Web Information Retrieval (67782)
Ex2: Index Construction

Submitted by: Daniel Kerbel (CSE: danielkerbel, ID: DELETED)

1 Index creation runtime

The computer which was used has the following specifications:

- OS: Windows 10 64-bit
- CPU: Intel i5-7300 HQ at 2.5-3.5GHz
- RAM: 24GB of RAM, only allocated 1GB to the Java process
- Storage: The data-set was stored on a 7200 RPM external hard drive

The dataset is the entire 1995-2013 Amazon Dataset (the `all.txt` file) which has the following statistics:

Number of products 34,686,770

Raw size 35,781,618,251 bytes

Total amount of tokens 2,909,591,866

Number of unique tokens 413,742,857

I've also tested a few smaller datasets which are included in this file.
Index creation was implemented using the SPIMI(merge based) algorithm, and I've split the measurement into two stages:

- Time taken by the "invert" stage (including parsing), where many temporary indices are created
- Time taken to merge all indices and create the final index

Click here to see the number of tokens processed as function of time, for each data-set
Click here to see the total time as function of the total reviews.

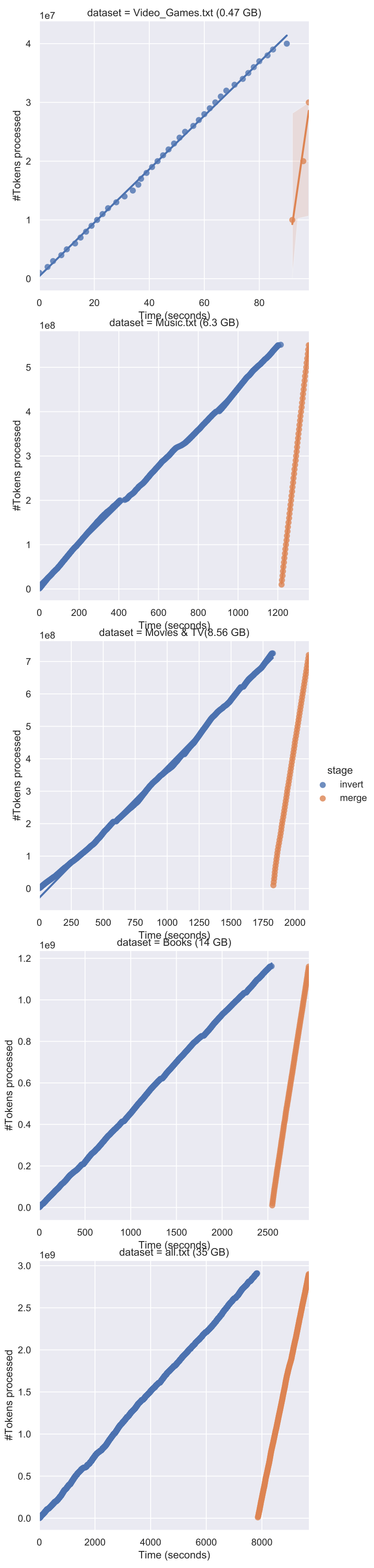
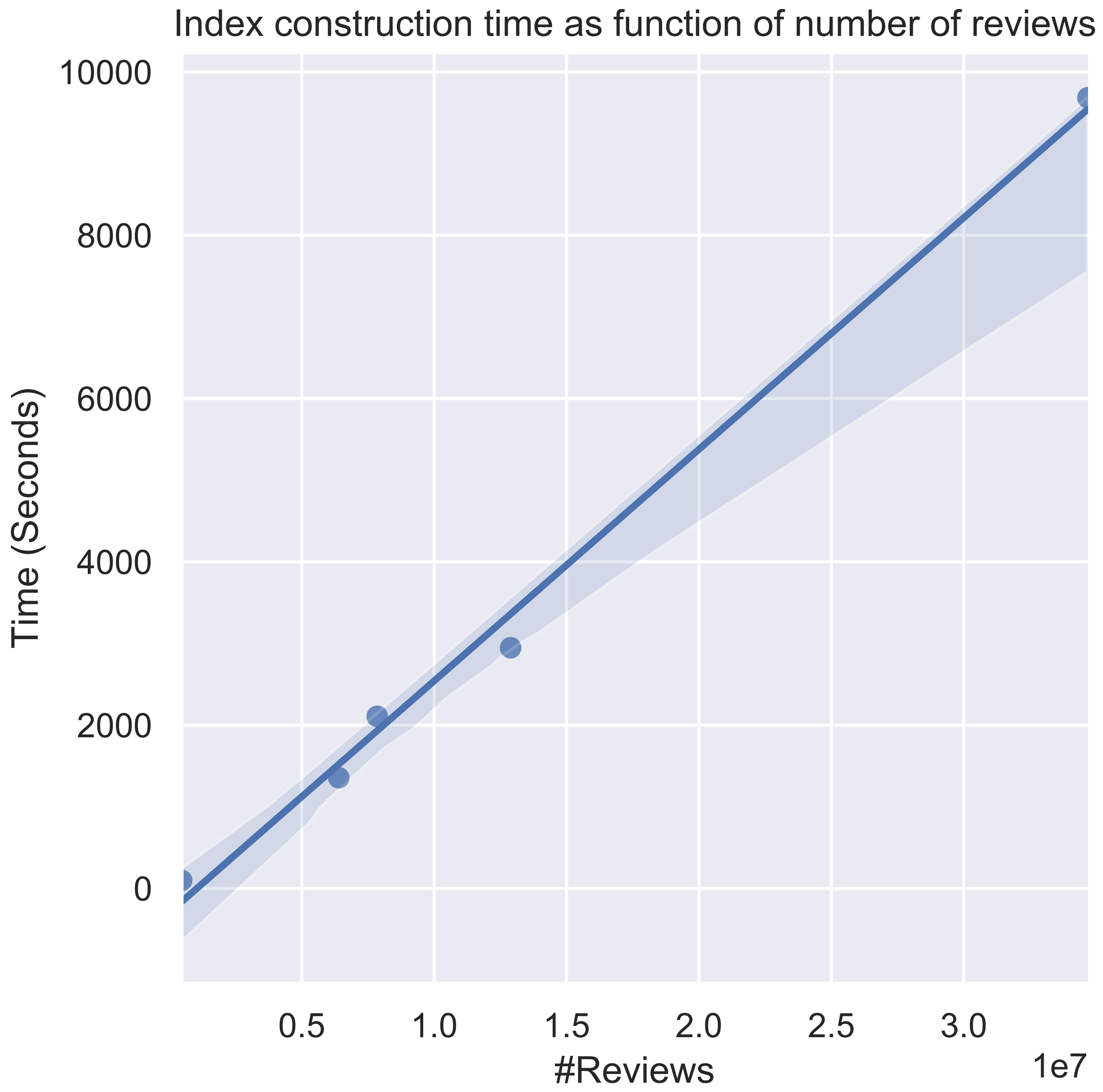


