No-Escape Search: Design and Implementation of Cloud Based Directory Content Search

Harshit Gujral, Abhinav Sharma, Sangeeta Mittal
Department of Computer Science & Engineering
Jaypee Institute of Information Technology
Noida, UP, India
harshitgujral12@gmail.com, sharma1997abhinav@gmail.com, sangeeta.mittal@jiit.ac.in

***Abstract*—Searching in-file content is a crucial task in everyday computing, made particularly difficult due to lack of efficient in-file content search system in windows operating system. The goal of this paper is to present a cloud-based exhaustive in-file content search algorithm using three dimensional hash data structure. To facilitate an abundant size of hash and to save user's memory space we have used cloud for accommodating hash structure. We aim at presenting the user with required retrieval in the time complexity of O (constant), usually within 3-8 seconds. Our retrieval is a comprehensible combination of user's input string, the filename(s) containing it. Also, unlike windows operating system's search system, it permits to include location(s) and even multiple occurrences of user defined string inside a file.**

Keywords: cloud; content search; firebase; hash; window search

I. INTRODUCTION

Most information retrieval technologies designed to facilitate in-file content search are slow and inefficient. File indexing plays a big role in the fast retrieval of contents. When a user searches for some folder or file which is not indexed, the search is slower by several orders of magnitude. Windows search can optionally build a full-text index of all files on a computer [1]. It runs as a service that indexes file names, contents and properties of file system for only few directories by default. Figure 1 illustrates such default indexing on a Windows 10 system. The indexing can be expanded by using "Modify" option and adding folders to be indexed. The indexing process itself is, however, too time consuming and index tables occupy significant memory space. Full indexing of a directory of even up to 10GB can take several hours to index [2]. Also, Windows search doesn't provide the exact location and multiple occurrences of the searched terms, if any.

In order to tackle current difficulties and facilitate the user with a more efficient and quick in-file search experience, we have developed and described a system named "No-escape Search" in this work that makes searching easy for the user. Our system has solved three major problems of the Windows indexing method. These are: memory wastage by Windows indexing, slow data retrieval and the inability to facilitate the user with location(s) of the input string inside the file.

The system comprises of an automated scanning algorithm that is invoked at the time of insertion of files in the target directory that is to be scanned. After it detects any new insertion in the directory, it further invokes a file scanning and uploading algorithm that presently scans files with txt, pdf and csv extension, and uploads words greater than four characters to cloud space along with their position and filename in data structure proposed for storing indexing information.

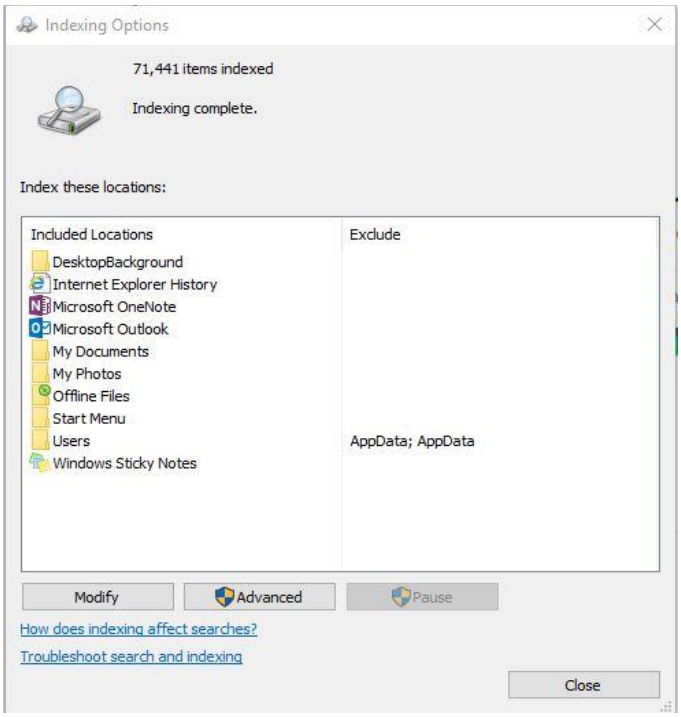

Fig. 1. Default indexed items by windows.

