4.1)
Let granularity for the problem be x
N = No of posting list entries = 96 Million entries
s = Size of each entry on disk = 2 bytes

Assuming the structure of on-disk posting list as in the textbook,
No of synchronization points - N/x
We need to read all these synchronization points from disk to perform binary search on these points for the first step, the number of bytes read for this purpose will be - N*s/x

Then we need to read the particular part of the list starting with the synchronization point, s1 that is returned by the binary search and read into memory x entries starting at s1.
The number of bytes read will be - x*s

Total bytes read for a given granularity, T(x) = N*s/x + x*s

We need to optimize granularity w.r.t the total I/O operations or total bytes read. For that we need to differentiate T(x) with x and set it to 0 to find x.

$T`(x) = - N*s/(x^2) + s = 0$

Granularity x = 9797.95 = **9798** approximated to the nearest integer.

The total bytes read = T(9798) = 96M * 2 / 9798 + 9798*2 = 39191.83 bytes = **39192 bytes**

The **last line** of the postings list will contain fewer terms = 96 Million % 9798 = 8994 entries. If the target posting entry is in this line then total bytes read will be = 96M*2/9798 + 8994*2 = 37583.83 bytes = **37584 bytes**

4.3)

1. Line 17 in figure 4.12 involves sorting an in-memory dictionary. Sorting dictionary entries in requires a time complexity of N*log(N) where N is the size of the list and thus introduces a logarithmic factor.
2. In figure 4.13, inside the if-block in line 12 we add the smallest term to the postings list. To do this we get the postings list and add it to the postings list for that term, this operation again adds a log(N) constant to the complexity.
3. If the number of index partitions is large (more than 10), then the algorithm can be improved by arranging the index partitions in a priority queue (e.g., a heap), ordered according to the next term in the respective partition. This eliminates the need for the linear scan in lines 7–10 of figure 4.12 . This is mentioned in the textbook and the heap again adds an additional complexity of log(N).

Paper Summary - Inverted indexes for phrases and strings

Phrase queries in traditional inverted indexes are performed by retrieving the inverted list for each word in the phrase and then applying an intersection algorithm.
The limitation is that they do not support string documents without word separation.
A naive solution is to maintain inverted lists for all possible phrases or strings.

To better this, the paper introduces a variant of the inverted index that works for both strings and phrases by maintaining a suffix tree. For each internal node in the suffix tree, an inverted list is maintained. The list associated with a node consists of pairs of the form (document, score), where the score indicates the relevance of the document to the pattern. The paper's approach is based on the observation that not all phrases need to be indexed. The use of suffix trees and their compressed versions allows for efficient pattern matching, making the search process faster.

The dictionary stores a select set of phrases, specifically those that are frequently queried. These phrases are identified by analyzing query logs. By focusing on frequently queried phrases, the system can optimize for the most common search scenarios, ensuring rapid retrieval times.

The process of choosing which phrases to store in the dictionary is based on their frequency in the query logs. The paper employs a two-step process:

1. Identification of Candidate Phrases: The system first identifies all the phrases present in the query logs. This provides a comprehensive list of all potential phrases that users have searched for.

2. Frequency Analysis: Once the candidate phrases are identified, the system analyzes their frequency. Only those phrases that exceed a certain frequency threshold are stored in the dictionary. This ensures that the dictionary remains compact while still covering the most commonly searched phrases.

The suffix tree approach enables exact phrase search by allowing the system to quickly identify whether a given phrase exists in the dictionary. When a user submits a search query, the system checks the dictionary to see if the exact phrase is present. If it is, the system can rapidly retrieve the relevant documents.

The technical brilliance of this approach lies in the structure of the suffix tree. The tree's nodes represent individual terms, and the paths between nodes represent phrases. This structure allows for quick identification of phrases, as the system can traverse the tree

based on the terms in the search query. If the exact sequence of terms (i.e., the phrase) exists in the tree, the system knows that the phrase is in the dictionary. Furthermore, the suffix tree's hierarchical nature ensures that even if only part of the search query matches a phrase in the dictionary, the system can still provide relevant results. This is because the tree can identify the longest matching sub-phrase and retrieve documents based on that.

This approach, as detailed in the paper, offers a balanced solution to the challenge of exact phrase search. By selectively storing frequently queried phrases and leveraging the efficient structure of the suffix tree, the system can provide rapid exact phrase search capabilities without the need for a significantly larger index.