A REPORT

ON

Information Retrieval - Assigment 1

BY

DIVYATEJA PASUPULETI	**2021A7PS0075H**
ADARSH DAS	**2021A7PS1511H**
KUSH AGRAWAL	**2021A7PS0142H**
SHIVAM ATUL TRIVEDI	**2021A7PS1512H**

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI,
HYDERABAD CAMPUS

February 2024

Contents

List of Figures

List of Tables

Chapter 1

Comparing grep with inverted index-based boolean retrieval

1.1 Introduction

In this experiment, the goal was to use `s2/s2_query.json` to benchmark the pure-python boolean retrieval implementation against the `grep` command.

1.2 Non-grep boolean retrieval

First, a custom `and` query was written for boolean retrieval as follows.

```
def custom_and_query(query_terms, postings, doc_freq):
    # postings for only the query terms
    postings_for_keywords = {}
    doc_freq_for_keywords = {}

    for q in query_terms:
        postings_for_keywords[q] = postings[q]

    # store doc frequency for query token in
    # dictionary

    for q in query_terms:
        doc_freq_for_keywords[q] = doc_freq[q]

    # sort tokens in increasing order of their
    # frequencies

```

```
    sorted_tokens = sorted(doc_freq_for_keywords.items(), key=lambda x: x[1])

    # initialize result to postings list of the
    # token with minimum doc frequency

    result = postings_for_keywords[sorted_tokens[0][0]]

    # iterate over the remaining postings list and
    # intersect them with result, and updating it
    # in every step

    for i in range(1, len(postings_for_keywords)):
        result = intersection(result, postings_for_keywords[sorted_tokens[i
    ][0]])
        if len(result) == 0:
            return result

    return result
```

Then for the 'non-grep' boolean retrieval, we defined the function as follows

```
def non_grep_boolean_retrieval_single_query(query: str, postings, doc_freq):
    try:
        output = custom_and_query(query.split(" "), postings, doc_freq)
        return output
    except KeyError:
        print(f"Error with query: {query}")
        return []
```

1.3 Profiling grep

In this part, `grep` was invoked from Python using the `subprocess` module. The rationale behind this choice was to enhance the efficiency of data retrieval while maintaining the same method for `json` parsing.

The code implementation is as follows.

```
def grep_boolean_retrieval_single_query(query: str):
    try:
        doc_count = 0
        for keyword in query.split(" "):
            command = f"grep '{keyword}' s2/intermediate/doc_to_tsv.tsv >
    experiments/experiment1_temp/output{doc_count}.tsv"
            subprocess.run(
                command,
```

```
                shell=True,
                check=True,
                stdout=subprocess.PIPE,
            )
            doc_count += 1

        if doc_count == 1:
            return
        command = f"grep -Fx -f {' '.join(f'experiments/experiment1_temp/output{i}.tsv' for i in range(doc_count))} > experiments/experiment1_temp/common.tsv"
        subprocess.run(
            command,
            shell=True,
            check=True,
            stdout=subprocess.PIPE,
        )
    except subprocess.CalledProcessError:
        print(f"Error with query: {query}")
```

1.4 Conclusion

Finally for comparison we used the below function to compare the two implementations.

```
def compare_grep_non_grep():
    postings, doc_freq = load_index_in_memory("s2/")

    with open("s2/s2_query.json") as f:
        queries = [i["query"] for i in json.load(f)["queries"]]

    # Temporary directory for storing the output of grep
    os.makedirs("experiments/experiment1_temp", exist_ok=True)

    grep_profiler = time_profile.Profile()
    non_grep_profiler = time_profile.Profile()

    grep_time = 0
    non_grep_time = 0

    results = []

    for query in tqdm(queries):
        logging.info(f"Query: {query}")

        non_grep_profiler.enable()
        non_grep_boolean_retrieval_single_query(query, postings, doc_freq)
```

```
        non_grep_profiler.disable()
        stats = pstats.Stats(non_grep_profiler)
        non_grep_current_iteration = stats.total_tt - non_grep_time
        non_grep_time = stats.total_tt

        logging.info(
            f"Profiled {non_grep_current_iteration} seconds at {index} for {non_grep_boolean_retrieval_single_query.__name__}"
        )

        grep_profiler.enable()
        grep_boolean_retrieval_single_query(query)
        grep_profiler.disable()
        stats = pstats.Stats(grep_profiler)
        grep_current_iteration = stats.total_tt - grep_time
        grep_time = stats.total_tt

        logging.info(
            f"Profiled {grep_current_iteration} seconds at {index} for {grep_boolean_retrieval_single_query.__name__}"
        )

        results.append([query, non_grep_current_iteration, grep_current_iteration])
    pd.DataFrame(results).to_csv(
        "experiments/profiles/experiment1_combined.csv",
        index=False,
        header=["Query", "Non-Grep Time", "Grep Time"],
    )
```

