COS 435, Spring 2015 - Problem Set 3

Due at 1:30PM, Wednesday, March 4, 2015.

Collaboration and Reference Policy

You may discuss the general methods of solving the problems with other students in the class. However, each student must work out the details and write up his or her own solution to each problem independently. For each problem, list the students with whom you discussed general methods of solving the problem (excluding very brief casual conversations).

Some problems have been used in previous offerings of COS 435. You are NOT allowed to use any solutions posted for previous offerings of COS 435 or any solutions produced by anyone else for the assigned problems. You may use other reference materials; you must give citations to all reference materials that you use.

Lateness Policy

A late penalty will be applied, unless there are extraordinary circumstances and/or prior arrangements:

- Penalized 10% of the earned score if submitted by 11:59 pm Wed. (3/4/15).
- Penalized 25% of the earned score if submitted by 4:30pm Friday (3/6/15).
- Penalized 50% if submitted later than 4:30 pm Friday (3/6/15).

Problem 1

Consider an inverted index containing, for each term, the postings list (i.e. the list of documents and occurrences within documents) for that term. The postings lists are accessed through a B+ tree with the terms serving as search keys. Each *leaf* of the B+ tree holds a sublist of alphabetically consecutive terms, and, with each term, a *pointer to* the postings list for that term. (See the B^+ tree example on slide 18 of the slides posted for Feb. 16.)

Part a. Suppose there are 14 billion terms for a collection of 10 trillion documents of total size 100 petabytes. We would like each internal node of the B+ tree and each leaf of the B+ tree to fit in one 32 kilobyte (32*1024 byte) page of the file system. Recall that a B+ tree has a parameter **m** called the *order* of the tree, and each internal node of a B+ tree has between **m+1** and **2m+1** children (except the root, which has between 2 and **2m+1**). Assume that each term is represented using 40 bytes, and each pointer to a child in the tree or to a postings list is represented using 8 bytes. Find a value for the order **m** of the B+ tree so that one 32 kilobyte page can be assigned to each internal node and leaf, and so that an internal node will come as close as possible to filling its page without overflow its page when it has **2m+1** children. If you need to make additional assumptions, state what assumptions you are making.

Part b. For your **m** of Part a, estimate the height of the B+ tree for the inverted index of the collection described in Part a. (Giving a range of heights is fine.) Also estimate the amount of memory needed to store the tree, including leaves but not including the postings lists themselves. Assume that leaves contain between **m** and **2m** terms just as internal nodes do.

Problem 2

In class, we discussed sorting each postings list using PageRank (or some other global, a.k.a static, measure of the document value) with document ID as secondary sort key to give a total order. We did this to support a method of efficiently approximating "highest k" retrieval. (This problem is called "inexact top *K* document retrieval" in our textbook *An Introduction to Information Retrieval*.) Now consider the following alternative: use a tiered index where each tier represents a range of PageRank values. The ranges for different tiers are disjoint. Postings lists within a tier are sorted on document ID.

Part a. Describe an algorithm for using postings lists sorted on PageRank with document ID as secondary sort key to quickly find a set of k documents that are good candidates for the top k documents satisfying a given multi-term query. Assume that the scoring function for ranking the documents with respect to the query includes PageRank as a feature, as well as term-based features, and possibly other features. An algorithm was broadly sketched in class. Give a more detailed version (or a different algorithm if you like). Be clear about which documents are considered for ranking and how they are chosen.

Part b. Describe an algorithm that uses the tiered index described above to quickly find a set of k documents that are good candidates for the top k documents satisfying a given multi-term query. Assume the same scoring function for ranking the documents with respect to the query as in Part a, including using the exact PageRank. Be clear about what documents are considered for ranking and how they are chosen.

Part c. Will the results of your algorithms for Parts a and b be the same for the same query? Justify your answer. If you answer "no", which algorithm do you think would give better results; justify your answer.

Part d. Is one of the algorithms of Parts a and b computationally faster than the other? Justify your answer.

Problem 3

Section 4.3 and Figure 4.4 of our textbook *An Introduction to Information Retrieval* present the "single-pass in-memory" algorithm for constructing an index given a sequence of pairs (term, docID) for a corpus. Using pairs of this type, one constructs an inverted index that only contains the list of documents for each term.

Part a. Modify the "single-pass in-memory" algorithm to process a sequence of tuples (term, docID, position in doc., attributes) and produce an inverted index that, for each term, records all positions at which the term occurs in each document and attributes of each occurrence. Clearly describe how each tuple is processed. You may assume that the input sequence of tuples is in the order produced by reading one document at a time, token by token, and producing the appropriate tuple each time a term is seen.

Part b. Does your algorithm of Part a. have the same efficiency (time) as the original algorithm of Section 4.3? Explain.

Part c. Compare the amount of buffer memory (not disk) needed by your modified "single-pass in-memory" algorithm of Part a with the amount needed by the "blocked sort-based indexing" algorithm when the latter algorithm is also processing a sequence of tuples (term, docID, position in doc., attributes). (This version of the "blocked sort-based indexing" algorithm was presented in class.) Do **NOT** consider only big-O measures of memory as a function of the number of tokens T. What, if any memory is wasted? For a given buffer size, can the "single-pass in-memory" algorithm and the "blocked sort-based indexing" algorithm process the same number of tokens?