**directories :**

( all directories stored in **helper** file ) **:**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | i\_index\_dir | -> base directory of inverted index |
|  |  | f\_index\_main\_dir | -> base directory of forward index |
|  |  | docs\_subdir\_log | -> log of forward indexed docs |
|  |  | i\_log | -> log of inverted indexed docs |
|  |  | docs\_path | -> path where data\_set is stored either in separate files or a |
|  |  |  | single huge file. |
|  |  | stopwords\_path | -> path of stopwords file |
|  |  |  |  |
|  |  |  |  |
| **Indexer :**  while 1:  index() **index() :** | | |

f\_process = multiprocessing.Process( target = f\_placer() ) i\_process = multiprocessing.Process( target = i\_placer() )

## f\_placer() :

**#** docs = [ name of all documents from dos\_path ]

**#** indexed\_docs = { documents that are already indexed, from docs\_subdir\_log }

# f\_index\_folders = [name of all folders in f\_index directory]

# for doc in docs not in indexed\_docs:

get\_out\_path\_for\_f\_index ( doc, f\_index\_folder ) path\_of\_nth\_doc = docs\_path + "\\" + str ( doc )

forward\_indexer ( stopwords\_path, path\_of\_nth\_doc, output\_path ) if forward\_indexer return 0 update sub\_dir\_log

## forward\_indexer ( stopwords\_file, data\_set, output\_file )

read stop words file open targeted file read it doc by doc lowercase all words

substitute non alphanumeric with space

tokenize

remove stopwords stem each token

enumerate through tokens and create desired dict pattern

( { ‘word’ : [ location\_weight, x ], ‘word\_n’ : [ location\_weight, x ] } )

out\_batch\_of\_file = out\_batch\_of previous docs + ”,” + doc\_no + ”,” + doc\_size + ”,” forward\_batch + ”\n”

write forward\_index onto file

# i\_placer ( ) :

dir\_dic\_f = get\_sub\_dir\_of\_findex( docs\_subdir\_log ) i\_dict = read\_ilog() for key in dir\_dict\_f :

if key not in i\_dict :

f\_index = f\_index\_main\_dir + "\\" + dir\_dict\_i[key] + "\\" + key

inverted\_batch = inverted\_indexer(f\_index) for word in inverted\_batch :

store\_on\_hashed\_directory ( word, inverted\_batch, i\_index\_dir, 1)

update i\_log

## i\_indexer (forwardindex\_file ) :

read forward index file and create your desired pattern for inverted index

I did

{ ‘word’ : { doc\_id : [ total\_words, x ] }, ‘word\_n’ : { doc\_id\_n : [ total\_words, x ] } }

return this inverted dictionary

## helper:

Variables :

directories ( already mentioned above )

dict\_rest = dictionary of word restricted by Microsoft in filenaming.

Methods :

get\_size(path):

query\_parser(stopwords\_path\_, query): get\_stopword\_path(): get\_qdict(path\_list): get\_wposting\_path(query\_string): unsorted\_result(idict, query\_list): sort\_result(r\_doc):

get\_hashed\_directory(higher\_directory, key\_word, mask): get\_hashed\_directory\_tyag(higher\_directory, key\_word, mask): check\_for\_path(hashed\_path):

output\_on\_hashed\_path(information, full\_hashed\_address, key\_word, restricted, single\_nested\_dict\_or\_str):

get\_sub\_dir\_of\_findex(sub\_dir\_log\_file\_path): read\_doc\_sub\_directories(): read\_ilog():

get\_out\_path\_for\_f\_index(doc, f\_index\_folder):

store\_on\_hashed\_directory(key\_word, information, base\_directory, single\_nested\_dict\_or\_str):

