Group members (Name, Sjsu ID) :

Homework 3

---

1. In Section 4.5.1 we discussed the performance characteristics of various dictionary data structures. We pointed out that hash-based implementations offer better performance for single-term lookups (performed during index construction), while sort-based solutions are more appropriate for multi-term lookups (needed for prefix queries). Design and implement a data structure that offers better single-term lookup performance than a sort-based dictionary and better prefix query performance than a hash-based implementation.

→ Do Exercise 4.5. Implement with pseudocode. Rather than just support prefix/begins with queries like foo*, you should also support more general queries like *foo (ends with foo), and *foo* (contains foo).

Ans:
We will employ a hybrid approach (different dictionary data structures for index time processing and query time processing): ***a hash-based dictionary and a sort-based dictionary:***

(1) ***A hash-based dictionary:***
The dictionary represents a major bottleneck in the index construction process, and *lookups and inserts* should be as fast as possible. Therefore we will use a hash table instead of sort based dictionary during the index construction process.

Algorithm:

1. Create a hash-based dictionary
2. Add tokens to dictionary till memory is exhausted
3. Sort the terms in the dictionary in lexicographic order
4. Write the dictionary to a file on a disk
5. Go to step 1 till are documents are processed
6. Finally, merge all sorted files using external merge sort to create a file with merged index.

```
buildIndex(inputTokenizer)
    output_ file = NEWFILE()
    dictionary = NEWHASH()
    while (free memory available)
    do token <-- next(inputTokenizer)
      if term(token) not in dictionary
        then postings_list = ADDTODICTIONARY(dictionary, term(token))
        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
      if full(postings_list)
        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
        ADDTOPOSTINGSLIST(postings_list, docID(token))
    sorted_terms <-- SORTTERMS(dictionary)
    WRITEBLOCKTODISK(sorted_terms, dictionary, output_ file)
    return output_ file
```

```
n = 0
while (all documents have not been processed)
do  n = n + 1
  tokens = getNextTokens(docs)
  fn = buildIndex(tokens)
MERGEFILES(f1, . . . , fn; fmerged)
```

(2) ***a sort-based dictionary:*** This index will be built using merged index file which is created at the end of index creation. It provides efficient support for single-term lookup and prefix lookup.

In order to support suffix lookups efficiently, we will maintain an additional sort-based dictionary index which will have reversed terms (eg. barfoo will become oofbar).
When suffix query is issued we will reverse it (eg: *foo will become oof*) and then we will execute it against this index to fetch results. An alternative approach for this could be the use of a suffix tree (or its variant compressed suffix tree).

Supporting queries of the form *foo* seems more complicated. One approach that we can think of is using char-gramming while constructing an index.
Or we can construct a giant suffix tree that will regard the corpus as one long string and each position in the text will be considered as a suffix.

2. Suppose we were using sort based indexing. After index time we want to support adding new documents to the corpus. Suggest how to do this, give pseudo-code. Analyze the performance of your proposal.

Ans:

Approach 1:

As we are dealing with only the addition of new documents to the index, we can update the existing index instead of constructing a new index from scratch. We will build a separate sort based index for new document. And then merge this index with the original index through a linear scan (similar to merge based strategy used in merge-based index construction) which will result in the new index.

Complexity: O(T) where T is the total number of tokens

Approach 2:
Keep two indexes: 1) Main static index and 2) index for new document additions.
We can run a query on both indexes and combine final results from each index. When the second index reaches some threshold we will merge it into the main index using merge based strategy.

3. The worst-case complexity of the document-at-a-time algorithm with heaps is
$\Theta(Nq \cdot \log(n) + Nq \cdot \log(k))$.

(a) Characterize the kind of input (i.e., document score distribution) that represents the worst case.

Ans:

The worst case scenario will occur when we have to carry out Reheap operations on terms heap and results heap for every query term and for every document. So the input for worst case scenario will be when all the documents have all the query terms in it (i.e. each document has all the query terms). Due to this we need to perform $(Nq \cdot \log(n))$ operations on terms heap, for every query term to restore the heap after every posting and $(Nq \cdot \log(k))$ operations on results heap, whenever a new input document is added to restore k top results.

(b) Prove that, on average, the complexity of the algorithm is strictly better than $\Theta(Nq \cdot \log(k))$. You may assume that $k>n$ and that document scores are distributed uniformly, that is, every document is equally likely to have the highest score, second-highest score, and so forth.

Ans:

In an average case scenario, not all the query terms are present in all the documents. So the total number of posting list will be smaller.

In algorithm for document_at_a_time_with_heaps,

sort terms in increasing order of nextDoc // establish heap property for terms
while terms[0].nextDoc < ∞ do
d ← terms[0].nextDoc
score ← 0
while terms[0].nextDoc = d do
t ← terms[0].term 12 score ← score + log(N/Nt) · TFBM25(t,d)
And so on….

we traverse all the documents and score the document through this while loop using single posting list. Using a single posting list of a term, we will score the document containing that term. Using this we will restore the terms heap and results heap together. Since the posting list will be smaller in average case scenario

Doing so, we will have kth best document on top of the heap. So to restore results set, it would take less operations than Θ(Nq · log(k)) since probability of swapping at top is more.