Figure 1: Cumulative index construction time at various stages, plot for each dataset. The invert stage also included IO for filling the ”compact storage” which includes reviews metadata (which is just filled sequentially as we encounter new documents, and doesn’t require any sorting or special processing). Not included is the time taken to sort productID - docID pairs, containing the lowest & highest review IDs per product. I’ve actually implemented this via an external sorting algorithm, but as it turns out, only 1 block was needed(the sort was performed entirely within memory), and finished in a couple seconds.

Figure 2: Total index construction running time



2 Disk usage

During the invert stage of **all.txt**, 15 temporary indices were created, each taking around 540 MB, and using a total of 8 GB. In addition to the original dataset of 35 GB, this means we need a total disk usage of 43 GB, which implies a 22% disk size overhead during index construction. [1]

Click here to see temporary index size(not final) for each dataset, as a function of time

Click here to see the final index size, as a function of total reviw count.

After merging, the final index takes **8.46 GB**. This means that during merging, we use a total of 16.46 GB (adding up the temporary and final indices, but omitting the input files as they aren't needed at this point)

[1]In theory, we could use something similar to 8 GB if we split the input file, and remove splits that we're processed, as we only go over the input once

Figure 3: Cumulative temporary index size(not final) over time

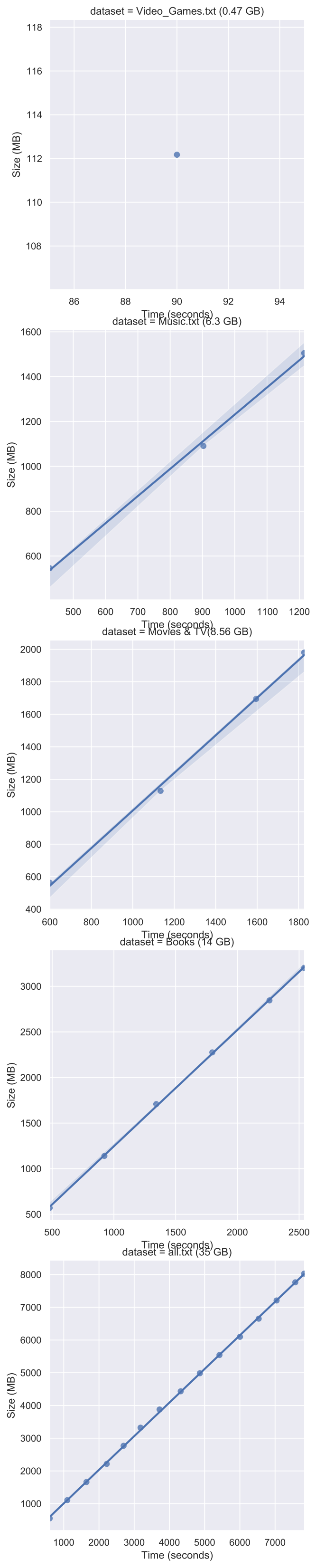
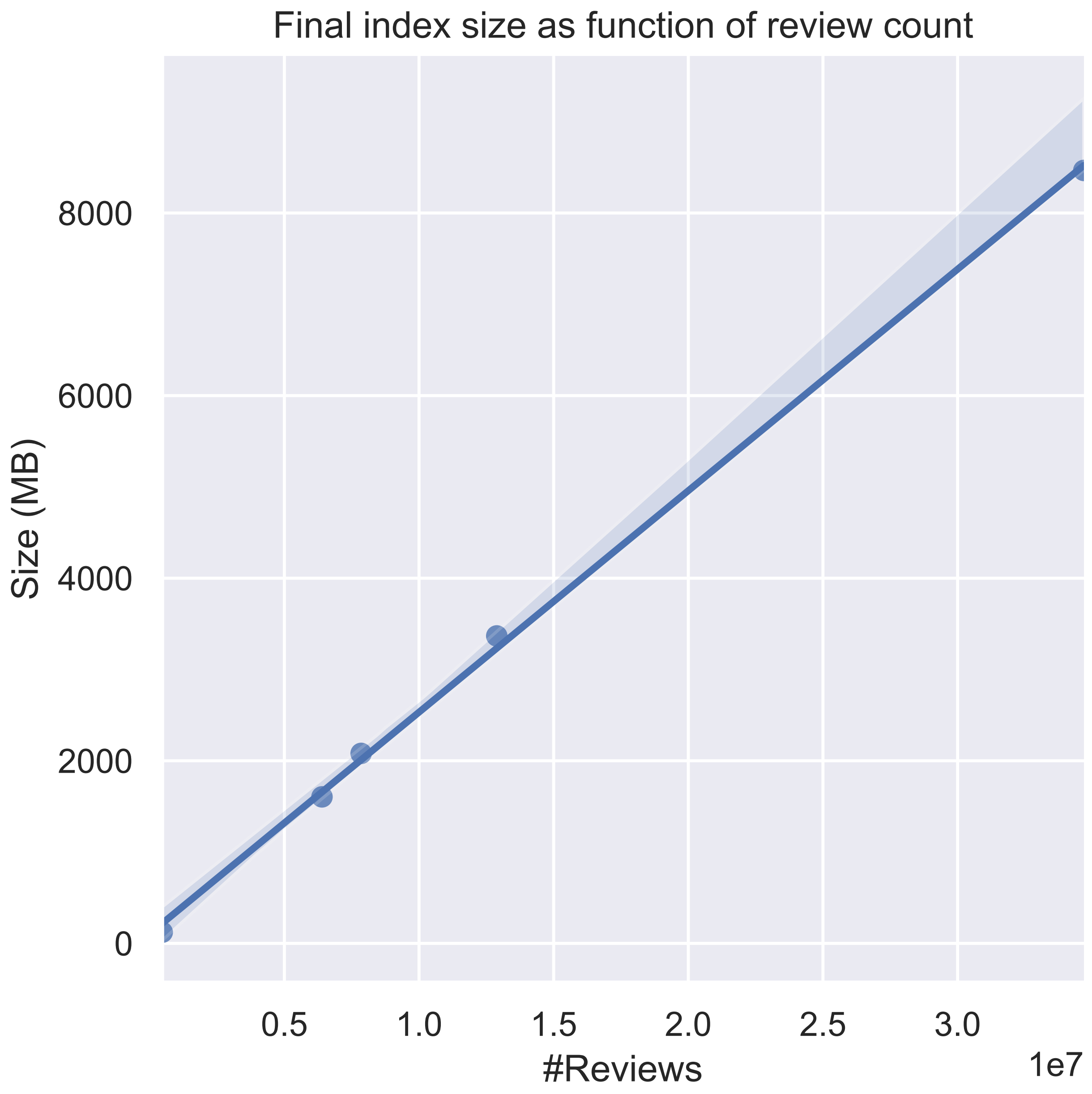


Figure 4: Final index size(not counting temporary or input files)



