Alex Hills (ahills)
Jeremy Greenfield (jsgreenf)

Project Part 1 Report

Problem 1

What data structures do you use for storing the inverted index? Do you use skip pointers? Explain your choice. In which format do you store the index on disk?

The structures used while creating the inverted index is actually just a dictionary where words (post-stemmed) are matched to a list of tuples (docID, position). On the disk they're stored in the same fashion, with structure:

Word [(doc1,1),(doc1,10),(doc2,15),(doc2,16),(doc6,152)...]

We do not use skip pointers. Given that the most frequent words are still not used *too* often, it doesn't give a huge speedup, particularly not enough to justify the extra storage space.

Problem 2

How would your data structures for postings change if new documents were added frequently to the collection?

If there were frequent changes, using skip pointers would be even less justified, because they'd have to be continually changing, increasing the frequency of disk access to the posting list and increasing the amount of time needed (if, say, we had to update the skip pointers many times per second...). We *would* sort the word list alphabetically and likely have a table of contents for instant skipping to the right line in the postings list. In terms of updating the list, the only place that would have to change would be the TOC, and the only time it would have to change is if a word were added or removed (not if a document was added).

Problem 3

Examine the lengths of the posting lists for the terms in the full collection and comment on their distribution.

There is generally an inverse relation between the lengths of the terms and the lengths of their associated listings. Additionally, instances of terms are distributed the most heavily among stems of words in common usage, while highly specific or technical strings of characters and numbers have much shorter listings. The lists themselves are longer (often) than the documents themselves, because a single word is represented by a document ID and a position (say: 40213, 17), which is actually 9 characters long (which, if it were the word "yours", adds four characters to the total length).

Problem 4

If you were not provided with the stop words list, how would you have created one?

We would have examined the distribution of words in the documents and taken the top few percent (this number would have to be tweaked to be optimal, though plenty of people have probably done the work figuring out what a reasonably optimal number is). If we wanted to spend manpower on it, a person could then go through that list and remove the words that are actually important contextually and only leave the ones that should be stop words.

Problem 5

Please describe how you process phrase and boolean queries and any optimizations you added for faster processing.

Phrase queries are processed by first removing operators including quotation marks, converting to lower case, removing any stop words, and stemming the remaining tokens. Each (document, location) tuple associated with the first word in the query is evaluated to check whether subsequent words have matching docID and correctly matching offset locations in that document, and non-match results are filtered. If there are m words in the query with an average of n (docID, loc) associated tuples, this algorithm has complexity $O(mn^2)$.

Boolean queries are processed by first escaping 'AND' and 'OR' operators, doing the usual cleaning and stemming, then converting back the escaped operators, then parsing the resulting expression using a boolean parser. The resulting tuple is then parsed recursively: if the left operator is 'OR', the union of the docs returned from parsing the right expression is returned, and if the left operator is 'AND', the intersection is returned.