URL [[2]](#footnote-2)of the document. The document identifier provides the sorting order of the documents and is also the index in the Term Incidence Matrix. This data structure is an *mx3* matrix, where *m* is the number of documents in the corpus.

The generalized data structure looks like the following, in *Python* terms:

doc\_key = {

|  |  |  |  |
| --- | --- | --- | --- |
| DocName1: | [DocID1, | DocLocation1, | DocURL1], |
| DocName2: | [DocID2, | DocLocation2, | DocURL2], |
| …, |  |  |  |
| DocNamen: | [DocIDn, | DocLocationn, | DocURLn] |

}

As a practical example, the contents of the **doc\_key** using the control group is:

doc\_key = {

|  |  |  |  |
| --- | --- | --- | --- |
| 'test1.txt': | [3, | '/EECS767/FiniteLoopSE/test1.txt', | 'no\_url'], |
| 'test3.txt': | [2, | '/EECS767/FiniteLoopSE/test3.txt', | 'no\_url'], |
| 'test4.txt': | [0, | '/EECS767/FiniteLoopSE/test4.txt', | 'no\_url'], |
| 'test2.txt': | [1, | '/EECS767/FiniteLoopSE/test2.txt', | 'no\_url'] |

}

The Term Incidence Matrix provides each term and its occurrence in the corpus as a *Python* dictionary[[3]](#footnote-3) where the term is the key and the document incidence with frequency is a list. Each document incidence list is aligned in order with the list provided in the doc\_key data structure. This data structure is an *nxm* matrix, where *n* is the number of terms and *m* is the number of documents in the corpus.

The generalized data structure looks like the following, in *Python* terms:

index = [

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| { Term1: | [tf1, | tf2, | …, | tfn] }, |
| { Term2: | [tf1, | tf2, | …, | tfn] }, |
| …, |  |  |  |  |
| { Termm: | [tf1, | tf2, | …, | tfn] } |

]

As a practical example, the contents of the **index** using the control group is:

index = {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 'truck': | [1, | 1, | 1, | 0], |
| 'arriv': | [1, | 1, | 1, | 0], |
| 'damag': | [1, | 0, | 0, | 1], |
| 'fire': | [0, | 0, | 0, | 1], |
| 'silver': | [0, | 2, | 0, | 0], |
| 'gold': | [0, | 0, | 1, | 1], |
| 'deliveri': | [0, | 1, | 0, | 0], |
| 'shipment': | [0, | 0, | 1, | 1] |

}

The Term Proximity Matrix provides a dictionary of each term, as the key, and a list of tuples as the value. The tuples identify the document and offset from the beginning of the document. Offsets are based on word distance from the beginning of the document after the tokenization and stop word parsing is complete. This data structure is an *nxp* matrix, where *n* is the number of terms and *p* is the number of non-unique terms in the corpus.

The generalized data structure looks like the following, in *Python* terms:

proximity = {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Term1: | [ [DocID, Prox], | [DocID, Prox], | … | [DocID, Prox] ], |
| Term2: | [ [DocID, Prox], | [DocID, Prox], | … | [DocID, Prox] ], |
| …, |  |  |  |  |
| Termm: | [ [DocID, Prox], | [DocID, Prox], | … | [DocID, Prox] ] |

}

As a practical example, the contents of the **proximity** using the control group is:

proximity = {

|  |  |  |  |
| --- | --- | --- | --- |
| 'truck': | [ (0, 0), | (1, 4), | (2, 3) ], |
| 'arriv': | [ (0, 1), | (1, 2), | (2, 2) ], |
| 'damag': | [ (0, 2), | (3, 2) ], |  |
| 'fire': | [ (3, 3) ], |  |  |
| 'silver': | [ (1, 1), | (1, 3) ], |  |
| 'gold': | [ (2, 1), | (3, 1) ], |  |
| 'deliveri': | [ (1, 0) ], |  |  |
| 'shipment': | [ (2, 0), | (3, 0) ] |  |

}

The title map provides a dictionary of document names and document titles, specifically for HTML pages by leveraging the contents of the title[[4]](#footnote-4). This information is metadata used to provide ‘summary’ text displayed in the results of the search.cgi script. This data structure is an *mx2* matrix, where *m* is the number of documents in the corpus.