We have designed a three dimensional hash data structure to achieve our objective of fast data retrieval along with position of searched term in the documents. This has been evident by the time required to index same directories by windows 10 vs our system. To solve another problem of main memory wastage, Google Firebase has been used as cloud to store our hash structure. Like Windows, the indexing can be paused at any time and restarted later, when the system is free and resources like Internet are available.

The details of proposed system and its implementation are organized in rest of the paper as four sections. Section II discusses related work done in this field. In Section III, detailed methodology of the proposed algorithm has been described

along with an example. Section IV explains the results of comparison of windows and the proposed method for searching in terms of search time as the function of directory sizes. The paper is concluded in the Section V.

II. RELATED WORK

Many research works have addressed design and implementation of search term retrieval systems in files and directories.

A document indexing technique for logical to physical mapping using hash tables was first proposed in [3]. They have considered an exhaustive list of search terms and represented them as bit vector of presence or absence in a particular document. This system proved to be faster than existing the hash table size increases exponentially with respect to number of unique terms.

Use of extendible hashing for implementation of structural data records has been demonstrated in [4]. The main drawbacks of this work are that the hash table has to be cached in primary memory by the OS. This is not desirable, as the number of documents may tend to become too large in today's scenario.

Authors in [5] propose a method of peer to peer search instead of a central server using distributed hash tables and hierarchical summaries of files available with one of the participants. High dimension points represent occurrence of each document. Time required for building summaries is O (n x D2), where N and D are the number of vector and the dimensionality of vector. The solution is cost effective for distributed content. However, same performance enhancement is not expected when the method is applied on local data.

Problem of maximal subset of keyword search has been addressed in this paper [6]. We have currently focused on all words being present in the documents and not the partial search.

There are other works that has enhanced the searches by applying encryption based security. HMAC based storage and retrieval described in paper [7] is an example of it.

Some recent searching techniques have proposed complex machine learning to include dimensions other than search terms like proximity and term dependencies within document for more relevant search results [8].

Positional information of searched terms in each document has not been addressed in any of these works. Moreover, our work had the objective of improving live searches in contemporary windows operating systems and by our solutions we were successful in improving the search time and capability of each search.

III. METHODOLOGY

Our system comprises of three major algorithms: first, automated directory scanning algorithm; second, file scanning and uploading algorithm unique for pdf, csv and txt files; third, retrieval algorithm.