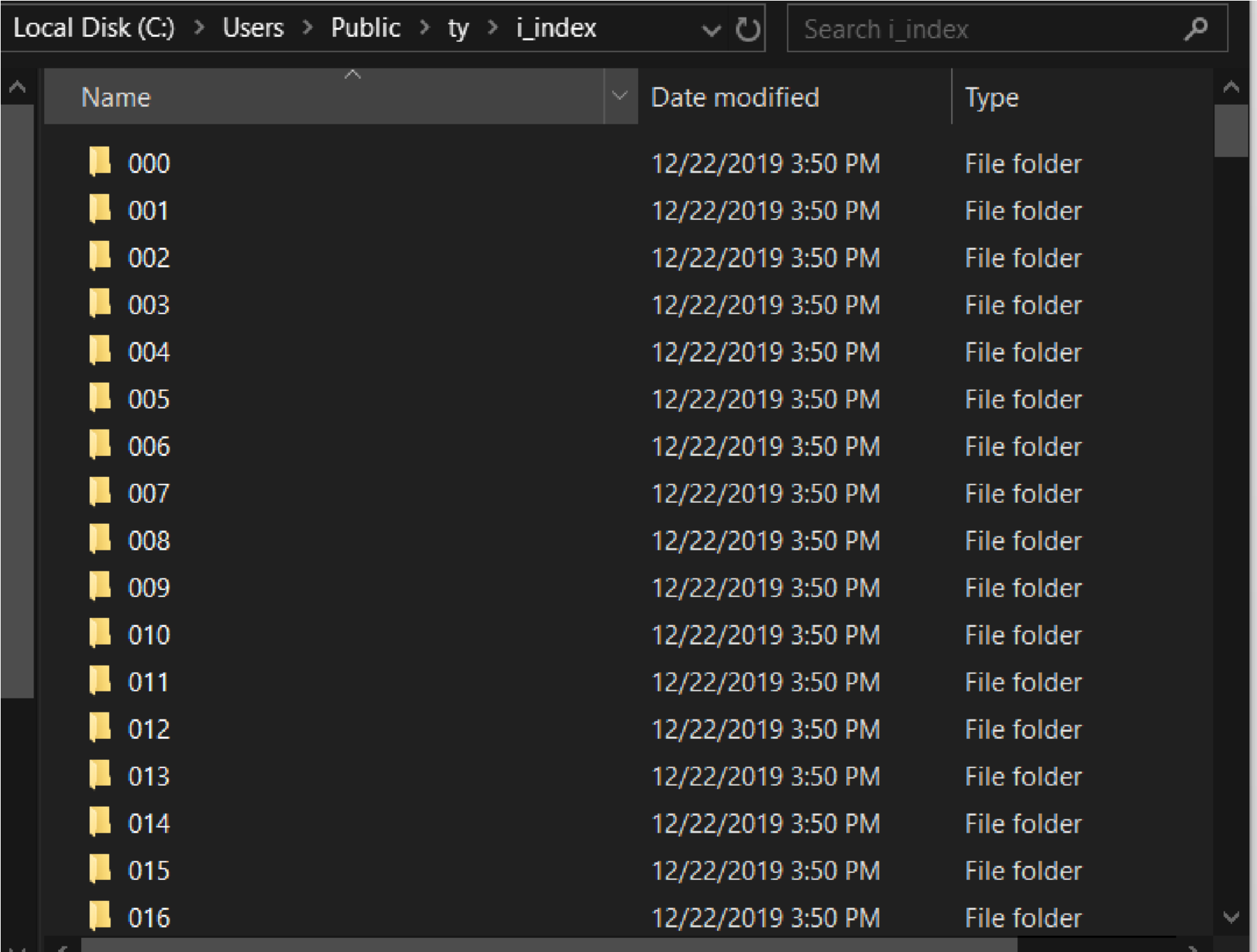
## Hashing to generate directory :

