Web Information Retrieval (67782) Ex1: Index Structure Analysis

Submitted by: Geffen Brenman, Liron Behar

1 General Explanation and Diagram

We decided to create two index data structures (two dictionary classes) and a few files that contains the data of the reviews. We created an abstract class Dictionary that the following two classes inherits from her:

* **TokensDictionary** – Contains all the data about the tokens of the reviews. We compressed the data using k-1 in k front coding method using k = 24 . We wrote into one file string that is an alphanumeric concatenation of all the tokens appears in all the reviews. For each k-successive token we removed a mutual prefix that we calculated. We have a frequencies file that contains the total frequency of each token, and a file that contains positions to the posting list of each token in the posting lists file.

The dictionary class contains the following arrays:

int[] sizeToken – contains the size of each token in the dictionary.

int[] prefixSize – contains the size of the mutual prefix with the previous token (0 for first token in block).

int[] postingLists – contains the positions to the posing lists of each token.

int []tokenPointer – pointers to each token in the block.

int[] totalFrequencies – contains the frequencies of each token in all the reviews.

* **ProductIdDictionary** – Contains all the products of the reviews . We compressed the data using k-1 in k front coding method using k = 6. We wrote into one file string that is an alphanumeric concatenation of all the products id's appears in all the reviews. For each k-successive product we removed a mutual prefix that we calculated. We have a frequencies file that contains the total frequency of each token, and a file that contains positions to the posting list of each token in the posting lists file.

The dictionary class contains the following arrays:

int[] sizeToken – contains the size of each token in the dictionary.

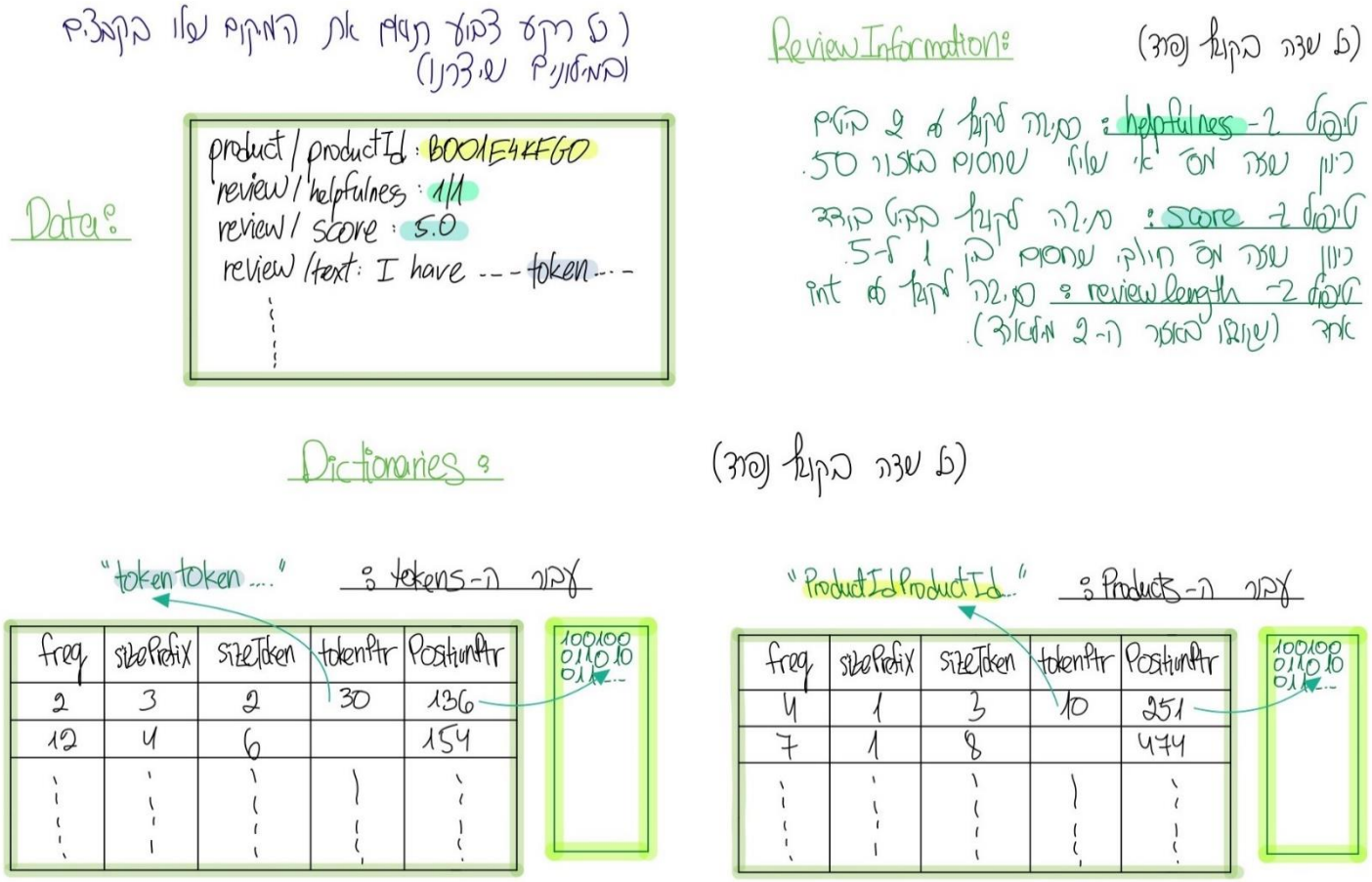
int[] prefixSize – contains the size of the mutual prefix with the previous token (0 for first token in block).