The generalized data structure looks like the following, in *Python* terms:

title\_map = {

|  |  |
| --- | --- |
| DocName1: | DocTitle1, |
| DocName2: | DocTitle2, |
| …, |  |
| DocNamen: | DocTitlen, |

}

As the control group does not contain any html pages, a relative example is not provided and left to the reader to imagine.

The number of docs is provided to ensure that there is a check between ingest and processing that the number of documents expected is there number of documents processed. This variable is simply an integer stored in the database file:

num\_docs = x

As in the case of the control group, the database reflects:

num\_docs = 4

# Processing to Query Processing Interfaces Data Structures

Processing provides the following data structures to the Query Processing module:

1. Document Key Matrix
2. Normalized Vector Space Model (VSM) – This file is stored in an artifacts database (processingArtifacts.db) for reference and troubleshooting)
3. Term Dictionary (based on VSM)
4. Term Proximity Dictionary
5. Term IDF Dictionary

The Document Key Matrix is forwarded, unaltered from what was received from the Pre-Processing module, see Section 3.1, Pre-Processing to Processing Interface Data Structures.

The Normalized VSM provides an alphabetically sorted list of vectors (lists). Each vector is in order as identified by the Term Index Look-Up Dictionary and each vector is in order of the Document Key Matrix. The vectors provide the normalized Term Frequency – Inverted Data Frequency (TF-IDF) weight of the term for each document.

The generalized data structure looks like the following, in *Python* terms:

docVector = [

|  |  |  |  |
| --- | --- | --- | --- |
| [WT1,D1, | WT1,D2, | …, | WT1,Dn], |
| [WT2,D1, | WT2,D2, | …, | WT2,Dn], |
| … |  |  |  |
| [WTm,D1, | WTm,D2, | …, | WTm,Dn] |

]

As a practical example, the contents of the **docVector** using the control group is:

docVector = [

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [0.356, | 0.858, | 0.0, | 0.0, | 0.0, | 0.0, | 0.0, | 0.356], |
| [0.118, | 0.0, | 0.57, | 0.0, | 0.0, | 0.0, | 0.805, | 0.118], |
| [0.27, | 0.0, | 0.0, | 0.0, | 0.65, | 0.65, | 0.0, | 0.27], |
| [0.0, | 0.378, | 0.0, | 0.755, | 0.378, | 0.378, | 0.0, | 0.0] |

]

The docVector, is used to create a look up table for the query module (evoked by search.cgi) to quickly locate terms from the query and perform the cosine similarity process. This resulting data structure is a dictionary of terms such that each term contains the document arrays of normalized weights for that term. This is a reduced set, as there are many 0 weights in the VSM (very sparse) – in order to reduce overhead and look up times, we created a term Dictionary to house tuples of weights and document IDs.

The generalized data structure looks like the following, in *Python* terms:

termDict = {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Term1: [ | [DocID, Weight], | [DocID, Weight], | … | [DocID, Weight] ], |
| Term2: [ | [DocID, Weight], | [DocID, Weight], | … | [DocID, Weight] ], |
| …, |  |  |  |  |
| Termm: [ | [DocID, Weight], | [DocID, Weight], | … | [DocID, Weight] ] |

}

As a practical example, the contents of the **termDict** using the control group is:

termDict = {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 'silver': [ | [1, 0.805] ], |  |  |  |
| 'truck': [ | [0, 0.356], | [1, 0.118], | [2, 0.27] ], |  |
| 'arriv': [ | [0, 0.356], | [1, 0.118], | [2, 0.27] ], |  |
| 'shipment': [ | [2, 0.65], | [3, 0.378] ], |  |  |
| 'damag': [ | [0, 0.858], | [3, 0.378] ], |  |  |
| 'deliveri': [ | [1, 0.57] ], |  |  |  |
| 'gold': [ | [2, 0.65], | [3, 0.378] ], |  |  |
| 'fire': [ | [3, 0.755] ] |  |  |  |

}

The Term Proximity Dictionary provides a listing similar to the Term Dictionary where instead of the weights, it provides a list of the proximities. Specifically, this data structure is constructed for ‘easy’ look up by using the term to find the document ID, to get the proximities.