3 Index query operations

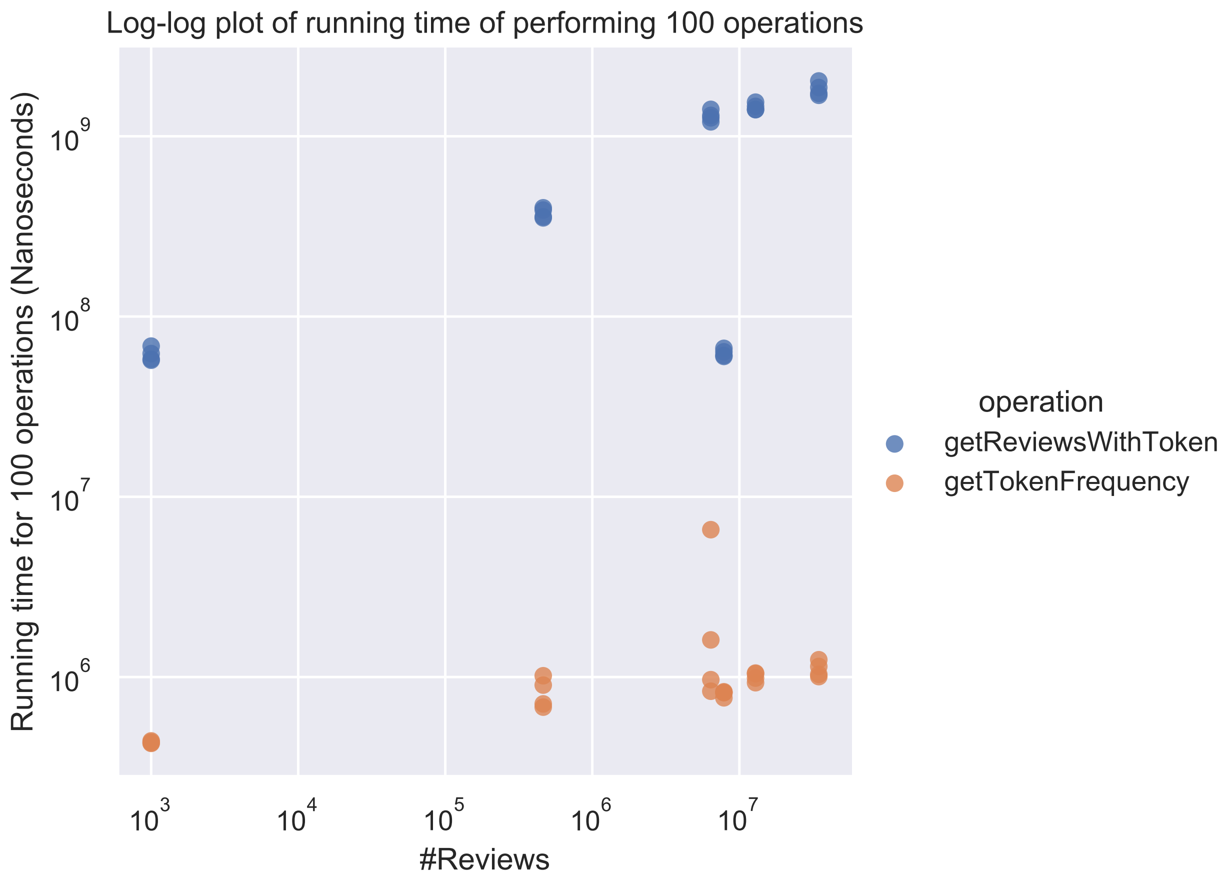
For each of the previous data-sets, I've measured the total time to perform 100 requests to `getReviewsWithToken` or `getTokenFrequency`, by using 100 terms chosen independently and uniformly from the entire set of tokens in the dataset.(Chosen for each data-set before timing began). It should be noted that I take a random subset of terms, and not tokens, so terms that appear frequently do not have a higher chance of being picked.

The resulting plot can be seen here. I've repeated the same same test a couple of times, each time with different randomly chosen tokens - this can be visualized by several points of the same color within the same vertical line(same data-set).

- `getTokenFrequency` has a logarithmic time complexity, this can be observed from the plot due to the fact that the running time increases much slower than the increase in the number of reviews, in fact the running time is nearly constant aside from an outlier. This makes sense considering it is performing binary search within main memory.
- `getReviewsWithToken`, which includes time for iterating the enumeration, is slower by about 2 orders of magnitude than the above due to the disk IO while iterating over posting lists. The trend appears linear but there's an outlier for one of the review sizes. The running time is affected by 2 factors:
  - Time taken to find the posting list pointer within the dictionary via binary search, which is logarithmic.
  - Time taken to iterate over the posting lists. On the one hand, as we increase the review size, the posting lists of common terms get longer(a linear increase in the iteration running-time), on the other hand, the vocabulary size increases, and rarer terms have shorter posting lists.

    Overall, the former case seems to be more dominant, explaining the linear trend, but one of the reviews is an outlier, which is explained by a richer vocabulary, with shorter posting lists per term.

Figure 5: Index query running times


Log-log plot of running time of performing 100 operations
Running time for 100 operations (Nanoseconds)
10^9
10^8
10^7
10^6
10^3
10^4
10^5
10^6
10^7
#Reviews
operation
getReviewsWithToken
getTokenFrequency