COMP6714 (13S2) MID-TERM EXAM

TIME ALLOWED: 1 HOUR

Name: _____

Student ID: _____

NOTE:

(1) Answer the questions briefly. Lengthy but irrelevant answers will be penalized.

(2) In most of the questions, you need to show major steps.

Q1. (*30 marks*)

Answer the following questions.

(1) What is the heuristic method to determine the execution order using the binary list merge algorithm to answer *conjunctive* keyword queries? Given a counter-example where this heuristic does not work well.

Follow the order of increasing cardinality of the postings lists of query keyword (i.e., from rarest to commonest).

It fails when two keywords are negatively correlated which, if processed first, will create a tiny intermediate result. Or when the two rarest query keywords are positively correlated.

(2) Given at least two reasons (with simple examples) why language identification is important when indexing documents.

One is to be able to correct parse dates/numbers, etc. The other is to be able to correctly perform stemming/lemmatization.

(3) Why specialized algorithms are needed to construct inverted index for large document collections?

Because the inverted index for the data is typically much larger than the memory size.

(4) What is the Heaps' Law and what is the Zipf's Law?

Heaps' law estimates the vocabulary size M as proportional to T^b where T is the number of tokens in the document collection, and b is a collection-specific constant.

Zipf's law estimates the collection frequency of the i-th most common term as proportional to $1/i$.

(5) Give an intuitive explanation of the *two* major factors we consider when choosing the best candidate for *context-sensitive* spelling correction. (For example, `taw` may be corrected to either `the` or `thaw`)

Based on the noisy channel model, one is the error model — the distance between the candidate and the observed spelling, and the other is the prior, which can be taken from a background language model, surrounding words, etc.

Q2. (*20 marks*)

Complete the pseudocode of the function `Q2` (shown below) that answers the Boolean keyword query "A `AND (NOT` B`)`, where A and B are two different keywords. Make sure your pseudocode is easily readable (i.e., with indentation and comments if necessary).

You can assume the following functions/methods on `LA` or `LB`, which represents (sorted) postings lists for keyword `A` and `B`, respectively:

- `cur()` returns the current docID in the list;
- `eol()` returns `TRUE` if the current list is exhausted;
- `next()` moves the cursor to the next posting.

```
function Q2(LA, LB)
  // Let LA and LB be the corresponding inverted lists
  // of A and B, respectively
  ANSWER = []
  ...
  ...
  return ANSWER
end
```

Solution:

```
function query(LA, LB)
  // Let LA and LB be the corresponding inverted lists
  // of A and B, respectively
  ANSWER = []
  while not (LA.eol() or LB.eol())
    if LA.cur() < LB.cur() then
      ANSWER.add(LA.cur())
      LA.next()
    elsif LA.cur() == LB.cur() then
      LA.next()
```

```
      LB.next()
    else // LA.cur() > LB.cur()
      LB.next()
    end
  end
  return ANSWER
end
```

Q3. (*25 marks*)

Given a list of sorted strings of possibly different length, describe a method to use binary search to answer prefix query `P*`, where `P` is a non-empty string (You may use an additional $O(n)$ space for some auxiliary data structures)? Why B^+-tree index is still preferred over this binary-search-based solution for large collection of strings?

Solution:
Use an array of pointers, A, to point to each string in sorted order (i.e., `*A[i] < *A[j]` if `i < j`. Then we can perform binary search on the pointer array.

The search cost is $O(\log_2(n))$, and with B^+-tree, it is $O(\log_f(n))$, which is usually a constant.

Q4. (*25 marks*)

Consider using one of the three dynamic indexing methods, namely *immediate merge*, *no merge*, and *logarithmic merge*, to build the inverted index for a collection. Let $|C|$ be the total number of bytes of the documents in the collection, M be the memory size in bytes, and B be the number of bytes in a disk block.

We only consider the total number of blocks read from or written to the disk as the I/O cost; and you can safely ignore the ceiling or floor functions in the analysis. For example, reading 1000 bytes and writing 2000 bytes from the disk costs $\frac{3000}{B}$ I/Os.

(1) How many sub-indexes will the *no merge* method create? What is the total I/O cost of indexing the collection?
(2) How many sub-indexes will the *immediate merge* method create? What is the total I/O cost of indexing the collection?
(3) How many sub-indexes will the *logarithm merge* method create (you may consider the worst case)? What is the total I/O cost of indexing the collection?

Solution:

(1) No merge: It results in $\frac{|C|}{M}$ sub-indexes. Total I/O cost is:
   - Reading the collection: $\frac{|C|}{B}$
   - Writing: $\frac{|C|}{B}$

   So total cost is $2\frac{|C|}{B}$

(2) Immediate merge: It results in 1 sub-indexes. Total I/O cost is:
- Reading the collection: $\frac{|C|}{B}$
- It performs $K = \frac{|C|}{M} - 1$ times merge; the i-th merge is between an in-memory index of size M and an on-disk subindex of size $i \cdot M$. Therefore, the totol cost is

$$\frac{1}{B} \cdot \sum_{i=1}^{K} (i * M + (i+1) * M) = \frac{M}{B} \cdot ((\frac{|C|}{M})^2 - 1)$$

So total cost is $\frac{|C|}{B} + \frac{M}{B} \cdot ((\frac{|C|}{M})^2 - 1) = O(\frac{|C|^2}{MB}) = O(\frac{|C|}{M} \cdot \frac{C}{B})$

(3) Logarithm merge: It results in $\log \frac{|C|}{M}$ sub-indexes. Total I/O cost is (assume $\frac{|C|}{M} = 2^h$ for some positive integer h)
- Reading the collection: $\frac{|C|}{B}$
- After 2^h rounds (each round builds an in-memory index of size M), there is only 1 sub-index on the disk with generation number h. Hnce it performs:
  - once merge of two generation $h-1$ sub-indexes
  - twice merge of two generation $h-2$ sub-indexes
  - ...
  - 2^{h-1} times merge of two generation 0 sub-indexes.

  Note that for simplicity and uniformity, we assume we write down conflicting index before triggering the merge (although a clever algorithm should be able to figure out the subindexes needed in the merge and perform only one merge). So the total cost is

$$\frac{1}{B} \cdot \sum_{i=0}^{h-1} 2^i \cdot (2 * 2^{h-i-1} + 2^{h-i}) \cdot M = h \cdot 2^{h+1} \cdot M = \log \frac{|C|}{M} \cdot 2 \cdot \frac{|C|}{B}$$

So the total cost is $\frac{|C|}{B} + \log \frac{|C|}{M} \cdot 2\frac{|C|}{B}$, or $O(\log \frac{|C|}{B} \cdot \frac{|C|}{B})$

(Note: it is possible to assume $\frac{|C|}{M} = 2^h - 1$ so that there is indeed h subindexes on the disk; this will only reduce the total cost by ...)