International Journal of Scientific Research and Engineering Development-– Volume 4 Issue 3, May -June 2021

Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Directory Simulation Using In-memory Tries

Sandhya S*, Sparsh G Sarode**

*(Computer Science and Engineering, R.V. College of Engineering, Bangalore, India
Email: sandhya.sampangi@rvce.edu.in)

** (Computer Science and Engineering, R.V. College of Engineering, Bangalore, India
Email: sarodesparsh@gmail.com)

----------**********----------

Abstract:

Many applications depend on file directory simulation to filter out files/directories before they are persisted in a storage system. This paper proposes an efficient way of using in-memory Trie data structure to deal with this problem and gives an upper bound to some of the operations that are performed on this data structure. In systems where moving/traversing large numbers of files using the underlying OS is costly, we can simulate it using in-memory Tries. Applications of this are not limited to just caching but also can be extended to virtual directories that provide an in-memory interface for other applications. Another application could be fast file searching through the directories.

In the following sections, we start with an introduction to Tries, giving algorithms and time complexity for some of the operations on Tries. The later sections redefine Tries for the specific application and finally gives an analysis of the obtained results.

Keywords **—Tries, Data Structures, Trees, File system, In-memory, Virtual directories**

----------**********----------

I. INTRODUCTION

Several computing applications involve the management of large sets of strings. For example, a huge amount of data are stored as text documents, starting from news archives, law collections, and business reports to repositories of documents gathered online. These collections involve many millions of words and this number keeps growing more or less linearly with collections size. Other applications also involve managing huge numbers of strings such as bibliographic databases and genomic data indexes. Tries find their way into most of these applications.

One another application of Tries includes simulating the directory. The nature of its recursive structure is very similar to the Tries structure, which brings us to the structure of the Trie. A Trie is a data structure that is used to store and retrieve information. The nodes in a Trie store the characters of a string. By arranging these nodes to form a tree(Trie), the information can be retrieved from it by traversing the tree. In 1959, the Tries were considered in computing for the very first time by Rene de la Briandais.

Though initially tries were mainly used on character strings, they are not limited to the character strings. In our case, we can key it with the individual segment of the file path (Eg: “to”, ”path” in “path/to/file/my.txt”).

A. Tries

Trie, denoted by ***t***, is a tree, in particular, it is a rooted tree. Let’s denote the root node of the tree as ***r***. Every node of the Trie has ***n*** pointer to different nodes. ***n*** depends on the application of the trie, for example, ***n*** is 26 (number of English alphabets) if trie is used to store English words. Every pointer is initialized to ***null***. In addition to pointers, every node of the Trie logically stores a ***character***. A ***character*** is the smallest atomic unit of the ***string*** to be stored in the Trie. The solid

definition of ***character*** and ***string*** depends on the application.

For the sake of describing the following algorithms, we will assume a few things about a ***node*** in the Trie. A ***node*** consists of three attributes:

- ***character*** – The ***character*** that the node stores
- ***map*** – A map that maps ***character*** to ***node***
- ***is_end*** – A boolean value that denotes whether the node represents an end of a string

Note that this is just a general description of a ***node*** and it is subject to change for different applications.

1) Insert Operation Algorithm:

The insert operation takes a ***string*** of ***characters*** and stores it in the Trie. The time complexity of this algorithm is ***O(c)*** where ***c*** is the number of ***characters*** in the ***string.***

```
function insert(T, string):
      var T // Trie
      var string // string to be
                 // inserted into the trie
      var ch // character in string
      var t // node in Trie

      t := root(T)
      for ch in string:
            if (t.at(ch) == null):
                  t.at(ch) = new_node(ch)
            t = t.at(ch)
      t.is_end = true
```

Fig. 1 Insert operation algorithm

2) Search operation algorithm

The search operation takes a ***string*** of ***characters*** and searches it in the Trie. The time complexity of this algorithm is ***O(c)*** where ***c*** is the number of ***characters*** in the ***string.***

```
function search(T, string):
      var T // Trie
      var string // string to be
                 // searched in the trie
      var ch // character in string
      var t // node in Trie

      t := root(T)
      for ch in string:
            if (t.at(ch) == null):
                  return false
            t = t.at(ch)

      return t.is_end
```

Fig. 2 Search operation algorithm

The logic of the above two algorithms can be used for other functions like ***has_prefix()*** and many more.