The generalized data structure looks like the following, in *Python* terms:

proxDict = {

|  |  |  |  |
| --- | --- | --- | --- |
| Term1: { | { DocID: [Prox’s] }, | … | { DocID: [Prox’s] } }, |
| Term2: [ | { DocID: [Prox’s] }, | … | { DocID: [Prox’s] } }, |
| …, |  |  |  |
| Termm: [ | { DocID: [Prox’s] }, | … | { DocID: [Prox’s] } } |

}

As a practical example, the contents of the **proxDict** using the control group is:

proxDict = {

|  |  |  |  |
| --- | --- | --- | --- |
| 'silver': { | 1: [1, 3] }, |  |  |
| 'truck': { | 0: [0], | 1: [4], | 2: [3] }, |
| 'arriv': { | 0: [1], | 1: [2], | 2: [2] }, |
| 'shipment': { | 2: [0], | 3: [0] }, |  |
| 'damag': { | 0: [2], | 3: [2] }, |  |
| 'deliveri': { | 1: [0] }, |  |  |
| 'gold': { | 2: [1], | 3: [1] }, |  |
| 'fire': { | 3: [3] } |  |  |

}

The Term IDF Dictionary provides a dictionary of each term, as the key, and the term IDF weight as well as the index of the term in the VSM (stored in the processingArtifacts). This data structure is used by query to quickly assign IDF values to the query terms.

The generalized data structure looks like the following, in *Python* terms:

termIDF = {

|  |  |
| --- | --- |
| Term1: | [ IDF, Index ], |
| Term2: | [ IDF, Index ], |
| …, |  |
| Termm: | [ IDF, Index ], |

}

As a practical example, the contents of the **termIDF** using the control group is:

termIDF = {

|  |  |
| --- | --- |
| 'silver': | [0.602, 6], |
| 'truck': | [0.125, 7], |
| 'arriv': | [0.125, 0], |
| 'shipment': | [0.301, 5], |
| 'damag': | [0.301, 1], |
| 'deliveri': | [0.602, 2], |
| 'gold': | [0.301, 4], |
| 'fire': | [0.602, 3] |

}

# HTML to Query Processing Interfaces Data Structures

The HTML module provides the following data structure to the Query Processing module:

1. HTML Text

The HTML Text dictionary provides a dictionary of each document, as the key to dictionaries of terms and summary text. This was created due to the large processing time for accessing the information. Each summarized file in the corpus results in a separate summary file. The summary file is the name of the document (from doc\_key) with a ‘.db’ suffix.

The generalized data structure looks like the following, in *Python* terms

htmlText = {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DocName1: | { Term1: Text, | Term2: Text, | …, | Termn: Text } |
| DocName2: | { Term1: Text, | Term2: Text, | …, | Termn: Text } |
| …, |  |  |  |  |
| DocNamen: | { Term1: Text, | Term2: Text, | …, | Termn: Text } |

}

As a practical example, the contents of the **htmlText** using the control group (for test1.txt) is:

htmlText = {

|  |  |  |
| --- | --- | --- |
| ‘test1.txt’: | { ‘fire’: gold damaged in a fire’, | ‘damaged’: ‘Shipment of gold damaged in a fire’, |
|  | 'gold': 'Shipment of gold damaged in a fire', | 'shipment': 'Shipment of gold damaged in' } |

}

# Query Processing to HMI Data Structures

The data passed to the HMI is done so directly with the CGI script importing the Query module directly for dynamic processing and results. The Query module provides the following data and data structures to the HMI for display to the user:

* Total Number of Results found
* Time taken to process query and return results
* List of results, ordered by relevance

The list of results is provided to the HMI as a list of document entries. Each entry contains a list of parameters to display to the user.