**One word index in one pickle file:**

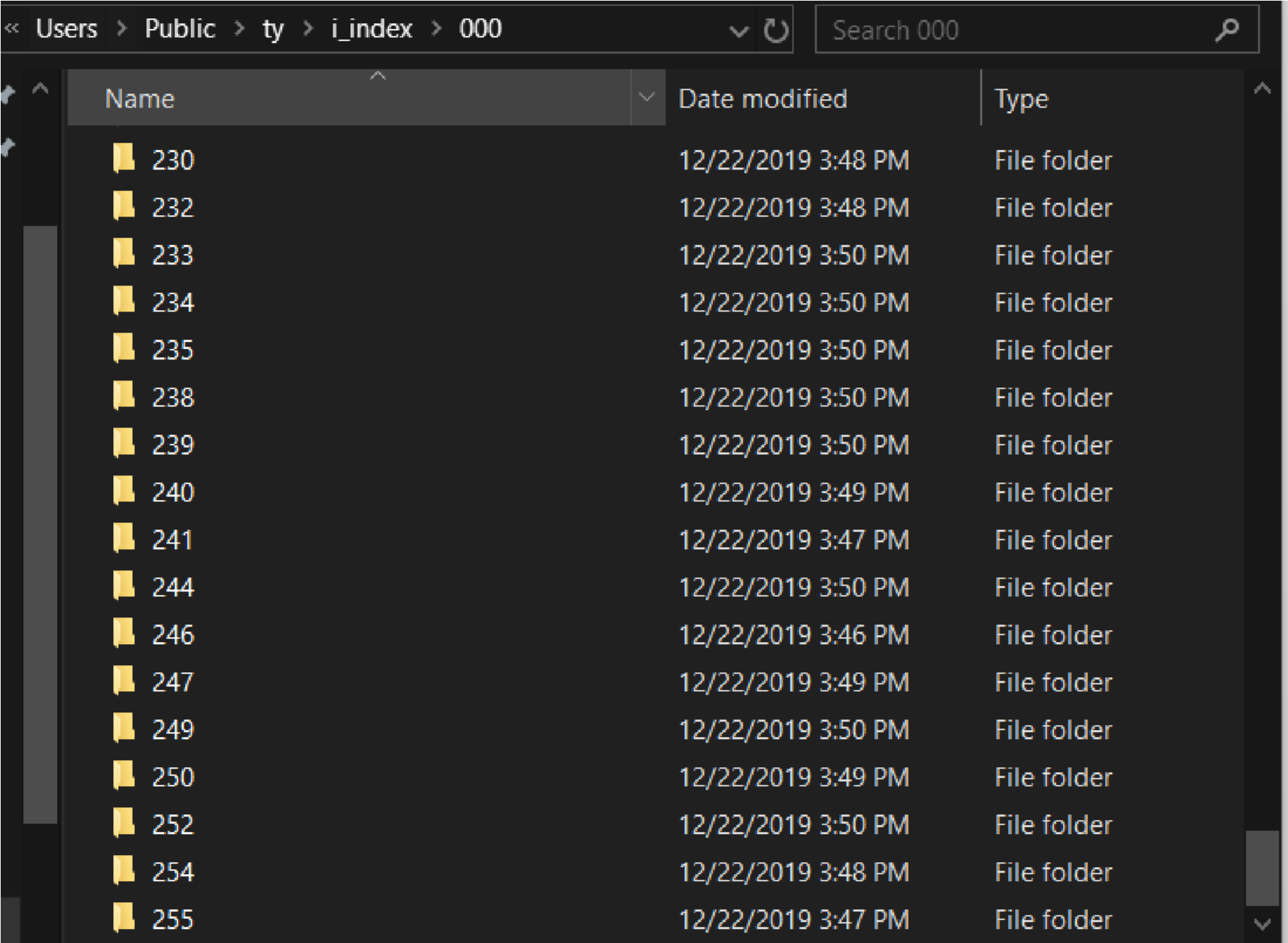
It is far better to get inverted index of word which we desire not of any other file, so inverted index is not stored in clustered form, each pickle file corresponds to inverted index of a single word. This saves file loading time, it takes 0.7 seconds to load a 3MB dictionary. If algorithm is to be scaled over a really huge dataset then storing indexes of words in clustered form could easily result in file size greater than 15 MB , this large pickle dictionary would take whopping 3.5 seconds to be loaded!

