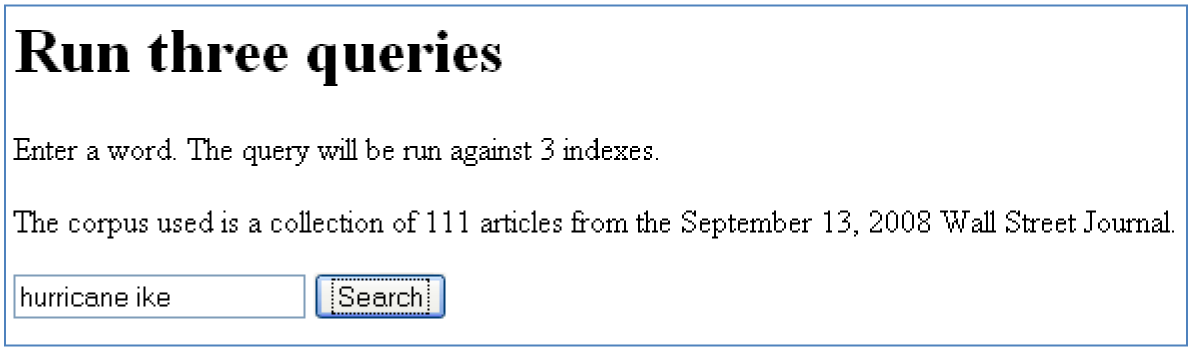
Information Retrieval Database with WordNet Word Sense Disambiguation

Caden Howell
Information Retrieval CSC 575 Dr. Joe Phillips
March 18, 2009

Using the System

The system is accessible on the World Wide Web at http://www.cadenhowell.com/ppp/public/wnindex/sendquery.html (Wall Street Journal corpus) and http://www.cadenhowell.com/ppp/public/wnindex/wikiquery.html (Partial Wikipedia Featured Articles corpus)

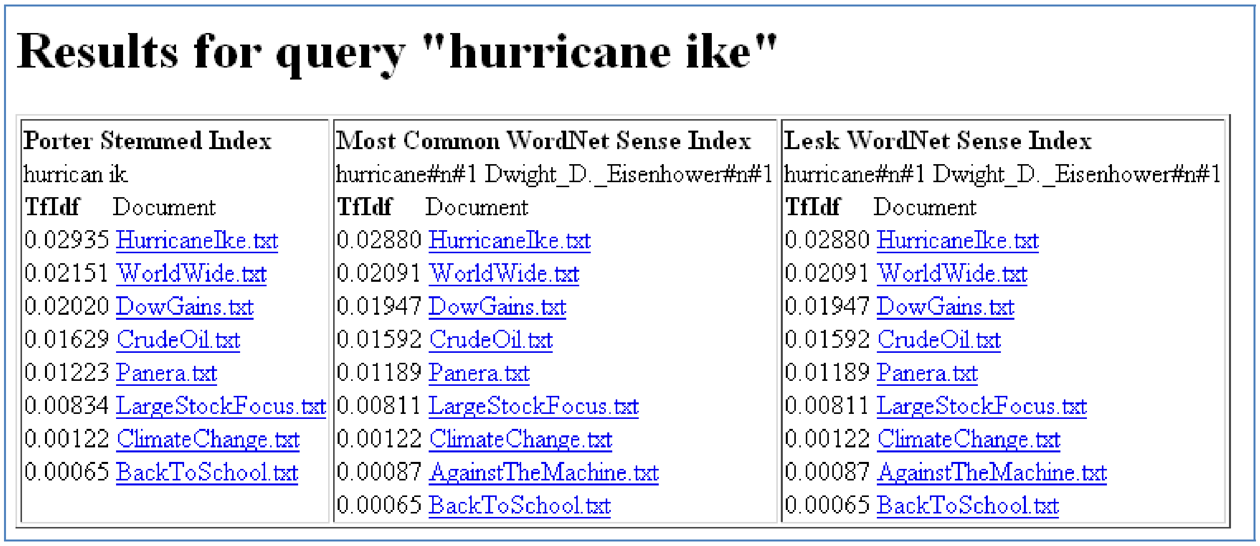
The user enters the query using a simple text box. Every query is treated as an OR query. That is, all of the terms are treated as if they were joined by Boolean OR.


Run three queries

Enter a word. The query will be run against 3 indexes.

The corpus used is a collection of 111 articles from the September 13, 2008 Wall Street Journal.