A. Automated Directory Scanning Algorithm

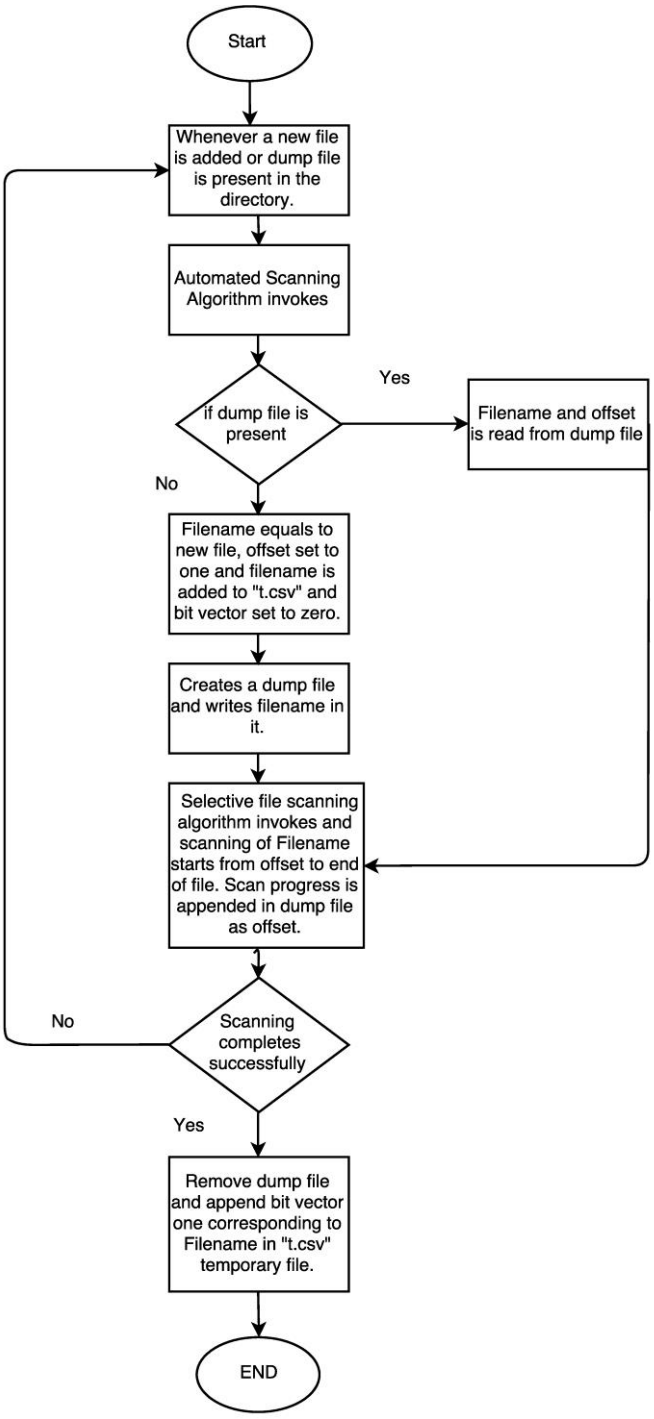

Fig. 2. Flowchart of Automated Scanning Algorithm

This library mainly provides three approaches to do the same: first, poll the directory every few seconds and compare the changes; second, the use of Win32's 'find first change notification' API; and third, use of Win32's 'read directory changes' API. For the purpose of our algorithm, we have used 'Find first change notification' API, exposed via the pywin32 file module because it only notifies when directory actually changes, so no polling is required.

It involves use of a 'wait for single object' call from pywin32 events which gets invoked when something is changed in our target directory. We have created a temporary file named 't.csv' inside the directory and used bit vector approach to manage the records of the newly added and already scanned files by allotting zero bit to newly added files and one to already scanned files. Whenever a new file is inserted in the directory, 'wait for single object' call invokes the automated scanning algorithm. It consists of three steps: first, new filename along with zero bit is added to the file 't.csv' and scanning and uploading of the file commences; second, to tackle errors like internet connection loss, a dump file is created in the directory that stores the filename. Moreover, the current scan progress is continuously appended in the dump file. The third step is that when the scanning and uploading of file completes then the dump file is automatically removed from the directory and bit vector corresponding to new file name is changed to one. Similar process works for multiple file insertion. Figure 2 shows flowchart of automated scanning algorithm. Similarly, this scanning algorithm also invokes if dump file is present in the directory.

Automated directory scanning algorithm results in invoking of selective file scanning and uploading algorithm based upon newly inserted file extension.

B. Automated File Scanning and Uploading Algorithm

Presently, this algorithm is scanning and uploading files with pdf, csv and txt extensions in our target directory. This is selective in nature as different approach is needed to scan these files.

We have scanned all the files with .pdf extension with python library known as PDFMiner.six [10]. We have scanned pdf documents from an offset page to the last page of the document. By default offset is page one, else it is defined by dump file in the directory (if any process is pending). Whenever a pdf file is inserted inside the target directory, scanning and uploading of all words greater than four characters commences. Along with the words we store their absolute page number at +x axis or third dimension of hash structure.

We have scanned CSV files using csv library of python and linearly scanned text files and stored words greater than four characters along with their row numbers and line number at cloud respectively.