The output from the benchmark is given below.

TABLE 1.1: Time - Boolean Retrieval Vs `grep`

Query	**Boolean Retrieval (s)**	**Grep (s)**
Query	Non-Grep Time	Grep Time
deep learning	0.000183477	0.074344299
artificial intelligence	6.18E-05	0.029476679
information retrieval	0.000159252	0.176850776
machine learning	0.000154192	0.083644471
question answering	6.39E-05	0.064826229
noun phrases	5.26E-05	0.033851017
penn treebank	4.75E-05	0.021505133

Continued on next page

Table 1.1 – continued from previous page

Query	Boolean Retrieval (s)	Grep (s)
speech recognition	0.00010033	0.059795822
data mining	0.000177908	0.273050264
computer vision	8.93E-05	0.088555036
reinforcement learning	0.000108028	0.031267968
natural language	0.00014556	0.092417298
autoencoder	3.49E-05	0.009334353
ontology	2.41E-05	0.007066162
sentiment analysis	0.000137775	0.03194099
sap	2.49E-05	0.010337599
lstm	2.31E-05	0.00853887
natural language processing	0.000143331	0.108906995
semantic web	0.000166542	0.088412574
mooc	2.36E-05	0.010144498
human computer interaction	0.000127733	0.115752558
eye movement clustering	6.89E-05	0.051858677
semantic relations	6.86E-05	0.07767778
efficient estimation of word representations in vector space	0.001177075	0.226480003
big data	0.000149046	0.057558611
audio visual fusion	8.46E-05	0.046869138
object detection	0.000120915	0.108291928
gfdm	2.32E-05	0.008542506
neural network	0.000133392	0.103607343
generalized extreme value	8.71E-05	0.042271402
information geometry	0.000104482	0.179225559
image panorama video	0.00010355	0.113313379
data science	0.00012133	0.278272184
semantic parsing	6.91E-05	0.070767498
augmented reality	5.91E-05	0.026902936
imbalanced data	0.000114044	0.036822981
recommender system	0.000139163	0.029916129
inverse reinforcement learning mixture	0.000117037	0.047056293
transfer learning	0.000104655	0.037281787
cnn	2.44E-05	0.013722693
dynamic programming segmentation	0.000104217	0.109063992
natural language interface	0.000105558	0.100603497

Continued on next page

Table 1.1 – continued from previous page

Query	Boolean Retrieval (s)	Grep (s)
genetic algorithm	0.000104474	0.038788725
prolog	2.35E-05	0.007255107
contact prediction	6.04E-05	0.033396395
wifi malware	4.19E-05	0.024786994
nsdi machine learning	2.21E-05	0.011411267
forensics and machine learning	0.000562069	0.061121764
words to speech	0.000538988	0.088794704
information theory	0.000140639	0.183379739
morphology morphological	4.76E-05	0.020938489
category theory	6.53E-05	0.02728745
graph theory	0.000101029	0.117200906
smart thermostat	4.63E-05	0.033395611
exploit vulnerability	5.14E-05	0.061789379
reinforcement learning and video game	0.000388299	0.095595068
system health management	0.000138048	0.262517517
spatial multi agent systems	6.62E-05	0.096191933
service composition	6.51E-05	0.069954402
mobile payment	6.93E-05	0.056916587
3 axis gantry	6.39E-05	0.844754558
softmax categorization	4.50E-05	0.04069431
cost aggregation	6.46E-05	0.074470631
chinese dialect	5.01E-05	0.020755837
depth camera	5.56E-05	0.041312569
mobile tcp traffic analysis	0.000126186	0.073566747
collective learning	9.91E-05	0.032521281
robust production planning	9.08E-05	0.083394791
memory hierarchy	6.89E-05	0.052517255
hashing	3.60E-05	0.008656487
comparable corpora	6.10E-05	0.038317619
knowledge graph	0.00010385	0.107972237
social media	9.92E-05	0.075006769
deep learning surveillance	0.000114389	0.073277224
cryptography	2.41E-05	0.006344218
parametric max flow	5.85E-05	0.039193206
deep reinforcement learning	0.000122713	0.070251533