**What about nesting of folders ?**

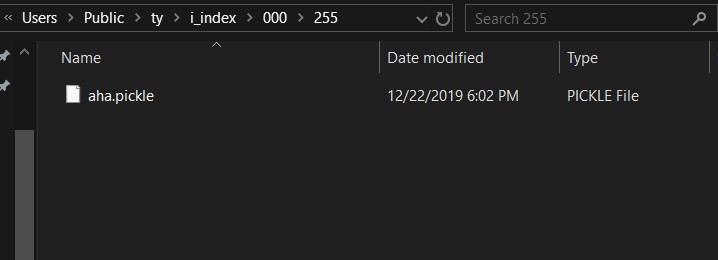
**Outer chain of folders**



**Inner chain of folders**



**Boundry :**



**WHY ALL THIS NESTING ?**

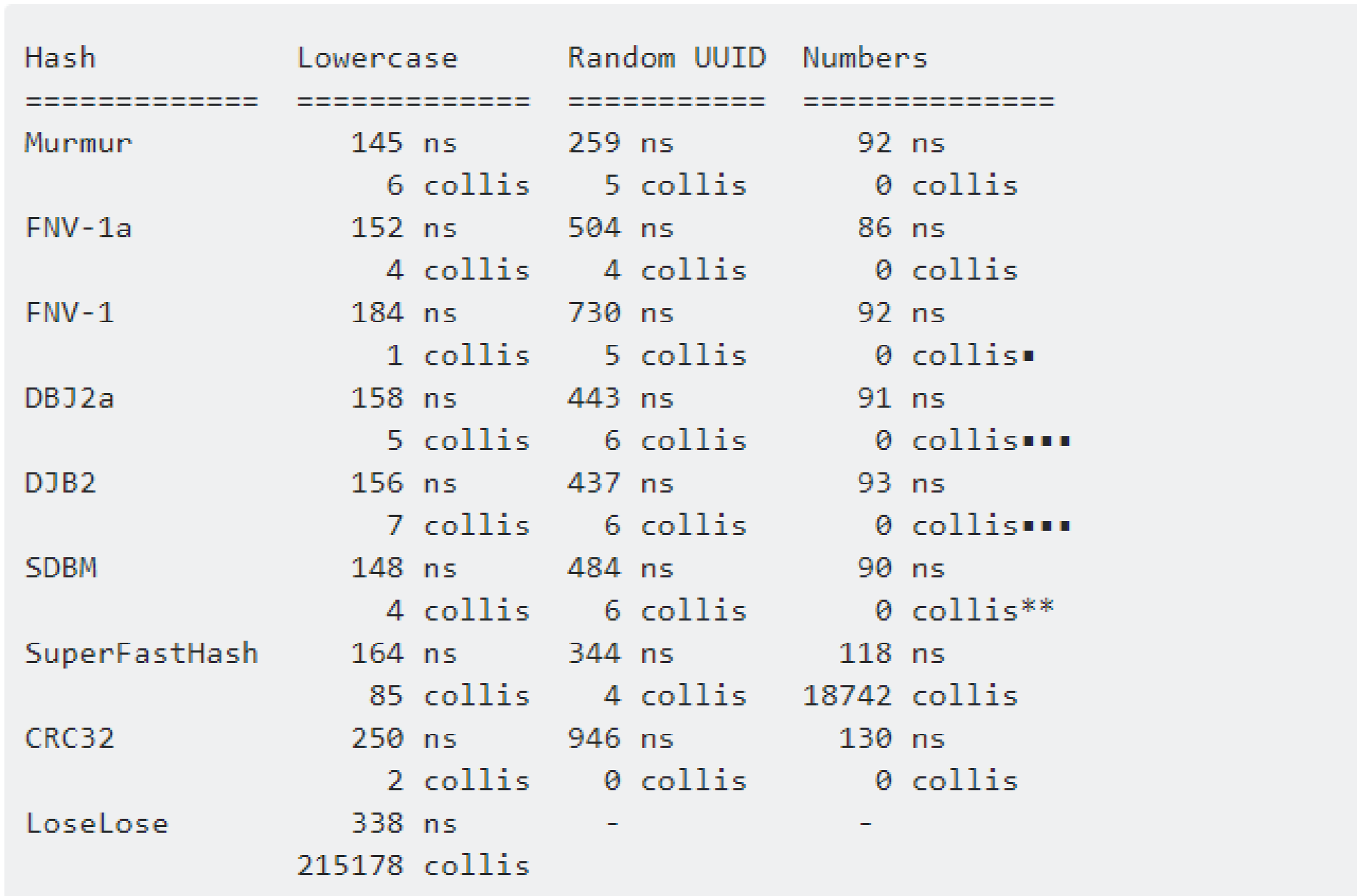
In window OS directory structure is a Tree. The time it took OS to search a query name handled to it by our Program ( in this caase the name of our pickle file ) is not actually O ( 1 ) even if wee know the complete path of file.