The following screen shows the result of running the query against three indexes.


Results for query "hurricane ike"

Porter Stemmed Index		Most Common WordNet Sense Index		Lesk WordNet Sense Index	
hurrican ik		hurricane#n#1 Dwight_D._Eisenhower#n#1		hurricane#n#1 Dwight_D._Eisenhower#n#1	
TfIdf	Document	**TfIdf**	Document	**TfIdf**	Document
0.02935	HurricaneIke.txt	0.02880	HurricaneIke.txt	0.02880	HurricaneIke.txt
0.02151	WorldWide.txt	0.02091	WorldWide.txt	0.02091	WorldWide.txt
0.02020	DowGains.txt	0.01947	DowGains.txt	0.01947	DowGains.txt
0.01629	CrudeOil.txt	0.01592	CrudeOil.txt	0.01592	CrudeOil.txt
0.01223	Panera.txt	0.01189	Panera.txt	0.01189	Panera.txt
0.00834	LargeStockFocus.txt	0.00811	LargeStockFocus.txt	0.00811	LargeStockFocus.txt
0.00122	ClimateChange.txt	0.00122	ClimateChange.txt	0.00122	ClimateChange.txt
0.00065	BackToSchool.txt	0.00087	AgainstTheMachine.txt	0.00087	AgainstTheMachine.txt
		0.00065	BackToSchool.txt	0.00065	BackToSchool.txt

The first index is a simple implementation of an information retrieval system using the Porter Stemming algorithm and TfIdf for document ranking. The second index is built assuming that the most commonly used WordNet sense of the term is intended by the query terms and index terms. The final index determines the word senses of the query terms using the Lesk algorithm, which uses the words in the neighborhood of a word to determine the appropriate word sense for the word.

All results found are returned. The results are not limited to a predetermined number of highly ranked documents.

System Architecture

The inverted index at the base of the system is emulated by a MySQL database. There are four main tables in the database: Document, Term, TermDocument, and TrialIndex. TrialIndex contains one row for each index in the database. Each corpus has an index for each indexing algorithm that was used. For example, the Wall Street Journal article corpus has three entries in the TrialIndex database, for the control algorithm group, the WordNet sense 1 algorithm group, and the WordNet Lesk algorithm group. The Document table contains one row for each document in a corpus. Document rows can be shared by multiple indices. The Term table contains one row for each term in an index. The term is stored in its stemmed form. The TermDocument table records the relationship between each document and the terms in that document. Note that "TermCount" stores the term frequency in a document, but the length of the document vector is stored in the DocLength table. This is because the length of a document vector can vary for different indices.