Continued on next page

Table 1.1 – continued from previous page

Query	Boolean Retrieval (s)	Grep (s)
varying weight grasp	5.66E-05	0.042134148
dirichlet process	7.87E-05	0.029443134
word embedding	8.52E-05	0.09520452
graph drawing	7.19E-05	0.12829751
robust principal component analysis	0.000158504	0.10948818
differential evolution	6.93E-05	0.032618556
seq2seq	2.52E-05	0.003765554
document logical structure	0.000113996	0.083963568
duality	2.37E-05	0.007688259
variable neighborhood search	0.000105881	0.060260169
urban public transportation systems	0.000132699	0.065732607
edx coursera	5.92E-05	0.005848714
fdir	2.49E-05	0.011367444
cryptography key management	9.85E-05	0.034409521
ontology construction	7.70E-05	0.032533241
go game	4.93E-05	0.315080744
personality trait	4.95E-05	0.03012473
sparse learning	0.000113179	0.051265606
directed hypergraph	4.75E-05	0.03251469
inventory management	9.18E-05	0.031109878
clojure	4.44E-05	0.013586923
ontology semantic web	0.000134959	0.054234834
convolutional neural network time series	0.000215868	0.072709576
TOTAL	0.01087382	7.503237322
AVERAGE	0.000108738	0.075032373
MIN	2.2056E-05	0.003765554
MAX	0.001177075	0.844754558

Python boolean retrieval (MiB)	Python & `grep` (MiB)
144.17	102.49

TABLE 1.2: Memory - Boolean Retrieval vs `grep`

During the profiling process, `tqdm` and other libraries were eliminated to streamline the performance analysis. The results indicate that `grep` outperforms the `Python` implementation in memory profile by 28.91%. But in the time profile `Python` has a better performance than `grep` by about 98.61%.

Chapter 2

Linguistic post-processing of the vocabulary

2.1 Methodology

Lemmatization is a process where we try to acquire the root of the word and replaces the query word with the root word to ensure better activity in the dictionary.

Stemming is the process of removing common suffixes from all the words.

```
# Initialize stemmer, lemmatizer and stop words
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
```

We use the above classes to lemmatize, stem and remove stop words from our postings dictionary. This will allow us to get better search results by ignoring unnecessary results. We use nltk for the same.

We end up using a memory of 160MiB in the process and a time of 3.940 seconds on average.

Chapter 3

Compare hash-based and tree-based implementation of dictionaries

3.1 Introduction

In this experiment, we compared the performance of hash-based and tree-based implementations of dictionaries. The primary focus was on evaluating their efficiency in handling boolean retrieval tasks using the provided dataset, s2/s2 query.json. We implemented a tree-based data structure, specifically a trie, and compared its performance metrics including time taken for retrieval and memory consumption against a hash-based implementation.

3.2 Methodology

We implemented a trie data structure to serve as the basis for our tree-based dictionary. We conducted boolean retrieval operations using the tree-based dictionary for all 100 queries provided in the 's2/s2 query.json' dataset.

We use the following snippet of code for the same:

```
def boolean_retrieval_multiple_words(self, sentence):
        words = sentence.split()
        result_array = None
        for word in words:
            array = self.search(word)
            if array is not None:
                if result_array is None:
                    result_array = array.copy()
                else:
```

```
10                      result_array = intersection(result_array, array)
11          return result_array
```

3.3 Time Comparision Report

We compared the time consumed by the tree-based implementation with a hash-based implementation.