int[] postingLists – contains the positions to the posing lists of each token.

int []tokenPointer – pointers to each token in the block.

int[] locationsReviews – pointers to where the product id starts in the long string.

Each dictionary has a corresponding file that saves the posting list of each product id / token. Those files are encoded using gamma code sequence of numbers. For the tokens the numbers represent the review id's of the current token following the frequency of the token (in the recent review id). For the product dictionary we wrote only the review id's that are connected to the current product id. The review id's are in an ascending order hence we encoded them using the gap encoded method.



2 Main Memory Versus Disk

When creating the IndexReader object we create two dictionary objects (for the productid dictionary and the tokens dictionary) and we load into the main memory all the data about positions and sizes (as noted above in the arrays details). All the posting list data we keep on the disk.

3 Theoretical Analysis of Size

N – Number of reviews
M – Total number of tokens (counting duplicates as many times as they appear)
D – Number of different tokens (counting duplicates once)
L – Average token length (counting each token once)
G – Average productId length (counting each productId once)
F – Average token frequency, i.e. number of reviews containing a token.
P – Number of different products id's (counting duplicates once)
S – Average suffix size for token (without it's prefix)
Q – Average prefix size for token
A – Average suffix size for productId (without it's prefix)
B – Average prefix size for productid
C – Average number of reviews id's for a single token

For the initial parsing of the data information we create 3 files that contains: scores, helpfulness and review length fields. The size of the score file is $N \cdot bytes$. The size of the helpfulness file is $(2 * 2) \cdot N \cdot bytes$. The size of the reviewLengths file is $N \cdot int = 4N \cdot bytes$. In total those files size is $9N \cdot bytes$.
In each dictionary we have 4 columns of size D. Also we have one more column of size $\frac{D}{K}$. In hence the size of each dictionary is: $D(int + int + int + int) + \frac{D}{K} \cdot int = 16D + \frac{4D}{K}$.
TokensDictionary: $16D + \frac{4D}{24} = 16D + \frac{D}{6}$ ProductIdDictionary: $16P + \frac{4P}{6}$
As for the file that contains the concatenation string it's overhead in Java is:
$8 \cdot bytes(header) + 4 \cdot bytes(reference\ to\ the\ array\ of\ chars) + 4 \cdot$

$bytes(offset) + 4 \cdot bytes(length\ of\ the\ string) + 4 \cdot bytes(hashcode) +$
$\left(8 \cdot bytes(header) + 4 \cdot bytes(length)\right)(the\ array\ of\ chars) = 36 \cdot bytes$

In our case the concatenation length is: $\left(\frac{D}{K} \cdot L + \left(D - \frac{D}{K}\right) \cdot S\right) \cdot char =$
$2\left(\frac{D}{K} \cdot L + \left(D - \frac{D}{K}\right) \cdot S\right) \cdot bytes$. Overall the concatenation size
$\left(36 + 2\left(\frac{D}{K} \cdot L + \left(D - \frac{D}{K}\right) \cdot S\right)\right) \cdot bytes$.

According to the calculation we saw in the lecture: the amount of bytes needed in order to encode F is $3 \cdot bytes$.