It depends on two factors the depth of node holding our required file and the number of childrens of it’s parent node. The files in directory structure are actually leaf nodes, The time it takes OS to find a leaf node is O ( n ).

Since there could be tens of thousands of nodes in one directory then searching for node that is end of list could be expensive.

This nested folder structure reduces the number of file nodes in any folder node, hence minimizing load time.

**Why chose Mur-Mur Hash :**

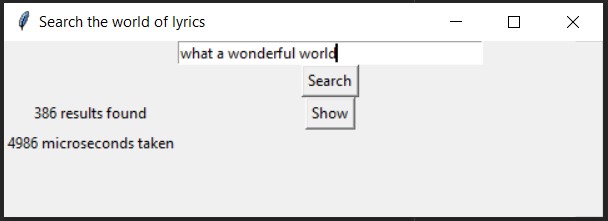


**Why no just build a lexicon and use word ID as key word in hashing, it would be fast as seen above ?**

Numbers are not hashable. **could use str ( word\_id ) ?**

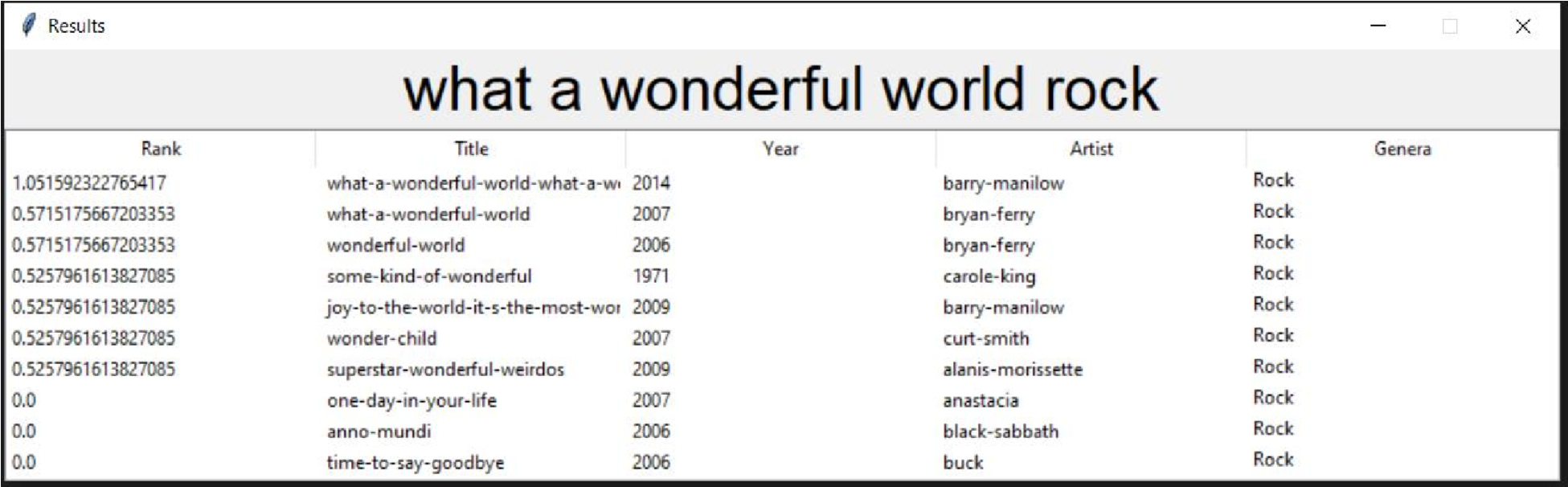
It would result in clustering of files within a folder, hence we won’t be getting proper benefit from hashing. Greater the number of character in word we are hashing lesser the number of collisions, and wider the range of hashed positions.