TABLE 3.1: Time - Tree-based Vs Hash-based

Query	Tree-based (s)	Hash-based (s)
deep learning	0.003702553	0.00050468
artificial intelligence	0.001173096	0.000333092
information retrieval	0.0043596	0.000353446
machine learning	0.004077311	0.000333525
question answering	0.00066511	0.000222195
noun phrases	0.00031978	0.000199431
penn treebank	0.000257335	0.000199051
speech recognition	0.001697404	0.000259516
data mining	0.004008402	0.000357331
computer vision	0.001444177	0.000242097
reinforcement learning	0.003360057	0.000269024
natural language	0.002786729	0.00030975
autoencoder	2.58E-05	0.000175994
ontology	2.53E-05	0.000172362
sentiment analysis	0.002866687	0.000294794
sap	1.67E-05	0.000177951
lstm	1.89E-05	0.000181781
natural language processing	0.004194796	0.00031264
semantic web	0.001915033	0.000273709
mooc	1.97E-05	0.000187021
human computer interaction	0.00328643	0.000329691
eye movement clustering	0.00137547	0.000278109
semantic relations	0.00158841	0.000292574
efficient estimation of word representations in vector space	0.024570932	0.001374097
big data	0.005004996	0.000274563
audio visual fusion	0.001570555	0.000231886

Continued on next page

Table 3.1 – continued from previous page

Query	**Tree-based (s)**	**Hash-based (s)**
object detection	0.0015128	0.000257594
gfdm	1.84E-05	0.000162174
neural network	0.00258813	0.000284154
generalized extreme value	0.001153101	0.000221421
information geometry	0.003114286	0.000248964
image panorama video	0.001946021	0.000257655
data science	0.004319128	0.000261947
semantic parsing	0.001303085	0.000210313
augmented reality	0.000392709	0.000203935
imbalanced data	0.005076926	0.00025208
recommender system	0.00331709	0.00028742
inverse reinforcement learning mixture	0.003730273	0.00026723
transfer learning	0.003647586	0.000291537
cnn	1.62E-05	0.000174921
dynamic programming segmentation	0.001853537	0.000268813
natural language interface	0.003400218	0.00026214
genetic algorithm	0.00258583	0.000385255
prolog	2.50E-05	0.000190091
contact prediction	0.000730543	0.000215295
wifi malware	0.000234278	0.000297989
nsdi machine learning	0.003629938	1.55E-05
forensics and machine learning	0.020885332	0.000825268
words to speech	0.020756532	0.001161216
information theory	0.006654304	0.000460011
morphology morphological	0.000537189	0.000310522
category theory	0.002094915	0.000347237
graph theory	0.002971645	0.00038564
smart thermostat	0.000487702	0.000299694
exploit vulnerability	0.000831481	0.000363275
reinforcement learning and video game	0.03026462	0.000576099
system health management	0.003671942	0.000300375
spatial multi agent systems	0.000527265	0.000223644
service composition	0.000741339	0.000224819
mobile payment	0.000855001	0.000226436
3 axis gantry	0.000542185	0.00024349

Continued on next page

Table 3.1 – continued from previous page

Query	**Tree-based (s)**	**Hash-based (s)**
softmax categorization	0.000315641	0.000232516
cost aggregation	0.000835212	0.000243687
chinese dialect	0.000322785	0.000198152
depth camera	0.000593918	0.000208042
mobile tcp traffic analysis	0.003315422	0.000287196
collective learning	0.003236544	0.000255445
robust production planning	0.001350968	0.000247939
memory hierarchy	0.000826858	0.000222868
hashing	2.03E-05	0.000180993
comparable corpora	0.000711375	0.000220617
knowledge graph	0.001876299	0.000297652
social media	0.000961045	0.000374946
deep learning surveillance	0.004807285	0.000282752
cryptography	3.04E-05	0.000175106
parametric max flow	0.000523099	0.000266105
deep reinforcement learning	0.002998361	0.000290733
varying weight grasp	0.000415611	0.000210827
dirichlet process	0.001297892	0.000231408
word embedding	0.000795556	0.000210581
graph drawing	0.000868595	0.000223396
robust principal component analysis	0.004391734	0.000309757
differential evolution	0.000801542	0.000216345
seq2seq	1.99E-05	0.000178451
document logical structure	0.001962618	0.000294243
duality	2.20E-05	0.000173632
variable neighborhood search	0.001573325	0.000256416
urban public transportation systems	0.00312614	0.000295618
edx coursera	0.000100529	0.00018448
fdir	1.58E-05	0.00017558
cryptography key management	0.002589797	0.000251525
ontology construction	0.000773574	0.000217964
go game	0.000519358	0.000209881
personality trait	0.00029554	0.000209363
sparse learning	0.004320311	0.000287674
directed hypergraph	0.000352807	0.000199147