The generalized data structure looks like the following, in *Python* terms:

results = [

[ DocName1, DocLocation1, Rank1, Summary1],

[ DocName2, DocLocation2, Rank2, Summary2],

…,

[ DocNamen, DocLocationn, Rankn, Summaryn]

]

# Ingest

Ingestion is performed by functions contained within the ingest.py file which operate on files stored within a local directory. The path to the directory is currently configured with a static path hard coded within the ingestion function. A function was created to allow for user entry of a different path location, however this functionality is currently deemed unnecessary and has been disabled via inline comments.

The native python library “urllib2” is used to read files and provide compatibility for processing various document formats including html. Each document within the specified directory is loaded into memory as a stream of raw characters and stored in an array, called “data,” with each document representing a single index within the array.

A dictionary called “doc\_key” is created to store the filename, document ID and file path for each document ingested. The document filename is used as the key for the dictionary while the value for each key is represented as an array containing the document ID and file path.

# Preprocessing

# Tokenization

Each document, stored as an index within the data array created during ingestion, is processed by the function “func\_tokenize” within the ingest.py file. The Python Natural Language Tool Kit (NLTK) (<https://www.nltk.org/>) is used to create stop word lists and a stemming function within func\_tokenize. This requires the installation of NLTK as well as downloading the stop word list prior to running the ingestion program. NLTK version 3.2.5 (the latest version as of this report) requires either Python version 2.7 or 3.5. Python 3.5 is not the most current release of Python 3 and may require some systems to install an earlier version of Python.

Preprocessing within func\_tokenize consists of 4 steps executed on each index of the data array:

1. HTML tags are removed from the data stream via regular expression pattern matching.
2. The data stream is converted to lower case, punctuation is removed, and the stream is split into tokens via the Python string.split() method.
3. Stop words contained within the NLTK stop word list are removed from the list of tokens.
4. Each token within the data stream is processed by the Porter stemmer provided by the NLTK library. There is a known issue when processing

Unicode characters with the NLTK Porter stemmer. Because of this, words containing Unicode characters are currently dropped in the stemming phase.

# Indexing

The number of documents processed during ingestion is used to determine the dimensionality of arrays within the terms dictionary. Each data stream, corresponding with a separate document, is parsed for unique terms. If a new term is discovered in a document, the term is added as a key to the dictionary called “terms” with an initial value of an array with multiple indices corresponding with the number of documents processed. The index corresponding to the document, wherein the term was observed, is then incremented by 1. Subsequent observations of the term within the same document, or future documents, results in incrementing the value stored in the term array at the index corresponding with the id of the document being parsed. This results in an array reflecting the term frequency for each term observed within each document.

Concurrently, an additional dictionary called “proximity” is created to record positional data pertaining to each occurrence of a term within the documents parsed. Similar to the “terms” dictionary, unique terms are used as key values in the dictionary. Values within the “proximity” dictionary are represented as an array containing tuples consisting of the document ID and the position the term appeared within the document. A new tuple representing the document id and term position is appended to the term array each time the term is observed within the data stream. This data structure is described in further detail within the “Data Structures” section of this report.

Finally, data is exported to a file called “ingestOutput.db” using the Python shelve library. The “terms” dictionary is exported as “index”, the “doc\_key” dictionary is exported as “doc\_key” and the “proximity” dictionary is exported as “proximity” within the output file.

# Processing

The processing module reads in the output file from the ingest function in order to acquire the pre-processing data structures. It then generates the TF-IDF, normalizes the vectors and stores the VSM for the query module. Additionally, processing generates a proximity matrix, similar to the VSM data structure and also stores it, along with the doc\_key structure into the output shelve file for use by the query function.

To accomplish this, the index data structure (the term incidence matrix) is sorted and then walked through to evaluate for the Document Frequencies of each term in order to generate the VSM. It does so by first calculating the non-zero indices in the document arrays along with calculating the idf (log n/df), where n is the length of the array (# of documents). With this data, the non-normalized weights are calculated, |Wi| = sqrt(sum(idf2)), and then added to the VSM (docVector). Finally, the module goes through the VSM and normalizes the document vectors and stores the data structure in the shelve output file for the query function.

