## Analysis

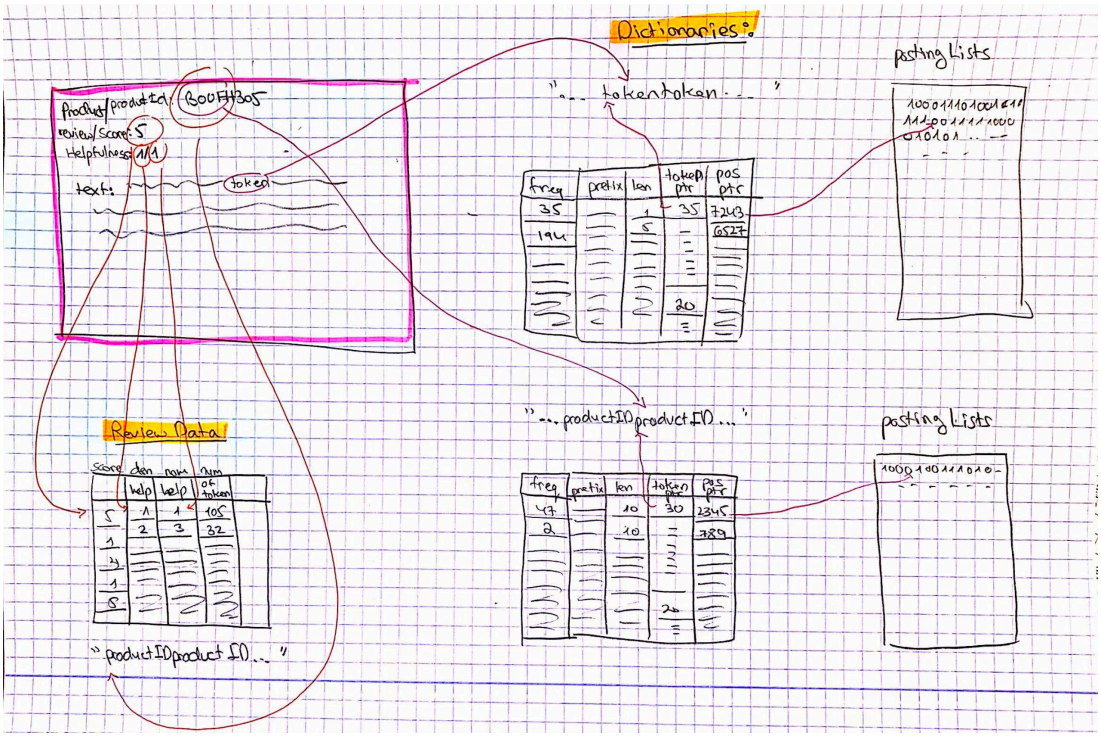
1. In our implementation there are 2 index data structures and another data structure that holds meta-data of the reviews. The 2 index structures are implemented as one object class (dictionary), but used for different types of data:
    - Token dictionary: Has all the tokens of the reviews in the file compressed using k-1 to k front coding method, where our chosen k is 100. It's implementation includes one large `String` that is an alphabetical concatenation of all the tokens of the reviews, where we discard a mutual prefix of each k-successive tokens. In order to find the beginning and ending of a token, the dictionary also has 5 arrays:
        - `byte` Prefix sizes
        - `byte` Tokens length
        - `int` Tokens Frequency, where frequenct is calculated as the total number of appearances (including duplicates) of a token.
        - `int` Pointers to the starting location of tokens in the concatenated `String`. We only keep a pointer for 1 in k tokens.
        - `long` For each token we hold a pointer to the beginning of it's corresponding posting list (which is saved in another file on disk).
    - ProductID dictionary: Implemented the same way as above, except that in this case we hold product IDs of the reviews in the file, instead of tokens. Frequency now represents the number of reviews that refer a given product ID.

    Each dictionary has a corresponding file holding it's elements' (token or product ID) posting lists. Those files encode for each element's posting list, using a group varint encoding, a sequence of numbers representing the review IDs where the element can found. Due to the fact that review IDs are in an ascending order, we encode them as the gap difference rather than the actual ID. As for the tokens posting list, we also encode for each token (using a group varint encoding) a corresponding list of frequencies of the token in each review ID that it appears in.

    Moving on, our third structure that holds the review meta-data is implemented using 4 arrays and a `String`. The arrays are of a size equals to the number of reviews holding:

    - `byte` for review scores.
    - `short` for each of numerator and denominator of review helpfulness.
    - `short` for the number of tokens in each review.

    The `String` is a concatenation of the product IDs of all reviews in ascending order of review ID, so instead of holding an array of product IDs ( `String` s) we only hold one `String` and minimized the memory overhead of `String` s.



2. When creating an IndexReader, we load into main memory both dictionaries and the review meta-data, but keep storing the posting lists for both dictionaries on disk.

3. Let's denote `n = number of reviews`, we have:

   - The size of the table of review data is: $n \cdot (short + short + short + byte) = 7n$ `byte`s. The `String` of the review meta-data has exactly `char`s (assuming each product ID is composed of 10 letters). Overhead of a `String` in Java has `4B` for the pointer of the array of chars, `8B` header, `4B` offset, `4B` length of the `String`, `4B` hashcode, the `char` array is of size: `8B` header, `4B` length and  padding. Together it sums to `byte`s.

   Let's denote `t = number of tokens`, we have:

   - , and overall `byte`s for the tables.

   Let's denote `d = average size of token`, `s = average size of suffix without mutual prefix`, we have:

   - The concatenated `String` has `36B` overhead plus $\left(\left(\frac{t}{K}\right) \cdot d + \left(t - \frac{t}{K}\right) \cdot s\right) \cdot char$, and overall $36 + \left(\frac{t}{100} \cdot d + \left(t - \frac{t}{100}\right) \cdot s\right) \cdot 2$ `byte`s.

   As for the posting list file, it's very hard to give a theoretical analysis in association with the size of memory it will require, because it depends on each of the actual numbers we're encoding. Trying to be more precise, diving into the implementation of group varint encoding, the expected size of one posting list of one token in the token dictionary will be as follows: Let's denote `M = number of groups to encode` using group varint (it is the number of reviews that a given token appears in). Let's denote variables x, y, z, w = number of groups out of M that represent a number that takes 1, 2, 3, 4 bytes respectively. We get that the size of one posting list is:

   - 4 bytes for writing the number M
   - $\frac{M}{4}$ control bytes (each takes 1 `byte`)
   - $1 \cdot x + 2 \cdot y + 3 \cdot z + 4 \cdot w$ `byte`s (for the actual encoded numbers)

   Now, for each token in the token dictionary we hold 2 posting lists successively, one for encoding the gap difference of each of the reviews the token appears in, and one for their matching frequencies. for each product ID in the product dictionary we hold just one posting list, the one of the gap differences.

   Overall we can say that we expect the encoded posting lists to be much smaller than the actual size that the numbers would take without encoding, due to the fact that we encode gap differences rather than actual numbers, and the fact that group varint doesn't waste bytes of padding for a fixed length, for representation.