Continued on next page

Table 3.1 – continued from previous page

Query	**Tree-based (s)**	**Hash-based (s)**
inventory management	0.000857154	0.000211311
clojure	2.02E-05	0.000217294
ontology semantic web	0.003135104	0.000270253
convolutional neural network time series	0.005688466	0.000332394
AVERAGE	0.002674169	0.000284568
MAX	1.5846E-05	1.5501E-05
MIN	0.03026462	0.001374097

3.4 Memory Comparision Report

Memory(in MiB) using Tree based	Memory(in MiB) using Hash based
354.6	199.5

Table 3.2: Memory - Tree based vs Hash based

3.5 Conclusion

After comparing hash-based and tree-based implementations of dictionaries, it's clear that hash-based dictionaries generally offer faster performance. Hash-based dictionaries demonstrate lower average lookup times (0.000284568 vs. 0.002674169 for trie-based dictionaries) and superior maximum lookup times (0.001374097 vs. 0.03026462 for trie-based dictionaries). Although trie-based dictionaries have a better minimum lookup time, hash-based dictionaries still showcase overall faster performance. Therefore, for applications prioritizing speed with acceptable memory overhead, hash-based dictionaries are the preferable choice.

Chapter 4

Building Permuterm and Tree based Indices for Wildcard Queries

4.1 Introduction

In this experiment, the goal was to use ‘s2/s2_wildcard.json‘ to benchmark tree based and permuterm based indexing.

4.2 Methodology for Permuterm Indexing

Permuterm indices are used with wildcard queries. For a query hello, we add a $ at the end to signify the end and then we generate all the other permuterm terms which will be hello$, ello$h and so on.

We use the following snippet of code for the same:

```
term = term + "$"
for i in range(len(term)):
    permuterm_index[term[i:] + term[:i]] = term
```

We store all these permuterms in another dictionary and map them to the original word and the original word is mapped to the listings. We have to understand that even though this will lead to an extra dictionary that should keep track of the permuterms we will be able to efficiently perform wildcard queries.

Now we can use the permuterm indices to perform prefix searches. Basically if we have a query ‘m*n$‘ and we are searching in a vocabulary, we will have to filter all strings that start with

m and end with n. We first convert this such that the * comes in the end and the query will basically become ‘n$m*‘

If we take man and moron, they will have the main permuterm as man$ and moron$. If we take iterations, we will also get, ‘n$ma‘ and ‘n$moro‘ respectively which will match with the previous query thereby giving us the result we need.

The attached code snippet will perform the same:

```
# PREFIX SEARCH ON PERMUTERM INDEX
def prefix_search_on_permuterm_index(permuterm_index: dict, query: str):
    query = query + "$"
    # We have to rotate it such that last char is *
    while query[-1] != "*":
        query = query[-1] + query[:-1]
    # We remove the * from the end
    query = query[:-1]
    # Now we can search for the query in the permuterm index
    for key in permuterm_index:
        if key.startswith(query):
            logging.info(permuterm_index[key])
```

4.3 Methodology for Tree based Indexing

We use a trie based indexing approach for this. A trie is a special data structure used in searches for strings. It reduces the time complexity of insertion, searching and checking a string in a given data.

```
# Trie Node for the tree based index
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
```

This data structure basically contains a dictionary of further nodes for the future words and an *is_end_of_word* boolean which represents if a word that ends here is a legitimate word.

Now the methodology for searching here involves taking a wildcard query. Say we have, ‘m*n$‘ and we are searching in a vocabulary. In the case of a tree based search we create 2 tries for the vocabulary one will contain all the words in correct order while the other one will contain all the words in reverse order. Now for the given query ‘m*n‘, we split into two parts across the * and this will give us two terms ”m” and ”n”.