C. Retrieval

Cloud is chosen for uploading hash data mainly because the time complexity of retrieval from hash data structure is O(1) while creation of hash takes O(n) time. Here, we define a ready state i.e. a state when the hash is created and ready to process retrieval. Suppose, the hash is stored in a file (say text file) and prior to every query system will process hash from text file to ready state. The major drawback is that It will take O(n) time before each query; second, Suppose if we keep hash always in the ready state then the major drawback is that In local machine (here user's computer) data structure is created and stored in RAM which is valuable and limited. It will continuously occupy RAM even if the frequency of querying is very low (say zero).

So to address these issues, we have stored hash data structure in the cloud. In Cloud i.e. Firebase, is basically a NoSQL database where hash data structure is always stored in the ready state. By this, the system saves valuable user's RAM and regardless of the frequency of querying facilitates user with the required result in O (constant) time.

The constraint of user's input is that keyword should be greater can four characters, This is done to focus search results mainly on jargons and not on words like "a", "the", "each" etc. As the user hits search button, retrieval algorithm calculates its index by multiplying ASCII value of first five characters of keyword to 26^ (5-i), where, i is position of character ranging from 1 to 5 and substituent maps this index to first dimension or +z axis and retrieves all of the data in second and third dimension in JSON format. We parse JSON and provide user with comprehensive combination of search results, the filename and the position of keyword inside corresponding file. Figure 3 shows data flow diagram of retrieval. If multiple queries are searched then a Thread-Pool will be created to utilize available resources and provide user with the required result in least time.

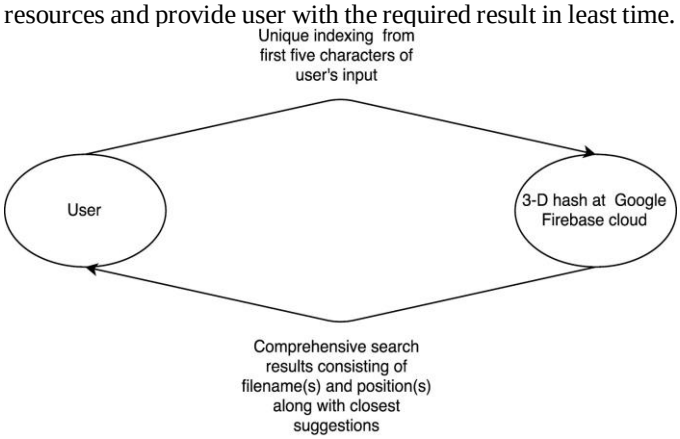

Fig. 3. Data flow diagram of Retrieval

D. Example

Figure 4 shows the 3D representation of our hash structure. The first dimension of the hash table consists of indexes on the +z-axis. The count of indexes can reach a maximum up to 26^5 (around 11 million). There are a total of 26^5 possible combinations ranging from "aaaaa" to "zzzzz". Suppose the hash only contains "aaaaa" (lowest cap) and "zzzzz" (highest cap), then Firebase will not store all 26^5 indexes, it will store only 2 indexes in hierarchical key-value pairs. Unique indexes are generated by multiplying the ASCII values of first five characters of a word to 26^ (5-i), where i is position of character ranging from 1 to 5. Second dimension of hash structure consists of specific words with similar first five characters. Example, word "create", "creates", "created" and "creating" will be mapped to first level indexing at 47315202.

ASCII (c) * 26 ^ (5-1) + ASCII (r) * 26 ^ (5-2) +

ASCII (e) * 26 ^ (5-3) + ASCII (a) * 26 ^ (5-4) +

ASCII (t) * 26 ^ (5-5)

99*456976+114*17576+101*676+97*26+116=47315202

All the words that will map to 47315202 will be stored at second level indexing of the hash structure for retrieval purposes. Second dimension is plotted on the +y axis. Third dimension of hash data structure is plotted on the +x axis and it stores filename(s) and position(s) of the word.

When the user will provide the search keyword, retrieval function will calculate +z axis indexing by multiplying ASCII value of first five characters of keyword to 26^ (5-i), where, i is position of character ranging from 1 to 5 and will retrieve all the information in +y and corresponding +x axis. All the words in +y axis will serve as the comprehensive suggestions to the input keyword along with their positions in the corresponding files. Figure 6 shows example of such retrieval.