The space complexity of the Trie depends on the number of common prefixes across all the strings that are stored in the Trie. Best case scenario: ***O(length of the longest string)***. Worst case scenario: ***O(sum of the length of all the strings)***.

II. METHODOLOGY

In the previous section, we formally defined a Trie and discussed different operations that can be performed on Tries and their time complexity. In this section, we will look into a specific application of Tries which is directory simulation.

A. Directory Structure

Let's start by exploring real directories implemented by any Linux OS. Directories have a recursive structure. Every directory has zero or more ***files*** and zero or more ***subdirectories***. Every Linux file system has a root directory. Both subdirectory and file names can be changed by traversing to the particular directory from the root directory. Every file in the file system can be represented in the form of a complete file path like ***"/user/john/desktop/words.txt"***. Here ***"/"*** is an OS-dependent file separator. For the sake of simplicity,

we will restrict ourselves to the Linux file system. This can be easily extended for Windows as well.

B. Definitions

We shall now see more specific definitions of ***characters*** and ***strings*** for our application.

- ***character*** – A ***character*** is either a file name with or without extension (Eg: words.txt) or a directory name without extension (Eg: desktop). Hence, there are two types of ***characters***: ***file*** and ***directory***.
- ***string*** - A ***string*** is a sequence of ***characters*** separated by OS file separator ***"/"*** with constraints being that only the last ***character*** may be ***file*** type and that all ***strings*** start with OS file separator. Examples of valid ***strings:*** "/tmp/words.txt", "/etc/hosts.txt", "/usr", "/"

Every ***node*** in Trie will have the following attributes/properties:

- ***files*** – A Set of ***file*** type ***characters***
- ***subdirectories*** – A Map that maps **directory** type **characters** to **nodes**

Note that in this specific Trie application, the ***is_end*** attribute, as defined earlier, is not required and the ***character*** attribute (as defined in the last section) is replaced by ***files***.

C. Operations on Trie

1) Insert Operation

The insertion algorithm remains the same as described in the previous section. The only difference is that we can no longer assume that maps take constant time to retrieve since they use strings as keys(instead of English characters). If we include the time taken by maps, the final time complexity will be ***O(c * log(l))*** where ***l*** is the average length of the ***character***.

2) Search Operation

The search algorithm also remains the same as described in the previous section. Again since we can no longer assume that maps take constant time, the time complexity will be ***O(c * log(l)).***

3) Depth First Search Operation

Once the Trie is built, there will be a requirement to traverse the Trie for further analysis. Depth First Search algorithm can be used to traverse the Trie in such cases. The time complexity of this algorithm is ***O(n + log(L))*** where ***n*** is the number of nodes in the Tries and ***L*** is the total number of entries in the map across all nodes.

```
function dfs(T):

      // Analysis on files

      for subdirectory in T.subdirectories:

            // Analysis on subdirectories

            // recursive traversal
            dfs(T.subdirectories[subdirectory])
```

Fig. 3 DFS operation

4) Merge Operation

There might be a requirement to merge two pre-built Tries to form a new Trie in some applications. Merge operation provides a recursive solution for that. The time complexity of this algorithm is ***O(n + log(L))*** where ***n*** is the number of nodes in the Tries and ***L***is the total number of entries in the map across all nodes.

```
function merge(T, t):
     var T // Main trie node
     var t // Trie node to be merged

     for file in t.files:
          T.files.add(file)

     for subdirectory in t.subdirectories:
          if(T.subdirectories.has(subdirectory):
               merge(T.subdirectories[subdirectory],
                    t.subdirectories[subdirectory])
          else:
               T.subdirectories[subdirectory] =
                    t.subdirectories[subdirectory])
```