We use ”m” on the trie which goes in order and finds all the words which have prefix: ”m”. We use ”n” on the reverse words trie and find all words which have prefix ”n” but this would imply they end with ”n”.

Now taking an intersection of these will give us the actual result needed. A small snippet showing the same is attached:

```
    # Split the term at the *
    first_half = term["query"].split("*")[0]
    second_half = term["query"].split("*")[1]

    # This will return all the words that start with the first half
    first_half_results = start_trie.starts_with(first_half)

    # This will return reverse of all the words that end with the second half
    second_half_results = end_trie.starts_with(second_half[::-1])

    # reverse all the words in the second half results because this trie has all
     reverse words
    second_half_results = [i[::-1] for i in second_half_results]

    final_results = list(
        set(first_half_results).intersection(set(second_half_results))
    )
```

4.4 Time Comparision Report

We notice that the number of function calls taken in a single run of permuterm is 23869367 function calls while in a single run of trie based retrieval comes out to be 5904202 and this is constant

Query	Time(in s) using Permuterm	Time(in s) using Trie
Query	Permuterm Time	Trie Time
l*ng	0.002349641	0.027953612
in*e	0.006932009	0.044048011
ret*val	0.000286584	0.001141495
mac*	0.00175833	0.438443873
*nswering	0.000172393	0.320846082
*ses	0.004055832	0.324650992
tree*	0.000770071	0.430481598
sp*h	0.000292845	0.007494341
m*g	0.004316828	0.03900688
v*n	0.001332634	0.022097075
rein*	0.000481064	0.402939278
lan*ge	0.00042705	0.002485307
auto*	0.002361518	0.4400495
on*y	0.000847217	0.019855787
senti*	0.00054254	0.417899167
s*p	0.000906921	0.031512376
l*m	0.00085843	0.013897755
nat*l	0.000409537	0.01439655
we*	0.007690947	0.415865777
mo*c	0.000480469	0.015392069
com*er	0.000452156	0.009078242
cl*g	0.000518573	0.023213967
s*ic	0.00320952	0.032531448
re*s	0.005449757	0.049600916
*ata	0.001730935	0.315090621
f*sion	0.000156759	0.010195249
o*ject	0.000243351	0.008336386
g*dm	4.43E-05	0.006831329
n*ural	0.000596496	0.011285357
e*treme	6.81E-05	0.011530731
TOTAL	0.049742782	3.908151771
AVERAGE	0.001658093	0.130271726
MIN	4.4285E-05	0.001141495
MAX	0.007690947	0.4400495

Table 4.1: Time - Permuterm vs Trie

4.5 Memory Comparision Report

Memory(in MiB) using Permuterm	Memory(in MiB) using Trie
1287.2 MiB	331.2

Table 4.2: Memory - Permuterm vs Trie

4.6 Conclusion

One key part here is the fact that for a set of queries the number of results is always same regardless of the number of trials that are done.

Second point of concern is the fact that the time used in permuterm based retrieval is lower than that of retrieval using two tries and searching from front and back. This could be because of the fact that in permuterm based retrieval, we use only one trie while in the second approach we use two trees and proceed to take intersection. We had initially used a hash based permuterm approach where the permuterm was coming out to be slower than the two trie approach but in this case where we are using one trie approach for permuterm the permuterm comes out to be faster.

With respect to the memory however, we observe that permuterm occupies a lot of memory especially due to multiple node data structures and the fact that we store a single word with all it's permutations. So for example let's say we have a term hello, in permuterm it's stored as hello$, ello$h, llohe, lohel, o$hell. With respect to two tries however we are just storing hello and olleh.

Chapter 5

Implementing Tolerant Retrieval

5.1 Introduction

In this experiment we needed to implement **tolerant retrieval** on the 30 special wildcard-boolean queries. Wildcard queries are of the form prefix*suffix, say wil*rd where '*' is a special symbol representing 0 or more characters **(do note that either of the prefix or the suffix may be an empty string)**. Basically, we need words that start with the prefix 'wil' and end with the suffix 'rd', say the word wildcard itself.

In the set of wildcard-boolean queries, each query may be a combination of one or more terms. In the case of more than one term, we need to take the intersection of the result of each term. This principle is called **boolean retrieval** .