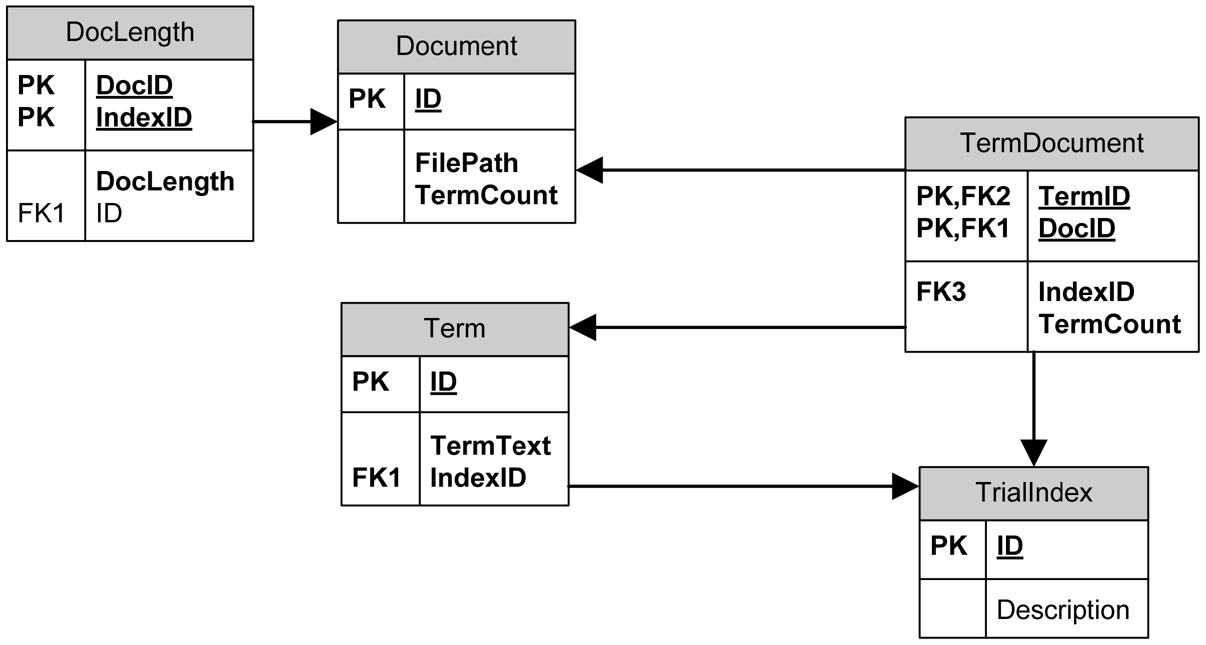


Figure 1 Structure of the MySQL Database

Rather than a linked list of postings, the terms and IDs of the documents where they appear are stored in tables in the database. To get a list of postings for a term, the term's row in the Term table must be joined with the term-document relationship rows in the TermDocument table.

In order to query the index, the user enters a query into a web form and submits the query to the web server. The query is entered as plain text, and will be treated as if the terms are joined by Boolean OR's. A Perl script and related libraries then query the database and return the results to the user. These steps will be addressed in more detail in the section "Querying Data."

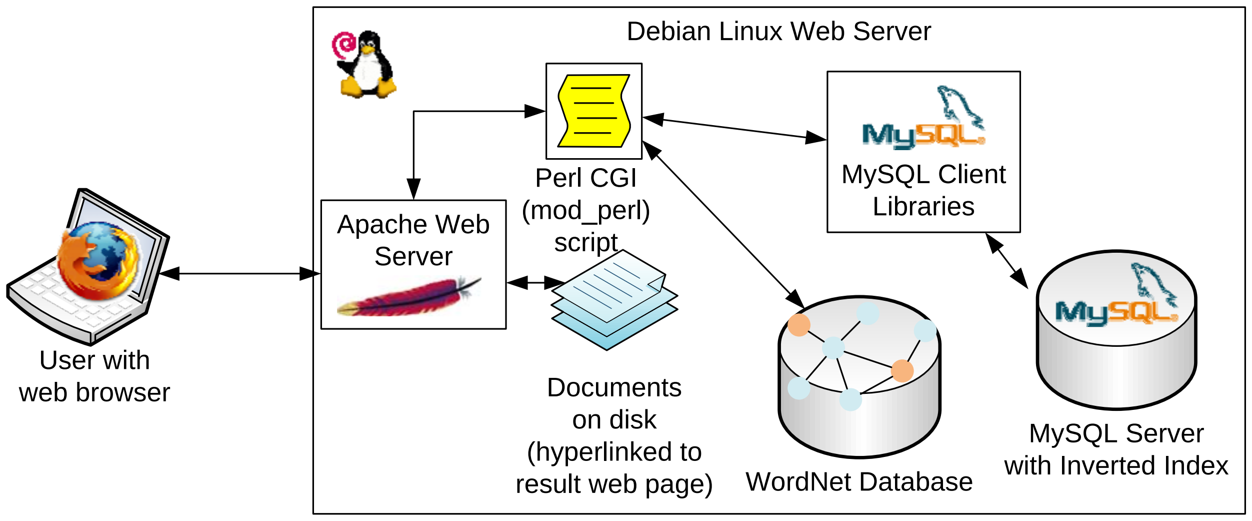

Debian Linux Web Server
Perl CGI (mod_perl) script
MySQL Client Libraries
Apache Web Server
User with web browser
Documents on disk (hyperlinked to result web page)
WordNet Database
MySQL Server with Inverted Index

Figure 2 Query application architecture

Building the Information Retrieval System

There were several stages in building the information retrieval system. First, corpus data was selected and harvested. Next, the data was converted to plain text. The data was then indexed and queried.

Harvesting Data

Three corpuses were selected for this project. Only the smallest corpus is completely indexed at this time. The three corpuses are:

1. The collection of over 2000 Wikipedia Featured articles. < http://en.wikipedia.org/wiki/Featured_Article> These are Wikipedia articles that are recognized to have good quality. I felt that they had a good breadth of subject matter coverage as well.
2. A collection of 111 stories from the Wall Street Journal on September 13, 2008.
3. The HTML version of our textbook. < http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html >

The collection of Wall Street Journal stories was small enough that I downloaded the individual articles manually from a database in the DePaul library. The other two collections were downloaded with a Perl script. This script used the LWP library, which acts like a web browser to pull information from the web so that it can be consumed by the Perl script.