Total Size:

TokensDictionary:

- Frequency: $3D \cdot bytes$
- Pointer to the inverted index: $4D \cdot bytes$
- For 1 out of every k (=24) tokens:
    - $(L + int(term\ pointer)) \cdot \frac{D}{K} = (L + 4 \cdot bytes) \cdot \frac{D}{24}$
- For k-1 out of every k tokens:
    - $(S + L + Q) \cdot (K - \frac{D}{K})$
- **Total:** $7D \cdot bytes + (L + 4 \cdot bytes) \cdot \frac{D}{24} + (S + L + Q) \cdot (K - \frac{D}{K})$

ProductIdDictionary:

- Frequency: $3P \cdot bytes$
- Pointer to the inverted index: $4P \cdot bytes$
- For 1 out of every k (=6) tokens:
    - $(G + int(term\ pointer)) \cdot \frac{P}{K} = (G + 4 \cdot bytes) \cdot \frac{P}{6}$
- For k-1 out of every k tokens:
    - $(A + G + B) \cdot (K - \frac{P}{K})$
- **Total:** $7P \cdot bytes + (G + 4 \cdot bytes) \cdot \frac{P}{6} + (A + G + B) \cdot (K - \frac{P}{K})$

As for the posting lists file we decided to use the k-1 in k front coding method. Due to the fact that the number of reviews id's for each token and the length of the gamma coding is unknow, it is hard to give theoretical

analysis to the size of the memory needed to store the posting lists. We can calculate approximately this size: the average number of bytes for encoding a number in gamma code is 4 bytes (including one byte of overhead).
TokensDictionary:$(reviewid + frequency) \cdot C \cdot gammacode = 8C \cdot bytes$
ProductIdDictionary: $reviewid \cdot C \cdot gammacode = 4C \cdot bytes$

From the written above we can conclude that for each token we write successively reviewid and frequency posting list. And for each product ID we write in the posting list only the reviews id's.

4 Theoretical Analysis of Runtime

- IndexReader(String dir):
  This function builds the dictionary objects (tokens and productid) by reading all the inverted index files. While building the arrays in the dictionaries we access the files as the array's sizes, so we will calculate the running time of the constructor as the I/O complexity.
  In total he sizes of all the arrays in the TokensDictionary class is: $4D + \frac{D}{K}$ and in the ProductId Dictionary: $4P + \frac{P}{K}$. The running time of this function is: $\frac{(D+P)(4k+1)}{K}$.
- getProductId(int reviewId):
  This function finds the index of the product according to the reviewId, then it reads from the posting lists file by the given positions and returns the productid. The running time of this function is: $O(P \cdot C)$
- getReviewScore(int reviewId):
  This function returns the score of the given reviewId, it reaches for the score in the file by position. The running time of this function is: $O(1)$.
- getReviewHelpfulnessNumerator(int reviewId):
  This function returns the numerator of the given reviewId, it reaches for the numerator in the file by position. The running time of this function is: $O(1)$.
- getReviewHelpfulnessDenominator(int reviewId):
  This function returns the denominator of the given reviewId, it reaches for the denominator in the file by position. The running time of this function is: $O(1)$.
- getReviewLength(int reviewId):
  This function returns the review's length , it reaches for the length in the file by position. The running time of this function is: $O(1)$.

- getTokenFrequency(String token):
  see in getReviewsWithToken.
- getTokenCollectionFrequency(String token):
  This function calls the binary search function to search for the index of the given token - $O\big(Dlog(D)\big)$. Afterwards it pulls the frequency of the token from an array saved in the main memory – using the index we got from the search function - $O(1)$. The running time of this function is: $O\big(Dlog(D)\big)$.
- getReviewsWithToken(String token):
  This function calls the binary search function to search for the index of the given token - $O\big(Dlog(D)\big)$. Afterwards it reads the posting list file by positions of the given token - $O(C)$. The running time of this function is: $O\big(Dlog(D)\big) + O(C) + O(Function^*)$. *Function – the function that creates an Enumeration<Integer> from a given int array.
- getNumberOfReviews():
  When creating the IndexReader Object we read from the inverted index the number of reviews and holds it into a local field. The running time of this function is: $O(1)$.
- getTokenSizeOfReviews():
  When creating the IndexReader Object we read from the inverted index the number of number of tokens and holds it into a local field. The running time of this function is: $O(1)$.
- getProductReviews(String productId):
  This function calls the binary search function to search for the index of the given productId - $O\big(Plog(P)\big)$. Afterwards it reads the posting list file by positions of the given productId - $O(C)$. The running time of this function is: $O\big(Plog(P)\big) + O(C) + O(Function^*)$. *Function – the function that creates an Enumeration<Integer> from a given int array.