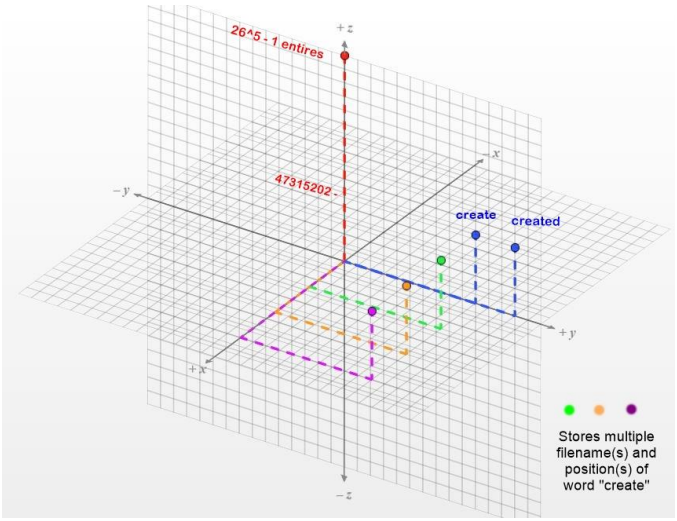

Fig. 4. A 3D Plot of used Hash Data Structure

Figure 6 shows output generated of keyword “create” mapped at index 47315202 at +z axis. It displays create in various filenames at various positions (example Page(s) 7, 7 means that keyword is present twice at page 7). Moreover we are supplying user with nearby suggestions to enhance our search (like Google). Here, our system also shows the result of “creates” and “created” along with “create”. The retrieval time majorly depends upon internet connection as data is directly fetched from cloud i.e. firebase. In most of the cases it’s between 2-5 seconds.

Figure 5 shows the sample hash at “47315202” location and data at second and third dimensional. To facilitate Firebase’s feature that data inside similar fields get overridden, third dimension has a unique attribute key. Figure 3 describes data flow diagram of retrieval algorithm.

If the user inputs a sentence then our algorithm presents the intersection of result of all the words in the respective sentence. Figure 7 shows output of searching a phrase “certain condition”. All sentences where both words occur in any order are reported as search result. There is a scope of finding results against partial word searches by providing a room for spell check by hitting spell check API (like Google does). For Example, the occurrence of "reate" in "create".