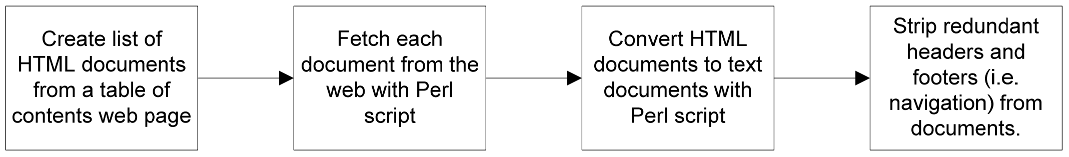

Create list of HTML documents from a table of contents web page
Fetch each document from the web with Perl script
Convert HTML documents to text documents with Perl script
Strip redundant headers and footers (i.e. navigation) from documents.

Figure 3 Corpus harvesting procedure

After collecting the articles, the HTML documents had to be converted to text. The simplest and most straightforward way to do this was to run the documents through the lynx web browser and save the text-formatted results. One of the scripts used is in Figure 3, below. The last part of the script trims away the last bit of the article, which was boilerplate text used in all Wikipedia articles.

```
#!/usr/bin/perl

@files = <fetched/*>;
foreach $file (@files) {
    $file =~ /fetched\/(.*)$/;
    $shortfile = $1;
    print "lynx -force_html -dump $file > ascii/$shortfile.ascii\n";
    `lynx -force_html -dump $file > ascii/$shortfile.ascii`;
    open(INFILE, "<ascii/$shortfile.ascii");
    open(OUTFILE, ">ascii/$shortfile.trim");
    while(<INFILE>) {
        if ($_ =~ /^Views/) { last; }
        print OUTFILE $_;
    }
    close(INFILE);
    close(OUTFILE);
}
```

Figure 4 Script to convert from HTML to text

Indexing Data

Before I was able to use WordNet as a tool for guessing at word senses and conflating words with synonymous meanings, I had to install the Wordnet database and tools on the indexing server. The indexing system required:

- Debian Linux (Etch)
- Wordnet 3.0 [3]
- MySQL 5.0 Server
- Perl
- Perl libraries:

  - Log::Dispatch
  - DBI::MySQL
  - WordNet::QueryData [4]
  - WordNet::Tools [5]
  - WordNet::SenseRelate::AllWords [6]
  - CustomSearch::Stemmer (essentially the Porter stemming code provided in class)

Once these were installed, most of the work was initiated and controlled through several Perl scripts and custom modules I created:

- CustomSearch::CustomLog, to log and debug
- CustomSearch::Indexer, which contains the indexing subroutines
- gotime.pl, which adds documents to the database and depends on CustomSearch::CustomLog and CustomSearch::Indexer
- addtime.pl, which adds additional term indices for existing documents to the database and depends on CustomSearch::CustomLog and CustomSearch::Indexer

The steps followed in the indexing algorithm are illustrated below in Figure 5. First the documents were normalized and stopwords were removed. Then, the terms in each document were stemmed using the Porter algorithm. The control index terms were then saved to the database. The WordNet stemmed documents went through an additional step, where the senses of the words were determined, and synonyms were converged to a single synonym “stem” which represented all words in that synset.

The final steps in the indexing which depended on a fully loaded corpus were done by running SQL scripts after the documents loaded. This is how the IDF was added to the Term entries and the document vector length was added to the Document entries.

