INF3800/INF4800

Assignment C

WHAT TO DELIVER:
Please submit a zip containing both the implementation task (the entire project) and the pencil-and-paper tasks. Exercise 2 is optional for INF3800 students.

DEADLINE:
The assignment must be submitted, using Devilry, by 13.03.17 at the latest.

On paper

Exercise 1:

Query evaluation

a) Explain what skiplists are and how they work. Are skiplists necessarily always beneficial for performance?
b) Explain the difference between term-at-a-time and document-at-a-time evaluation. Which approach would you use for an index where the posting lists are impact-ordered, and why?
c) Postings lists can be ordered according to a static quality score $g(d)$. Explain why this can be beneficial, and provide some examples from web search of what such a static quality score might represent.

(This exercise is taken from the V13 final exam)

Exercise 2: (Optional for INF3800)

Index compression

The procedure to compress a search index usually includes encoding the gaps between consecutive postings, based on appropriate compression techniques.

a) How would you compress the value 1337 using VB coding? How would you compress the value 13 using gamma coding?
b) Kurt has a big search index, and wonders whether he should choose VB or gamma coding. Which advice would you give to Kurt?
c) Explain how index compression affects the performance of a search system.
d) Entropy is a central concept in compression. For which discrete distribution over N values does entropy have its maximum, and what is its maximum value?

(This exercise is taken from the V11 final exam)

In code

Many search engines use AND as the default operator between query terms when evaluating user queries. E.g., the user query *foo bar baz* would require that all three query terms *foo*, *bar* and *baz* are present in each document in the result set. This is problematic for queries that contain many query terms, since it's then easy to arrive at an empty result set or a result set where relevant documents have been omitted.

A common approach to this problem is to instead use OR as the default operator, possibly only after an AND interpretation has first been verified to yield 0 results. This is problematic for efficiency reasons, since OR queries typically are a lot heavier to evaluate than AND queries, and, in the case of the two-query approach, it might substantially increase the load on the search servers. Additionally, the user gets a false sense of how many relevant documents that satisfy the information need, since an OR interpretation will greatly inflate the size of the result set.

In this assignment you will be asked to implement an efficient query evaluator that performs N-of-M matching over a memory-based inverted index. I.e., if the query contains M query terms, each document in the result set will be guaranteed to contain at least N of the M terms. For example, 2-of-3 matching over the query *foo bar baz* would be logically equivalent to the following recall predicate:

$$(\textit{foo}\ \text{AND}\ \textit{bar})\ \text{OR}\ (\textit{foo}\ \text{AND}\ \textit{baz})\ \text{OR}\ (\textit{bar}\ \text{AND}\ \textit{baz})$$

Sensible values for N and M will vary between queries, hence the evaluator should use the fraction N/M as a recall parameter. Note that N-of-M matching can be viewed as a type of "soft AND" where the degree of recall can be smoothly controlled to mimic either an OR evaluation (1-of-M) or an AND evaluation (M-of-M) or something in between.

The retrieved results should be ranked according to relevancy. For example, continuing the previous example, it's reasonable to expect that a document that contains all of *foo*, *bar* and *baz* will, ceteris paribus, be ranked above a document that contains *foo* and *bar* but not *baz*. No more than the 10 highest-ranked results should be returned back to the client.

Download assignmentC.zip from the course homepage. Unzip the Java project, and load the project into Eclipse. Note that ObligCTest.java contains two JUnit tests (respectively on the CRAN and WeScience document collections) that you can use as a basis for testing your implementation.

The implementation of QueryEvaluator.java is incomplete, and it is your task to finish it. Basically, you should be able to run ObligCTest.java, which should build an inverted index over a small test file, issue a test query and check the search results against the expected ones. The only task is to complete the evaluate(...) method in QueryEvaluator.java, so that it returns a sane ResultSet. You should try to factor your implementation into sub-methods to improve the readability and modularity of your implementation.

Here are a few hints:
(1) for the evaluation, you should perform document-at-a-time-scoring rather than query-term-at-a-time scoring.
(2) some data structures like heaps and sieves are already provided, exploiting them will make your implementation easier!