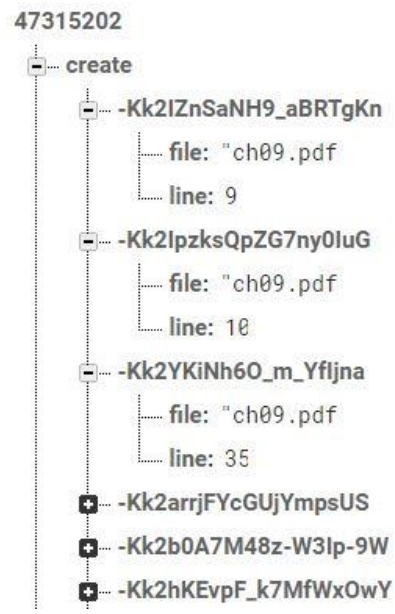

Fig. 5. Part of hash at 47315202 index in JSON format over cloud

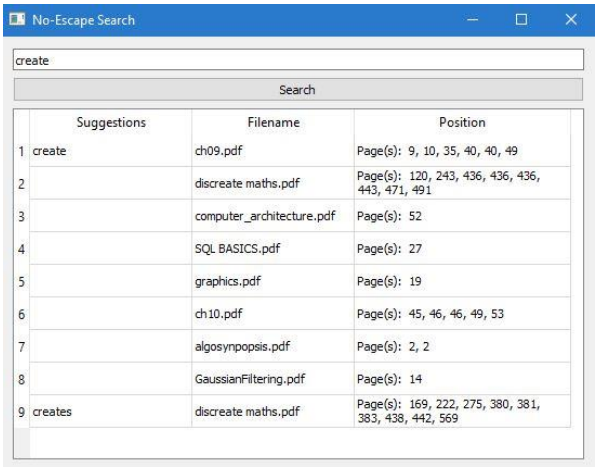

Fig. 6. Shows UI of “No-Escape Search” system

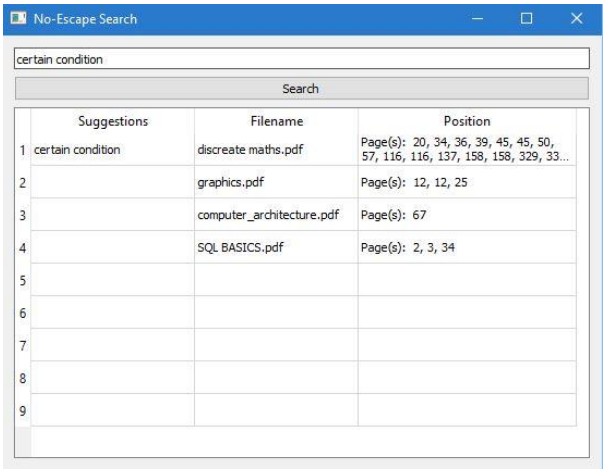

Fig.7. Output of Multi-Term Search

IV. RESULTS

In order to compare searching efficiency of window search and no-escape search test are performed in three sets: first, analysis of index creation in both searches, second , analysis of retrieval in indexed window search and no-escape search and third, analysis of retrieval in un-indexed window search and no-escape search on two different datasets .

There are two sets of data: first, directories i.e. D1: 0.4 GB, D2: 3.94 GB, D3: 19.9 GB and D4: 96.6 GB, second: files i.e. consisting F1:1000 Queries, F2:10000 Queries, F3:100000 Queries and F4:500000 Queries (here a Query can be a string or integer with length greater than four). First data set will provide the insight of finding a string in directories of various size while second set will provide the insight on querying rate/speed in fetching results corresponding to multiple searches.

The search results are observed on Windows-10 machine with 2 GB RAM, 500 GB Hard Disk Drive and i3 2.27 GHz Intel processor.

A. Analysis of Index Creation/Insertion Time

The index creation time is observed for window indexing search and no-escape search on second dataset, given queries are indexed and results are plotted in figure 8 where NES refers to No-escape search and Windows refers to Window Search. In windows indexing process largely depends on the RAM while no-escape search is internet dependent. By observing the results mean of number of queries per second can be calculated i.e. 1399 /s for windows index search and 1819.5/s for no-escape search. These results are observed on average upload speed of 4 MBps.

The data compression ratio i.e. ratio between the uncompressed size and compressed size (here between uncompressed original file and compressed indexed file) for windows is approximately 2.78 while data compression in no-escape search is approximately 1.52. For Example, Window index of a 25.04 MB file will occupy approximately 9 MB of user's disk storage while it will occupy 16.47 MB of cloud storage.