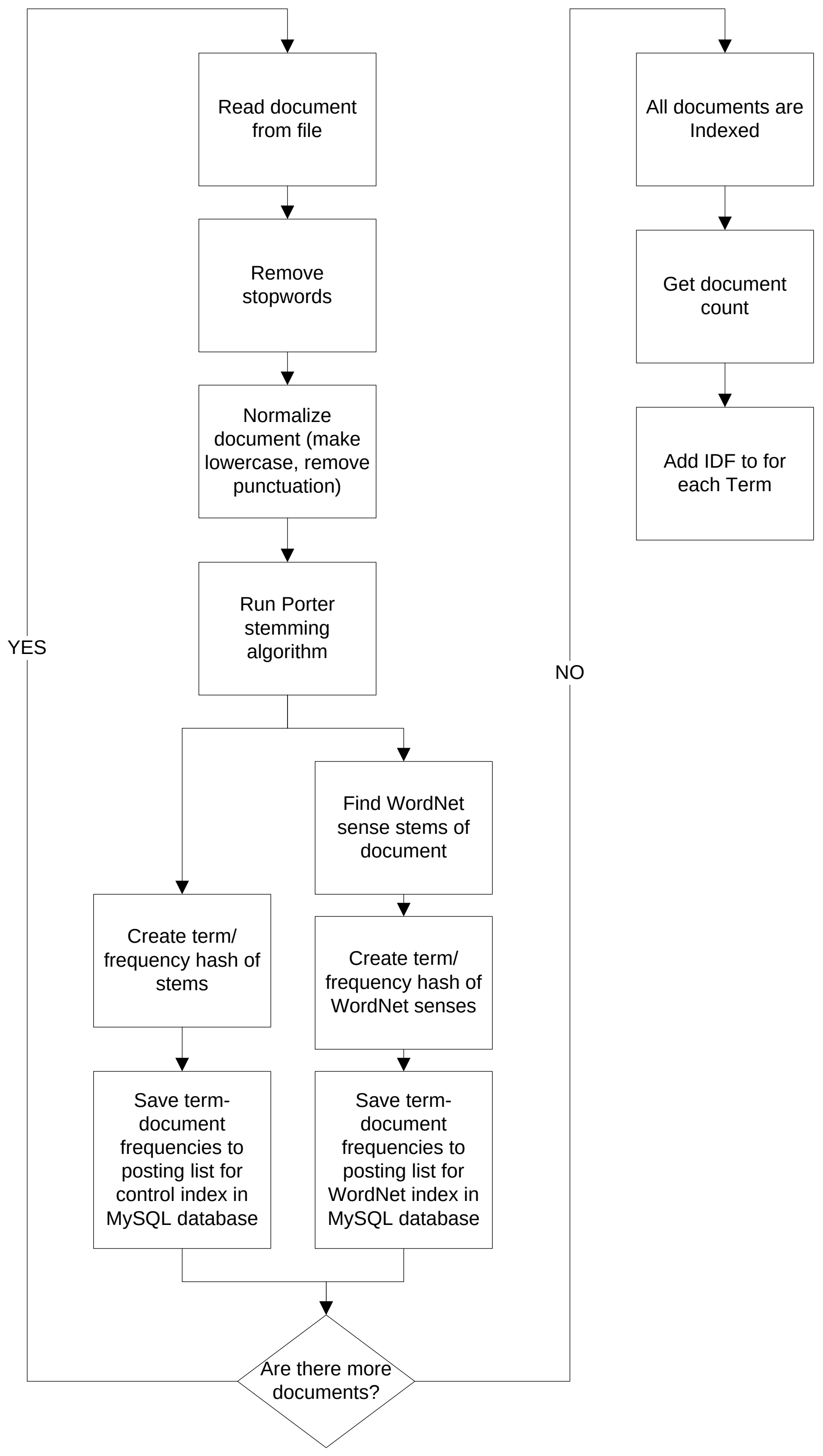

Read document from file
Remove stopwords
Normalize document (make lowercase, remove punctuation)
Run Porter stemming algorithm
Find WordNet sense stems of document
Create term/ frequency hash of stems
Create term/ frequency hash of WordNet senses
Save term-document frequencies to posting list for control index in MySQL database
Save term-document frequencies to posting list for WordNet index in MySQL database
Are there more documents?
YES
NO
All documents are Indexed
Get document count
Add IDF to for each Term

Figure 5 Steps in creating the indices

Before choosing the word sense disambiguation algorithm to be used in the indices, I ran a simple benchmark of several disambiguation algorithms using the Perl Benchmark module. Descriptions of each algorithm can be found in the CPAN documentation for the WordNet-Similarity Module.[7] Lesk was not the best performer, but I did notice that Lesk consistently disambiguated more words than the other algorithms. Other algorithms would return a majority of “word sense not determined” results, and Lesk at least gave a best guess.