The proximity file provided by ingest contains a dictionary of tuples where each tuple indicates the term and offset.

|  |  |
| --- | --- |
| n - # of terms, m = # of documents, p = # of term occurrences | |
| ***Prepare index, document key, and proximity key from ingest*** | **O(n log n + 2n + m)**  **=~ O(n log n)** |
| *sortedTerms = sorted(list of dictionary keys)* | O(n log n)[[5]](#footnote-5) |
| *sortedTermIndex = array of term hashes based on sortedTerms* | O(n) |
|  |  |
| *sortedDocs = array of document hashes based on doc index id* | O(m) |
|  |  |
| *sortedProximity = array of term prox hashes based on sortedTerms* | O(n) |
|  |  |
| ***Generate TF-IDF*** | **O(n x 2m)** |
| *For each word in termIndex* | O(n) |
| *DF = sum of all non-zero indices in the doc array* | O(m) |
| *Calculate IDF* |  |
| *Calculate weights for word in each document* | O(m) |
|  |  |
| ***Normalize Vectors*** | **O(n x m)** |
| *Create doc length (unit) for each document* | O(m) |
| *For each word in termIndex* | O(n) |
| *Normalize each weight for each document* | O(m) |
|  |  |
| ***Process proximity*** | **O(m x p)** |
| *For each document* | O(m) |
| *For each term occurrence (tuple)* | O(p) |
| *Process tuple and append to term index array* | O(1) |
|  |  |
| Totals  O(n log n) + O(n x 2m) + O(n x m) + O(m x p)  O(n log n) + O(n x m) + O(m x p) | O(nm) or O(mp)  Where p is # of total words in corpus |

# Query

The dictionaries “doc\_vector” and “doc\_key” are passed to query module using a file processingOutput.db which is Python shelve library. “doc\_vector” dictionary provides the alphabetically sorted list of vectors (lists). Each vector is in order as identified by the Term Index Look-Up Dictionary and each vector is in order of the Document Key Matrix. The vectors provide the normalized Term Frequency – Inverted Data Frequency (TF-IDF) weight of the term for each document. Weights in this dictionary are compared with another vector “query\_vector”.

Search query inputted by user is first preprocessed by removing the stop words and processing by the Porter Stemmer provided by NLTK library. Once processed, query tokens are stored in the query vector, whose weights are compared with the document vectors to calculate the cosine similarity. The cosine similarities are calculated using the method similarity() which sorts the similarity, and ranks the results in descending order on the basis of doc\_key associated with each document vector. The ranks are stored in queryOutput.db, which would be passed to CGI to display the results.

Current work which is going on includes getting the proxVector from processing module, and re-rank the top 10 documents obtained from cosine similarity, on the basis of proximities.

# Search

The search Common Gateway Interface (CGI) pulls together the user interaction with the results of crawling, ingesting, processing, and active query of the corpus….

# Appendix

The Makefile provides the necessary functions to ensure that the environment will work correctly, provided the user is in a Linux-based environment running python 3.5. The FiniteLoopSE search engine was developed to be executed in the EECS environments and run in a user’s student web page area via the *people.eecs.ku.edu* web hosting. Due to the limitations on cycle servers being able to access external pages, other than a few git sites, we ran the niche web crawler on a student workstation within the ITTC domain and transfer the resulting cached documents to the EECS domain for ingest and processing.

1. EECS 767, Information Retrieval, Spring 2018 [↑](#footnote-ref-1)
2. For instances where a URL isn’t available, such as for local test files, a default value of ‘*no url*’ is used [↑](#footnote-ref-2)
3. Note, *python* dictionaries are not sorted, the order is not guaranteed [↑](#footnote-ref-3)
4. Extracts text between <title>some text</title> [↑](#footnote-ref-4)
5. Python sort function is a hybrid of merge sort, average performance O(n log n) [2] [↑](#footnote-ref-5)