Fig. 4 Merge Operation

III. RESULTS AND PERFORMANCE EVALUATION

This approach has been successfully implemented and tested against some of the Linux commands such as 'find' and 'locate'. 'find' is a command-line tool that searches for files in the file system in UNIX-like OSs. It does so by fetching file details from the underlying file systems as we may expect. So, the in-memory approach that's proposed here theoretically should be much faster when compared to using 'find'. On the other hand, 'locate' uses a database to search through the files in the system. But this requires it to be updated frequently as and when there are file changes in the file system. Theoretically, the in-memory approach should be at least as fast as using 'locate'

A) Experimental Evaluation

In order to test the performance, 2,00,000 files were created on Debian-based Linux OS and the 'find' command was used to search for a particular file. This took 35 mins to fetch results. The in-memory Tries approach could fetch the same result in 1.34s as expected. However, the Tries approach took 10 minutes to build the Trie from the file system which is still better than using the 'find' command. The 'locate' command took 15 minutes to update the database and 1 minute to fetch the results.

IV. CONCLUSION

In this paper, a new in-memory approach using Tries has been proposed to simulate a file directory. The primary motivation of this paper is to make file searching operations faster than other available methods, but the applications of this are not limited to file searching. Other applications primarily involve the cloud where there could be conditional migration of data based on types of files stored in different storage nodes. In such situations merging all the files in a separate storage node just to validate the condition will increase the cloud cost. The proposed solution finds its way into solving such problems without actually having to move the files to a separate storage node.

However there are trade-offs for this approach. One such limitation is the initial overhead of constructing Tries from the file system. This limitation is evident and cannot be overcome since the files stored in the file system have to be visited at least once. Another limitation is that additional usage of memory might result in increased cloud costs. However, to tackle this problem, caching techniques such as LRU caching can be used.

This paper also provides a few implementation details which can be incorporated into one's specific application easily. The performance analysis of this approach has been done and found to be more efficient than some of the existing Linux tools such as 'find' and 'locate' both of which use different approaches to find a file in the file system.

ACKNOWLEDGMENT

I would like to thank my college, R.V. College of Engineering and Computer Science and Engineering Department for giving me an opportunity to conduct this extensive research. I also thank my professor Dr. Sandhya S for providing valuable guidance during the research.

REFERENCES

[1] Heinz, Steffen & Zobel, Justin & Williams, Hugh. (2002). Burst Tries: A Fast, Efficient Data Structure for String Keys. ACM Trans. Inf. Syst.. 20. 192-223. 10.1145/506309.506312.

[2] Ribeiro, Pedro & Silva, Fernando. (2010). G-tries: An efficient data structure for discovering network motifs. Proceedings of the ACM Symposium on Applied Computing. 1559-1566. 10.1145/1774088.1774422.

[3] R. Salunkhe, A. D. Kadam, N. Jayakumar and D. Thakore, "In search of a scalable file system state-of-the-art file systems review and map view of new Scalable File system,"*2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, pp. 364-371, doi: 10.1109/ICEEOT.2016.7755371.

[4] C. Ghasemi, H. Yousefi, K. G. Shin and B. Zhang, "A Fast and Memory-Efficient Trie Structure for Name-Based Packet Forwarding,"*2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 302-312, doi: 10.1109/ICNP.2018.00046.

[5] J. Park, J. Park and J. Choi, "Web-Based Document Classification Using a Trie-Based Index Structure,"*2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops*, 2007, pp. 52-55, doi: 10.1109/WI-IATW.2007.70.

[6] W. Lu and S. Sahni, "Packet Classification Using Space-Efficient Pipelined Multibit Tries," in *IEEE Transactions on Computers*, vol. 57, no. 5, pp. 591-605, May 2008, doi: 10.1109/TC.2007.70846.

[7] J. Tan, X. Liu, Y. Liu and P. Liu, "Speeding up pattern matching by optimal partial string extraction,"*2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2011, pp. 1030-1035, doi: 10.1109/INFCOMW.2011.5928778.