Table 1 Results of diambiguation benchmark

```
              s/iter    hso vector_pairs vector  lesk  wup  lch  lin  res  jcn path random
hso              351     --          -86%   -97%  -98% -99% -99% -99% -99% -99% -99%  -100%
vector_pairs    48.0   630%            --   -80%  -82% -92% -93% -93% -94% -94% -96%   -98%
vector          9.52  3587%          405%     --   -8% -60% -64% -66% -68% -70% -81%   -92%
lesk            8.75  3907%          449%     9%    -- -56% -61% -63% -66% -67% -79%   -92%
wup             3.83  9049%         1153%   148%  128%   -- -11% -16% -21% -26% -53%   -81%
lch             3.43 10144%         1303%   178%  156%  12%   --  -6% -12% -17% -48%   -79%
lin             3.21 10830%         1397%   196%  173%  19%   7%   --  -6% -11% -44%   -77%
res             3.02 11537%         1493%   216%  190%  27%  14%   6%   --  -5% -40%   -76%
jcn             2.85 12211%         1586%   234%  207%  35%  20%  13%   6%   -- -37%   -74%
path            1.79 19446%         2576%   430%  388% 114%  91%  79%  68%  59%   --   -59%
random         0.735 47635%         6436%  1195% 1091% 422% 366% 337% 310% 288% 144%     --


              s/iter   lesk fixed         lesk       random lesk sense 1
lesk fixed      7.65           --          -2%          -92%         -94%
lesk            7.48           2%           --          -91%         -94%
random         0.645        1086%        1060%            --         -25%
lesk sense 1  0.486        1474%        1439%           33%           --
```

Indexing the porter-stemmed documents took only a few seconds per document. However, I soon found that indexing using the Lesk similarity algorithm took about 30 minutes per document. With the Wikipedia Featured Article corpus, this would take 1000 hours. At this point, I decided to institute a simple sort of parallel processing, by installing all the necessary software on one additional Debian machine and two additional Windows machines running vmware and Debian virtual machines. The resulting architecture is illustrated in Figure 6 below.   Even with the additional processing, I was only able to complete 800 documents by the assignment deadline.

In addition to the Lesk index, I decided to create the WordNet index using “sense1”.   The “sense1” algortihm not use the senses of neighboring words to determine word sense at all. Instead, it always returns the most popular sense of a word, which is stored in the WordNet database.

In order to improve performance for synonym searches, I used WordNet to look up the synset of every word, and replace that word with the first word appearing alphabetically in the synset. As a result, a word like “win” might be replaced with a less popular but word or phrase like “bring_home_the_bacon” which would appear earlier in the dictionary. The idea was that synonyms would always converge to the same word, which was stored in the index.

