Assignment Report on

Inverted Index Representation of 100 Wikipedia articles

Adikar Bharath N V S (171IT202)

Under the Guidance of,

Dr. Sowmya Kamath S.

Department of Information Technology, NITK Surathkal

Date of Submission: 8th September 2020

in partial fulfillment for the award of the degree

of

Bachelor of Technology

In

Information Technology

At

Department of Information Technology

National Institute of Technology Karnataka, Surathkal

September 2020

Table of Contents

Chapter 1

Preprocessing techniques applied and results

1.1 Original number of tokens

First, 100 Wikipedia articles spanning multiple topics were taken, removing the References section and pictures. Each line was processed and using whitespace as a delimiter, tokens were generated.

The number of tokens turned out to be 44872.

1.2 Seperate based on punctuation

For each set of tokens corresponding to a document, using well-known sentence delimiters such as the comma, question mark, and period, tokens containing these symbols were further split.

The number of tokens at this stage is 35835.

1.3 Convert to lowercase

Since there is no linguistic difference between the upper and lower case forms of a word, all tokens were made lowercase to eliminate duplicates.

The number of tokens at this stage is 32498.

1.4 Remove stopwords

Words like or, and, thus etc. do not carry much importance towards token representation of a document as these are quite common across sentences in the English language. These

are known as stopwords and tokens corresponding to them are removed.

The number of tokens at this stage is 30620.

1.5 Remove Wikipedia-related noise

Wikipedia text generally has one or both of the following present in its articles:

- References (e.g. [1], [24])
- [citiation needed]

These are removed as well.

The number of tokens at this stage is 24018.

1.6 Remove symbols

Any more symbols that are present at this stage are attached to tokens and need to be removed, and thus is done so.

The number of tokens at this stage is 23222.

1.7 Final preprocessing

1.7.1 Stemming using NLTK's PorterStemmer

The NLTK (Natural Language Toolkit) library is a Python library that aids in NLP tasks. We use the PorterStemmer class to reduce a word to its stem by removing suffixes e.g. amazed to amaze, having to hav etc.

The number of tokens at this stage is 17363.

1.7.2 Lemmatization using NLTK's WordNetLemmatizer

The WordNetLemmatizer class is used to reduce a word to its lemma i.e. where there is no morphological difference between the words that have the same meaning.

e.g. having's stem is hav, but is treated differently than have. Both words should be reduced to the lemma hav.

The number of tokens at this stage is 21078.

Chapter 2

Storing inverted index

There are numerous ways to store tokens in the inverted index as well as the postings list associated with each token in the index. The data structures under consideration are explained below

2.1 Sorted Array

An array is a fixed-size contiguous allocation of elements that allows for quite fast access of an element at a particular index. But due to its fixed-size nature, whenever an element has to be inserted to an array at full capacity, a new array with a bigger size must be created, which is a costly operation in practice. Deletion may result in a lot of unused space which has to be cleared periodically, again by initializing a new array with the remaining elements. Since the array may not be sorted, search of a particular element happens in linear time.

Sorted arrays decreasing the time complexity of search of a particular element from linear to logarithmic, while also increasing it for insertion and deletion.

2.1.1 Storage

Space Complexity: O(N)

2.1.2 Insertion

Time Complexity: O(N)

2.1.3 Deletion

Time Complexity: O(N)

2.1.4 Search

Time Complexity: O(logN)

2.2 Sorted Linked List

To mitigate the issue caused by storing a collection of elements with an upper fixed-size, linked lists are used. In linked lists, each element is stored as a node, which contains the value of an element as well as a pointer to the next node in the collection. This ensures that insertion and deletion can happen in constant (if before the first node) or linear (if inserted somewhere within the collection). Search still takes linear time.

Sorted linked lists do not offer any time or space complexity advantages for insertion, deletion and search, but is advantageous when performing operations that involve the modification of two or more linked lists (e.g. merge two linked lists).

2.2.1 Storage

Space Complexity: O(N)

2.2.2 Insertion

Time Complexity: O(N)

2.2.3 Deletion

Time Complexity: O(N)

2.2.4 Search

Time Complexity: O(N)

2.3 Binary Search Tree

Linked Lists, while being dynamic with respect to size, still have linear time complexities with respect to insertion, deletion and search. Thus, trees are used, where a tree contains data as well as two or more pointers to next nodes, referred to as child nodes, which could again be the root, or uppermost element, of a tree. In a Binary Search tree, which is modelled after performing binary search on a sorted array, each node can have two

children, known as the left and right child. Nodes are inserted according to whether it is less than or equal to (left), or greater than (right) its ancestors.