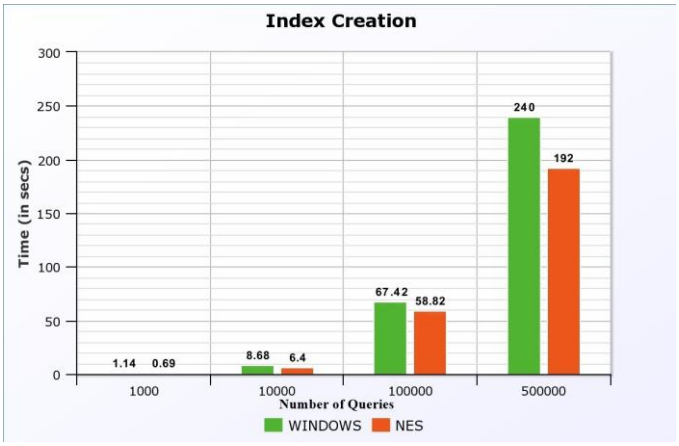

Fig.8. .Index creation time of multiple queries

B. Analysis of retrival on first data set

As window search is an indexed powered search, while searching a keyword/string through each directory D1, D2, D3 and D4 mostly few results displayed within 5-10 seconds because they were already indexed by the windows. After that time increased by several order of magnitude. Window indexing is by default selective and wastes much of the system space. Moreover, window search results do not comprises of position(s) and occurrence(s) of the word nor it gives nearby word suggestion. As shown in figure 9, For previously un-indexed directories - D1, D2, D3 and D4 searching took 3 minutes, 17 minutes, 35 minutes and 58 minutes respectively. Although window search system is further less efficient in old window versions before Window 7 [6].

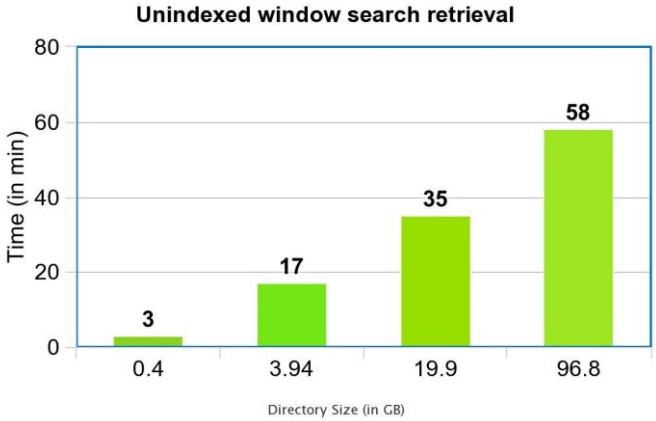

Fig. 9. Search Times in Windows Search in Directories of different sizes

While, "no-escape search" uses hash data structure. The virtue of linear-hash data structure is data retrieval with time complexity of O (1) and retrieval time is independent of size of hash table, but here we have implemented 3-D hash over the cloud that increases our complexity to O (constant) where this constant majorly depends upon internet connection over HTTP. Hence while searching through every directory i.e. D1, D2, D3 and D4 using no-escape search, the data retrieval time observed was 3.06 seconds, 4.10 seconds, 7.19 seconds and 10.05 seconds respectively. Here, the slight increase of the time is due to data processing before displaying over UI module. Figure 10 shows the results. No-escape search is extremely effective against slow unindexed window search.

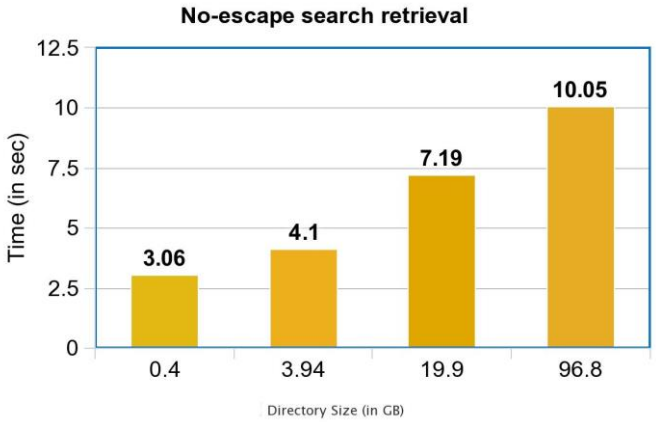

Fig. 10. Search Times in Windows Search in Directories of different sizes

C. *Analysis of retrival on second data set*

Considering second dataset it is defined that given number of queries are present in the index and system will retrieve data given number of times. For example, Consider F1, it consists of 1000 indexes/keys and system will retrieve those 1000 keys in continuation. Manually, analyzing window retrieval speed over thousands of Queries is not possible so we have used window's command line tool 'FIND', Documentation and example [11]. Pre-indexed window directory is used to collect results. Results of no-escape search is observed on average download speed of 1 MBps. Time taken by no-escape search can be further decreased by increasing internet speed. By calculating mean of per query retrieval of data in figure 11, where NES refers to No-escape search and Windows refers to Window Search.