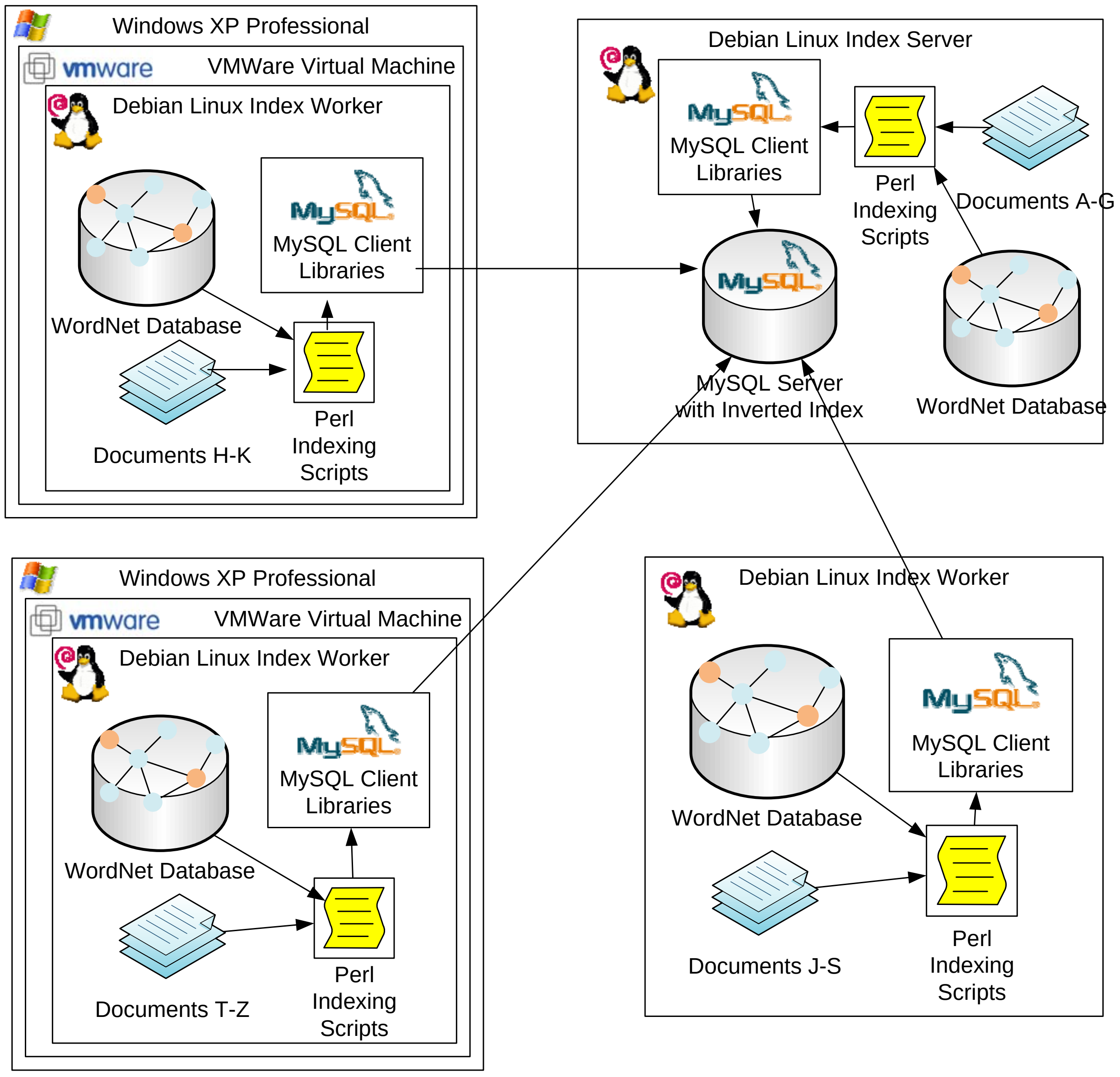


Figure 6 Architecture of indexing procedure

Querying Data

After the data was indexed, I exported the MySQL tables from my computer at home and moved them to a third-party production site which has a publicly accessible web server. I installed WordNet and the same Perl libraries on this server as were needed on the indexing server. In addition, I wrote a new module called CustomSearch::Query which contained the routines to run and sort the queries.

The steps in processing the query data are similar to those in processing the document data to be saved into the index. These steps are illustrated in Figure 7 below.

After the documents relevant to a query are retrieved, they are ranked and sorted according to a normalized TfIdf score.

After reviewing the initial search results, I noticed that the Lesk algorithm did not work to disambiguate most word senses when it only had 1 or 2 words to work with as context. In these cases, I allowed the algorithm to use the 'sense1' word sense of words that with unknown senses. I considered as an alternative retrieving documents which matched all word senses for these unknown words, but I was afraid that this would destroy the system's precision.

Based on what we learned in class, I realized that the best solution would probably be to use relevance feedback, and ask the user what sense of the word they intended to look up. The system would have an additional step then of looking up the query terms in WordNet, returning the list of senses of any ambiguous terms to the user, and using the senses of the words selected by the user. This would be an intuitive next step for this project.

