Report for part 2 of search engine project

1) Describe the format of your inverted index justifying your choice. What is its total size on disk for the given collection of documents, in absolute terms (number of bytes) and as % of the size of the original collection file. (5 sentences)

Our inverted index is stored as a compact binary file, manipulated by our custom C-based "searchio" Python module. It is comprised of a header (number of documents, number of terms, and offset where postings lists begin — more on that later), followed by the terms, followed by the corresponding postings lists. Each term is stored inline with the relative location in the file of its postings list, and the number of entries in the posting list, which allows postings lists to be lazily loaded from the file as needed.

An index of fullCollection.dat weighs in at about 314.6 MB, or roughly 93% the size of the original collection. By contrast, our original text-based format weighed in significantly *larger* than the original collection.

2) What is the time and space complexity of your createIndex program? Explain your answer. (5 sentences)

The algorithms in createIndex are relatively close to (if not exactly) O(n). After being parsed from their original XML representation, documents are tokenized, and then the frequency of the tokens is collected into an index dictionary before being written to disk. Although some things are looped over several times, there are no nested loops of terms, and things like the tokenize(…) method are intentionally designed to only ever process each character once.

3) At execution time, when and how does your queryIndex program access the inverted index? (3 sentences)

At start up, queryIndex loads the index file as a "sparse index" object, which opens the file, reads in the header and terms, and builds a cache of terms mapped to the file offsets of their postings lists (which takes less than 2 seconds for fullCollection.dat). Then, whenever a term is retrieved from the index, the program seeks to the appropriate location in the file, and loads the corresponding postings list on-the-fly (or else returns the previously loaded version). The sparse index object exposes a normal Python dictionary interface, so the high-performance lazy-loading mechanics are hidden behind normal idioms like index["foo"] and ("foo" in index).

4) Describe the data structures that your queryIndex program uses for manipulating the dictionary (once it is in memory), specifying how you use them to answer wildcard queries. What is the time complexity of answering a wildcard query (without taking the ranking into account)? (7 sentences)

Once the dictionary is loaded into memory, a permuterm-index, using the BTree data-structure (and provided python BTree module), is created from the terms in the dictionary. For each wildcard word w*, I first retrieve all of the terms matching that w*. Retrieving this list of terms is worst-case logarithmic time, ie, O(logN) where N is the size of the permuterm-index BTree. As stated in lecture 3, a permuterm index is approximately quadruple the size of the dictionary. Therefore, retrieving the matching list of terms is worst-case O(log(4n)) where n is the size of the dictionary. I then union the postings lists for each of the terms retrieved to match w*, keeping the maximum wf for each document, and the max idf. This step is linear in the total number of posts in all the retrieved postings lists, where the coefficient is the number of matching terms, ie O(TK) where T is the number of terms matching w*, K is the number of total posts in the retrieved postings lists.
Therefore the total time complexity is O(TK) + O(log(4n)) ~ O(M) where M is the number of documents in the collection.

5) Try your search engine on the given collection with your own queries, a few of each type. Which type of queries does it perform best/worst on, in terms of result relevance? How does this depend on the query words (common/uncommon/specific/etc.)? Provide at least 2 examples for each query type, one positive and one negative.

OWQ's perform worst. They work only with very unique words that that are rarely used, such as 'Anachrophobia'.
WQ's and WPQ's universally seemed to work very well. Nearly every query we ran had at least some highly relevant documents in the top documents.

OWQ:
'Unix' is a negative example - The documents returned were about specific operating systems and specific unix versions or commands. The very specific unix articles were ranked above the generic unix articles.
'Russia' is a more positive example - The top returned documents were either about things in Russia or about a Russian historical event.

FTQ:
'Russia prison' is a negative example - Only one of the top three documents even mentioned Russia, and that document was about the show Prison Break, which is not at all relevant to Russia.
'unix ls scp' is a very positive example - The first document was about the Secure Copy command, and the second document was about the ls command.

PQ:
"Brown University" is a negative example - We got documents that reference Brown University or

about things somewhat related to Brown University.
"Brown university interactive language" is a positive example: the top example was exactly an article about "Brown university interactive language".

BQ:
‘brown OR university’ is a negative example - the top documents were about people that happened to have the last name ‘Brown’.
‘embedded AND operation AND system’ is a positive example - the top document was about embedded operating systems and the second highest document was about embedded systems in general

WQ:

‘theo*’ is a positive example -- all of the top documents were about theory or theorem.
‘ro*velt’ is a negative example - the top documents were not directly about Roosevelt. This is bad compared to how accurate our other tests were!

WPQ:
"*arall* systems" is a positive example - the top two documents were about parallel systems
"graphical *ser" is a negative example - the top documents were not about graphical user interfaces.

Answer!