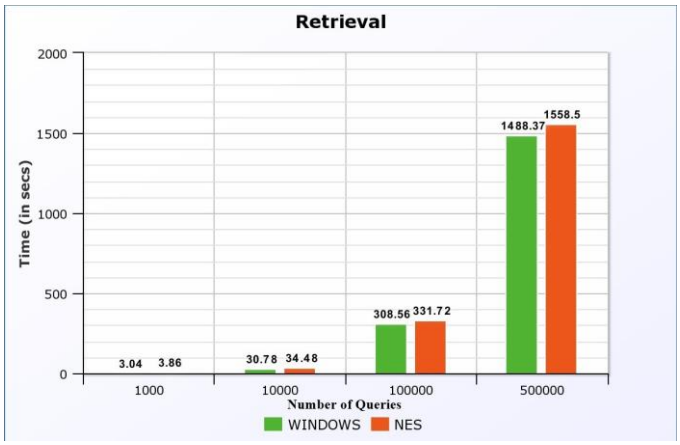

Fig.11. Retrieval time of multiple queries

V. CONCLUSION

We have presented "no-escape search", a cloud-based in file content search tool in this paper. As compared to windows based search, main advantages of this tool are drastic reduction of data retrieval time, reporting position(s) of searched words inside corresponding files and retrieval of similar words simultaneously. Moreover, the use of 3D hash data structure makes it majorly independent of directory size. Automated directory scanning and uploading to cloud is the only time taking but one-time event. The time taken to upload can be reduced by creating compressed JSON file of words for reporting results. Moreover, as cloud's data is globally available we could develop this module into file sync able system like Google Drive etc.

VI. REFERENCES

[1] Microsoft Cooperation, "Window Search Frequently Asked Questions", https://web.archive.org/web/20110924212903/http://www.microsoft.com/windows/products/winfamily/desktopsearch/technicalresources/techfaq.mspx, May 2014.

[2] Microsoft Support, "Availability of the Windows Desktop Search add-in for Files on Microsoft Networks", https://support.microsoft.com/en-us/help/918996/availability-of-the-windows-desktop-search-add-in-for-files-on-microsoft-networks, Jan 2012.

[3] Allen, Frank W. "A File Index for Document Storage and Retrieval Utilizing Descriptor Fragments." *The Computer Journal* 25.1 (1982): 2-6.

[4] Tang, Rongfeng, Dan Meng, and Sining Wu. "Optimized implementation of extendible hashing to support large file system directory." In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pp. 452-455. IEEE, 2003.

[5] Shen, Heng Tao, Yanfeng Shu, and Bei Yu. "Efficient semantic-based content search in P2P network." *IEEE Transactions on Knowledge and Data Engineering* 16, no. 7 (2004): 813-826

[6] Tao, Yufei, and Cheng Sheng. "Fast nearest neighbor search with keywords." *IEEE transactions on knowledge and data engineering* 26, no. 4 (2014): 878-888

[7] Bhandari, Akshita, Ashutosh Gupta, and Debasis Das. "A framework for data security and storage in Cloud Computing." In *Computational Techniques in Information and Communication Technologies (ICCTICT), 2016 International Conference on*, pp. 1-7. IEEE, 2016.

[8] Hui, K., Yates, A., Berberich, K. and de Melo, G., 2017. A Position-Aware Deep Model for Relevance Matching in Information Retrieval. *arXiv preprint arXiv:1704.03940*

[9] Travis-Sun, "pywin32", GitHub repository, https://github.com/Travis-Sun/pywin32 , Dec 2012

[10] Yusuke Shinyama, "pdfminer.six", GitHub repository, https://github.com/pdfminer/pdfminer.six, June 2016.

[11] Microsoft TechNet, "Command line reference", https://technet.microsoft.com/en-us/library/cc725655(v=ws.11).aspx, April 2012