This ensures that the average time complexity of inserting an element is O(logN), corresponding to a perfect binary tree (where each level except the last level is full), as well as deletion and search.

The worst case time complexity for the three options is still O(N), as the tree's height could be equal to the number of nodes i.e. the tree could take the shape of a linked list, offering no real benefit.

2.3.1 Storage

Space Complexity: O(N)

2.3.2 Insertion

Time Complexity: O(N)

2.3.3 Deletion

Time Complexity: O(N)

2.3.4 Search

Time Complexity: O(N)

2.4 AVL Tree

Due to the possibility of Binary Search Trees extending too far in height and resembling linked lists, some kind of reordering of the tree is to be done, so that the tree can be as perfectly balanced as possible. This reordering, known as balancing, is one of the driving factors behind an AVL Tree.

In an AVL tree, a constraint is applied on each node, such that the absolute difference between the heights of the left and right child is not more than 1. Insertion happens as normal, but if the the absolute difference exceeds 1, the subtree is rebalanced to ensure logarithmic insertion, deletion and search for future nodes. The same process is followed during deletion. The downside is that this rebalancing can happen for each ancestor of the node that is inserted, which doesn't increase the time complexity, but does increase the actual time taken.

Thus, the worst case time complexity of search is now reduced to logarithmic, offering a great performance benefit for systems where search operations are prioritized over insertion and deletion.

2.4.1 Storage

Space Complexity: O(N)

2.4.2 Insertion

Time Complexity: O(logN)

2.4.3 Deletion

Time Complexity: O(logN)

2.4.4 Search

Time Complexity: O(logN)

2.5 Red-Black Tree

In an AVL tree, the insertion and deletion of nodes can be quite costly due to the number of times rebalancing of a tree can happen. Hence, to ensure that rebalancing happens in constant time, a new kind of tree, known as a 2-4 Tree, is used. The binary tree version of a 2-4 Tree is known as a Red-Black Tree.

A Red-Black Tree has several constraints on insertion and deletion, reducing the actual time taken for the two operations, while increasing the actual time taken to search for an element (with time complexity remaining the same).

2.5.1 Storage

Space Complexity: O(N)

2.5.2 Insertion

Time Complexity: O(logN)

2.5.3 Deletion

Time Complexity: O(logN)

2.5.4 Search

Time Complexity: O(logN)

2.6 Hash Table

A hash table is a collection of key-value pairs, with keys being an identifier of an element, and the value being the data of an element, which can be a simple value or a collection of elements. The keys are hashed to an integer before insertion, and various hashing techniques exist to reduce the number of collisions i.e. two keys having the same hash value. The hash table stores elements corresponding to a hash value of a key in a collection, which can be one of the above mentioned data structures, resulting in almost near constant, or amortized access time.

There is also the concept of rehashing, wherein if one hash value has too many elements under it, then the table is rebalanced to properly distribute the elements.

2.6.1 Storage

Space Complexity: O(N)

2.6.2 Insertion

Time Complexity: Amortized O(N)

2.6.3 Deletion

Time Complexity: Amortized O(N)

2.6.4 Search

Time Complexity: Amortized O(N)

2.7 Choosing the right data structure

For the best possible performance across all parameters, the following choices are made:

- To store the tokens along with their document indices, a hash table is used.
- To store the document indices corresponding to a token, a red-black tree is used.

Python3 implements the hash table in the form of the dictionary as well as the set class, and thus both are used in the implementation.

Chapter 3

Sample operations

3.1 term1 AND term2 AND term3

The worst case time complexity is 2*O(min(N1,N2,N3)), where N1 is the number of document indices corresponding to term 1, N2 for term 2, and N3 for term 3. The worst case happens when there are no common indices between the three, thus their intersection yields no elements.

3.2 term1 OR term2 AND NOT term3

The worst case time complexity is O((N1+N2)*N3), where N1 is the number of document indices corresponding to term 1, N2 for term 2, and N3 for term 3.