**Speed PERFORMANCE and Result relevance:**





**TO FILTER BY TITLE OR YEAR OR GENRE JUST TYPE it in QUERY**

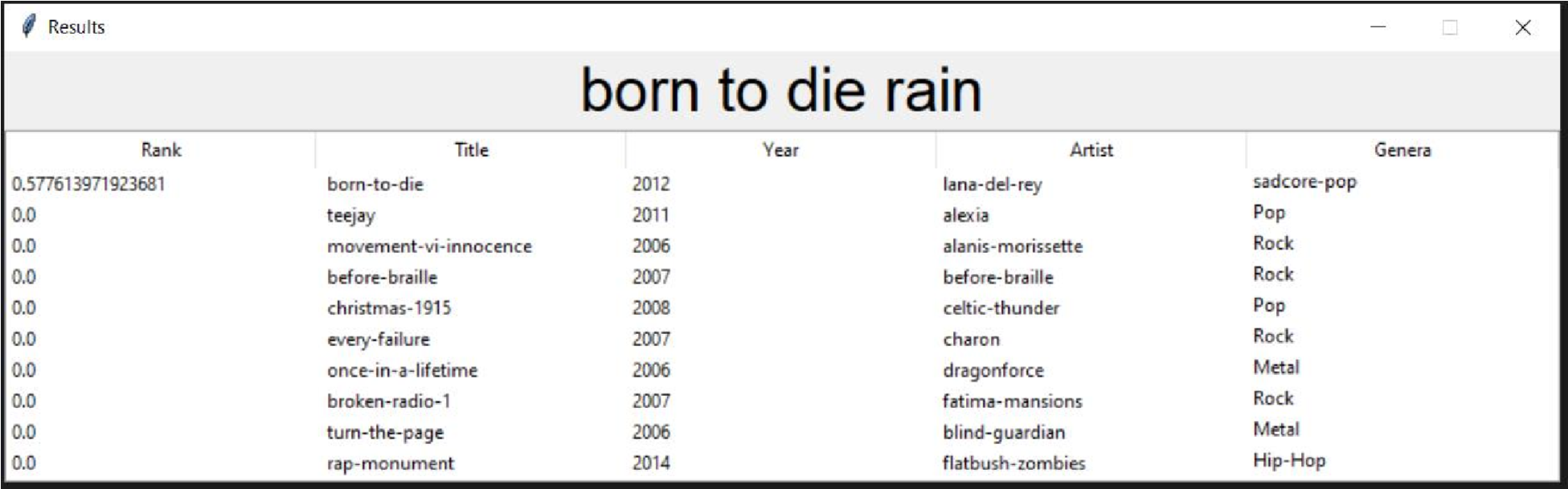


**WHAT ARE ZERO RANKED RESULTS in ABOVE SNIPPET ?**

These are the documents do not contain entire phrase as a whole instead they have parts of it scattered in document

**SEE THE CHANGED RANKING WITH CHANGED QUERY :**





**SHOW DOC :**

