*2021 FALL*
***SOLUTION SKETCH***

***SIZING***

(a) *In total we would then have T = 2*10^3*10^9 = 2*10^12 = 2 trillion tokens. Heaps' law, assuming some typical values for k (30) and b (0.5), then says that we should have seen about M = 30 * sqrt(T) = 42.43 million unique tokens.*

(b) *[The exam contains an unfortunate typo: "700,000" should have been "700,000,000" [7*10^8], or about 3.5% of all word occurrences. The students are of course not liable for this.] Assuming that the 10 million documents (1% of the full collection) are representative we would then expect "of" to occur about 7*10^10 times across the full collection. Assuming that "of" is still the second most frequent term across the full collection and that Zipf's law is a decent predictor for this collection, we'd then expect the most frequent term to occur about 1.4*10^11 times and the third most frequent term to occur about 4.67*10^10 times. Zipf's law says that $cf_i = c/i$, and for simplicity we've here assumed c=1. A semi-realistic estimate of c could be deduced from figure 5.2 in the textbook.*

(c) *The previous estimate is about collection frequencies, but the length of a posting list equals the term's document frequency. We're talking about the most frequent terms here, so it's not unreasonable to expect that each of these terms are found in almost every document. A conservative upper bound is thus that each of these posting lists will be a billion entries long. If you apply the heuristic that places skips every sqrt(10^9) = 31623 entry, you'll have 5 * sqrt(10^9) = 158,114 entries in your skip lists in total across all 5 skip lists.*

(d) *Assume the same conservative bounds as above. In the posting lists each gap would then be 1, a number you can gamma code using a single bit. So for the 5 posting lists in total you'd need about (5 * 10^9 * 1 / 8) = 625 Mb. Slightly more, because the assumption that each gap is 1 is conservative, and numbers larger than 1 need more than a bit. In the skip lists each gap would be 31623, a number you can gamma code using 29 bits. So for the 5 skip lists in total you'd need about an additional (5 * 31623 * 29 / 8) = 573 Kb. That makes for a grand total of about 626 Mb.*

***QUERY SUGGESTIONS***

(a) *See figure 4.6 here. The mapper consumes the log files and produces a list/set/stream of (f(<query>), 1) pairs. Here, f is a function that does some basic normalization, e.g., based on case or whitespace. For example, the mapper would emit [..., (oslo festival, 1), (jonas gahr støre, 1), (oslo pizza, 1), (jonas gahr støre, 1), ...]. The reducer would consume a pair consisting of a (possibly normalized or otherwise transformed) query and a list of ones, and emit a pair consisting of the query and the list sum. For example, the reducer would consume (jonas gahr støre, [1, 1, ...., 1]) and emit (jonas gahr støre, 314).*

(b) *The earlier we filter the better, generally. So injecting the filtering logic into the mapper rather than the reducer makes sense, since this might generate less work for the reducer downstream.*

*The function f above could be augmented to scan <query> against the dictionary: Organize the dictionary as a trie, and do a trie walk similar to the Aho-Corasick algorithm and as done in Assignment B. Emit, e.g., ("REDACTED", 1) if you get a match in the dictionary, to produce statistics on how many queries that are filtered away. Or don't emit anything in these cases if you don't care about such statistics or collect such statistics in a different way.*

(c) *[For simplicity, we assume that the dictionary fits in memory on one machine. If that is not the case, we partition the dictionary across N machines, fan out the lookup to each, and then merge the results.] Could be as simple as keeping the entries in a lexicographically sorted list, with two binary searches (to locate the beginning and end) and a scan of the identified range to find the 5 entries having the highest associated frequency count. Another approach, and which also compresses our entries so that we could fit more into memory, is to compile the dictionary down to a trie or finite state automaton, and then consume the prefix. All nodes "below" the node we arrive at are then visited to find the 5 entries having the highest frequency count.*

(d) *With either solution above, the shorter the prefix the more to scan. So for very short prefixes (say, all prefixes of length 1-2 characters or all prefixes that are very frequent query-starters), simply pregenerate the results ahead of time and serve back these cached results.*

(e) *See, e.g., the "Tries for Approximate String Matching" paper by Shang and Merrett.*

(f) *Could use a suffix array as implemented in Assignment B, see, e.g., the "Suffix Arrays: A New Method for Online String Searches" paper by Manber and Myers. Could also imagine using a standard inverted index as implemented in Assignment C but with support for phrase searches, but this has the complication that we'd only get a match if the last word in the query prefix is a complete word. So there are some significant pitfalls there.*

***RELEVANCE***

(a) *Dynamic indexing with logarithmic merging comes to mind. See, e.g., here. The question is about searchability and ensuring low indexing latency, not the role of freshness/age in ranking.*

(b) *Lots of room for creativity here! For example:*
- *Include the document's age as a feature/component in your ranker. The lower the value, the higher the score.*
- *Let the document's static score (called g(D) in the textbook) be a measure of how trustworthy the document's publisher is. The higher the value, the higher the score. The g(D) values could be, e.g., PageRank values computed from a graph that encodes which publishers/sites that link to each other.*
- *Apply named entity recognition to the content during document processing. Stuff these identified strings into a document field that gets boosted during evaluation.*

(c) *Some discussion points:*
- *You'll want to build up a data table similar to that shown in Table 15.3. This implies collecting both (i) features and (ii) judgments/labels. For (i), features like cosine score, window width term proximity, g(D), document age and many, many more make sense. For (ii), you could look at clicks on the SERP in your live system (much more scalable than human judgments): E.g., top- or high-ranking documents that match or exceed expected*

*click patterns (adjusting for position bias) are "relevant". The ones that don't are "nonrelevant". Could also sample random document to supplement "nonrelevant". With such a data table in hand, you could do something like shown in Figure 15.7, and sort by distance to the plane. (So, given a document, the ranker emits a score.)*

- *A better solution is a ranking SVM: Adjust (ii) by analyzing user click patterns to capture lots of "document preference pairs" for each query as discussed in Section 15.4.2. So, for each query we have a collection of pairs that says that document i should be higher ranked than document j for that query. Logically, we can then build a data table of feature differences for these pairs, and train the SVM from the data in this table. (So, given a pair of documents i and j, the ranker tells you if document i should appear higher up in the result set than document j.)*

(d) *Define a suitable metric that we can use to compare the two (NDCG, MAP, F-score, etc). Basic machine-learning hygiene about not evaluating the model on the data from which it was trained needs to be obeyed. Could discuss things like training/testing/hold-out splits and cross-validation (see the "Classifier Evaluation" supplementary paper). Also A/B testing.*

(e) *Techniques from Chapter 7 could all be discussed: Index elimination, champion lists, sorting posting lists by g(D), tiered indexes, impact ordering, etc. Caching/precomputing result sets for the most frequent queries (as determined by query log statistics) and serving these.*