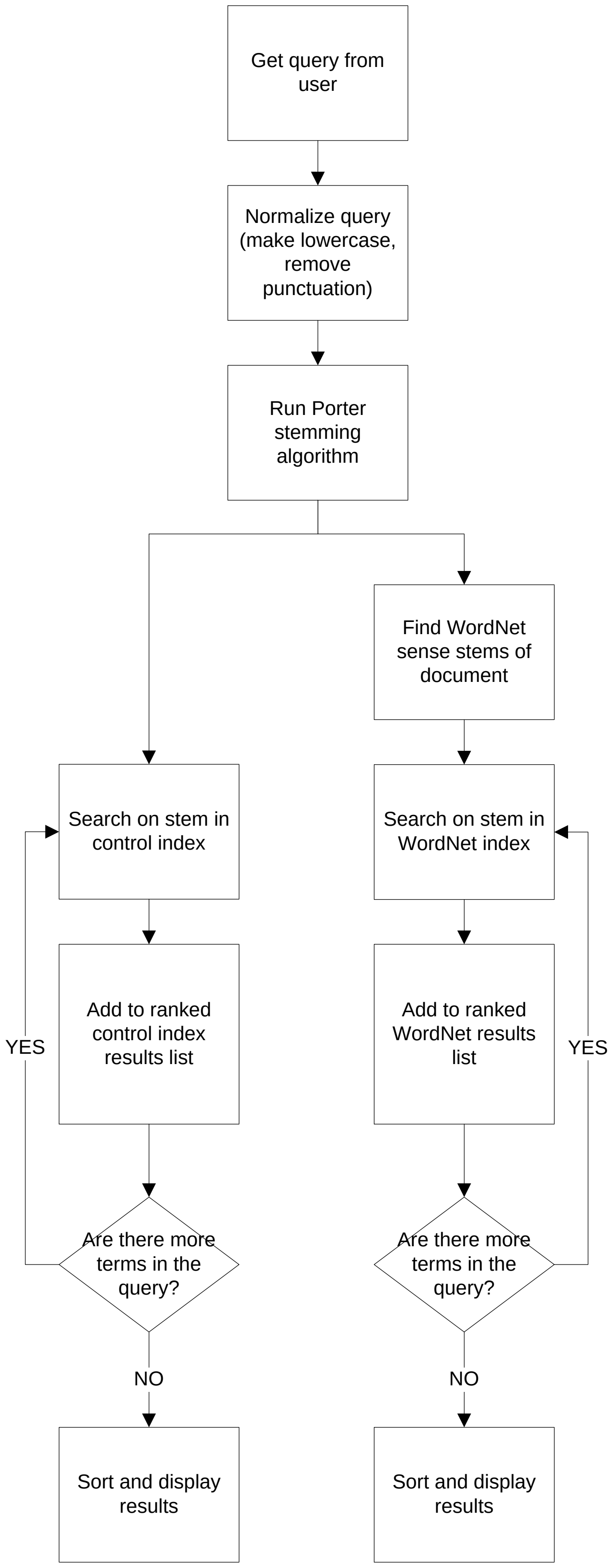

Get query from user
Normalize query (make lowercase, remove punctuation)
Run Porter stemming algorithm
Find WordNet sense stems of document
Search on stem in control index
Search on stem in WordNet index
Add to ranked control index results list
Add to ranked WordNet results list
YES
YES
Are there more terms in the query?
Are there more terms in the query?
NO
NO
Sort and display results
Sort and display results

Figure 7 Steps in querying the index

Suggestions for improvement and next steps

As stated in the previous section, a major weakness in this design is that the queries were too short to reliably determine the senses of the words in the queries.

In addition, the query performance of the system is quite slow, due in part to the large volume of communication between the MySQL database and the Perl scripts. A logical next step would be to refactor some parts of the design so that there was less interaction between the database and script. I believe that more of the document scoring could be moved to the database server, but it would require some trial and error with SQL to find suitably fast queries.

I was reluctant to attempt to tweak the Lesk algorithm, but I would be interested in making the parallel processing method I attempted more reliable and smart. Basically, I partitioned the problem by dividing the document corpus among the indexing workers. There was no communication between the workers except that they all wrote to the MySQL server. This made it inconvenient to monitor the system and restart the servers.

Because these corpuses are not standardized, with a set of agreed-upon queries and expert relevance rankings, it is difficult to rank the quality of the results. I would like try this system with a known corpus so that I can judge the effectiveness of the WordNet indices with some objectivity.

Conclusion

I was not, as I had hoped, amazed at how easy it was to use WordNet to make powerful improvements to queries. WordNet is a powerful database that represents a great investment of time and effort, but it cannot read the user's mind and choose among results to deliver those that the user wants. Frequently, the ranked results for the control index and WordNet index were similar or identical. That is not to say that WordNet could not be used to make an information retrieval system more effective. However, it comes at the price of effort and, as I found, speed.

[1] An Adapted Lesk Algorithm for Word Sense Disambiguation Using WordNet, Satanjeev Banerjee and Ted Pedersen, Lecture Notes In Computer Science; Vol. 2276, Pages: 136 - 145, 2002.

[2] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. New York: Cambridge University Press, 2008. http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html

[3] http://wordnet.princeton.edu/obtain

[4] http://search.cpan.org/~jrennie/WordNet-QueryData/

[5] http://search.cpan.org/~tpederse/WordNet-Similarity-2.05/lib/WordNet/Tools.pm

[6] http://search.cpan.org/~tpederse/WordNet-SenseRelate-AllWords-0.16/lib/WordNet/SenseRelate/AllWords.pm

[7] Pedersen, Ted. " Wordnet-Similarity." http://search.cpan.org/dist/WordNet-Similarity/