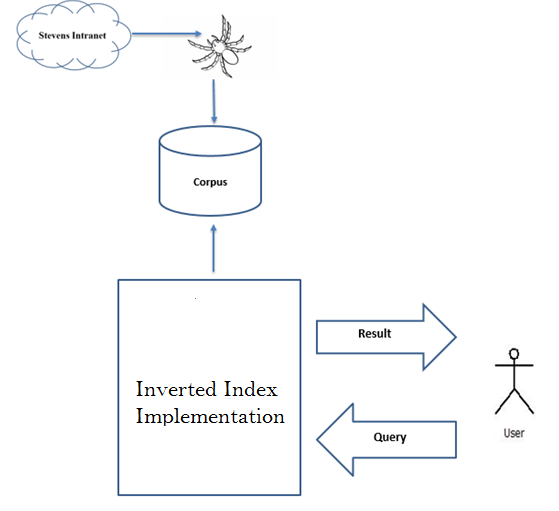
**Search Engine Understanding and Implementation:**

I have implemented this search engine using python which asks user to enter **search query** and based on the query parameter I am returning the relevant. I am returning the pages in descending order of occurrence of the query in the page. To have multiple documents for quering I have downloaded CS syllabus links file present in Stevens Intranet.

Use all the words in the pages as an index term excluding stop words such as articles, prepositions, and pronouns. The core information stored by a search engine is a dictionary, called an inverted index or inverted file, storing key-value pairs (w,L), **where w is a word and L is a collection of references to pages containing word w along with the count**. The keys (words) in this dictionary are called index terms and should be a set of vocabulary entries and proper nouns as large as possible.

**Diagram for Implementation**



The implementation of my search engine contains an inverted index with a data structure consisting of the following:

The overall system is distributed into three main modules which are:

• Stevens Intranet and Web Crawler

• Corpus

• Search Engine

Web Crawler crawls the syllabus links on the intranet and convert the syllabus xml files into text files. Corpus is dictionary of all names of text files in which we are searching matching query.

Search Engine is a Dictionary containing all the **unique words** **as keys present in all documents which has value,** which is also a dictionary containing key as **document number** and number of occurrences of the word in that document.

**Package specification**

* NLTK package for removing punctuation and Stop words
* CElementTree creating text files.
* PrettyTable to display the data
* Math and os function for data operations.
* BytesIo for stream handling.
* Pycurl to execute curl command to create request and fetch the data.

**Sequence of Execution:**

* Copy the link into txt file and save the same, right now the file name is **syllabus-links.txt** containing links of syllabus of all cs courses.
* Run the code with file name **searchEngine.py**, from our code we are fetching data from each link and saving it in **new file**. File names are **CS501 to CS609**.
* After creating these file our code will ask to enter search query and our code will try to find out the document which has maximum occurrence of the query word.
* **We are retrieving the documents in the decreasing order of cosine similarity which means higher the cosine value more relevant the document is.**
* To exit from the code **just give space and enter** you will be out of the code.

**Modules:**

The entire project is divided into two modules, Corpus data and Inverted files implementation.

Vector Space module is composed below functions. Each function and its details are as outlined below:

1. **initialize\_terms\_and\_postings()**

This function reads in each document in document\_filenames, splits it into a list of terms (i.e., tokenizes it), adds new terms to the global dictionary, and adds the document to the posting list for each term, with value equal to the frequency of the term in the document.

1. **initialize\_document\_frequencies()**

In this function, for each term in the dictionary, it counts the number of documents it appears in, and store the value in document\_frequncy[term].

1. **initialize\_lengths()**

This function Computes the length for each document.

1. **tokenize(document):**

Returns a list whose elements are the separate terms in document. Something of a hack, but for the simple documents we're using, it's okay. Note that we case-fold when we tokenize, i.e. we lowercase everything.

1. **initialize\_document\_frequencies():**

For each term in the dictionary, count the number of documents it appears in, and store the value in document\_frequncy[term].

1. **imp(term,id):**

Returns the importance of term in document id. If the term isn't in the document, then return 0.

1. **inverse\_document\_frequency(term):**

Returns the inverse document frequency of term. Note that if term isn't in the dictionary then it returns 0, by convention.

1. **do\_search():**

Asks the user what they would like to search for, and returns a list of relevant documents, in decreasing order of cosine similarity. This function makes calls to below functions for generating the desired output.

* Tokenize()
* Intersection()
* similarity(query,id)

1. **Union(sets):**

Returns the union of all sets in the list sets. Requires that the list sets contains at least one element, otherwise it raises an error

1. **similarity(query,id):**

Returns the cosine similarity between query and document id. Note that we don't bother dividing by the length of the query vector, since this doesn't make any difference to the ordering of search results.

Apart from these functions, we have also defined some global variable.

1. **document\_filenames**

We use a corpus of four documents. Each document has an id, and these are the keys in the following dict. The values are the corresponding filenames.

Example:

document\_filenames = {0 : "documents/CS501.rdf ",

1 : "documents/ CS502.rdf",

2 : "documents/ CS548.rdf",

4: "documents/CS526.rdf",

5: "documents/CS513.rdf"}

1. **Dictionary**:

Dictionary: a set to contain all terms (i.e., words) in the document corpus.

1. **Postings**:

a defaultdict whose keys are terms, and whose corresponding values are the so-called "postings list" for that term, i.e., the list of documents the term appears in.

The way we implement the postings list is actually not as a Python list. Rather, it's as a dict whose keys are the document ids of documents that the term appears in, with corresponding values equal to the frequency with which the term occurs in the document. As a result, postings[term] is the postings list for term, and postings[term][id] is the frequency with which term appears in document id.

1. **document\_frequency**

a defaultdict whose keys are terms, with corresponding values equal to the number of documents which contain the key, i.e., the document frequency.

1. **length**: a defaultdict whose keys are document ids, with values equal to the Euclidean length of the corresponding document vector.