5.2 Methodology

5.2.1 Non-Wildcard Terms

For non-wildcard terms we use the file postings.tsv to obtain the postings list (dictionary of term -¿ [document list]). For each term in this dictionary, we check if the edit distance is less than the minimum threshold or not. This is done to retrieve words that may actually be the query term but have been misspelt in the document. This feature uses principles of tolerant retrieval.

In the function for calculating edit distance, addition and deletion have a cost of 1 whereas substitution has a cost of 2. Keeping in mind both we keep a threshold of 2. Terms with an edit distance of less than equal to 2 mean that either a character has been added/deleted from the original query term or one of its characters has been replaced/substituted. Once we have the

postings of all the similar terms, we take the union of all the sets of postings.

```
1
2     for query_ in tqdm(queries["queries"]):
3             logging.info(f"Query: {query_}")
4             words = query_["query"].split()
5             final_doc_list = list()
6
7             for w in words:
8                 if "*" in w:
9                     pass
10                else:
11                    logging.info(f"::> Non Wildcard Term: {w}")
12                    docs = set()
13                    for term in postings:
14                        if edit_distance(w, term) <= threshold and abs(len(w)-
    len(term)) <= 1:
15                            logging.info(
16                                f"::> Non Wildcard Term that's closer to this: {
    term}"
17                            )
18                            docs = docs.union(set(postings[term]))
19                    final_doc_list.append(docs)
```

5.2.2 Wildcard Terms

For wildcard terms we use a Trie based retrieval system as done in experiment 4. This data structure basically contains a dictionary of further nodes for the future words and an is end of word boolean which represents if a word that ends here is a legitimate word. Similar to experiment 4 we search for words beginning with the prefix and for the words that on reversing start with the reverse of the given suffix. We take an intersection of the 2 sets of words to get the desired result. Now, similar to what we did for the non-wildcard terms, for all the words in this set we find the set of documents containing that word from the postings list and we keep on taking their union.

```
1
2     for query_ in tqdm(queries["queries"]):
3             logging.info(f"Query: {query_}")
4             words = query_["query"].split()
5             final_doc_list = list()
6
7             for w in words:
8                 if "*" in w:
9                     first_half = w.split("*")[0]
```

```
                    second_half = w.split("*")[1]
                    first_half_results = start_trie.starts_with(first_half)
                    second_half_results = end_trie.starts_with(second_half
    [::-1])
                    second_half_results = [i[::-1] for i in second_half_results]
                    wildcard_words = list(
                        set(first_half_results).intersection(set(
    second_half_results))
                    )
                    wildcard_docs = set()
                    for word in wildcard_words:
                        wildcard_docs = wildcard_docs.union(set(postings[word]))
                    final_doc_list.append(wildcard_docs)
```

Now, finally we take intersection of the set of documents corresponding to each term since we need to perform boolean retrieval.

```

    result = set(final_doc_list[0])
    for i in range(1, len(final_doc_list)):
        result = result.intersection(final_doc_list[i])

    result = list(result)
```

5.3 Time Comparision Report

Query	Time
l*ng	0.075043002
in*e	0.09645921600000001
ret*val	0.0037211590000000017
mac*	0.8434273480000001
*nswering	0.5632397350000002
*ses	0.597304694
tree*	0.8688084310000002
sp*h	0.016582435999999756
m*g	0.07371032899999985
v*n	0.06722674899999959
rein*	0.8658513080000008
lan*ge	0.004064508999999994
auto*	0.9667116999999994
on*y	0.047340473000000216
senti*	0.9371008640000005
s*p	0.06304003800000046
l*m	0.0367404269999998
nat*l	0.0392452969999999
we*	0.9691681149999996
mo*c	0.023181322000000115
com*er	0.021528790999999714
cl*g	0.03608246699999995
s*ic	0.057559685000000194
re*s	0.11368526900000031
*ata	0.623448107999999
f*sion	0.023530097000000083
o*ject	0.0169889560000005
g*dm	0.017481895000001302
n*ural	0.01804549599999916
e*treme	0.02610997700000084

5.4 Memory Comparision Report

```
Line #    Mem usage    Increment  Occurrences   Line Contents
```

```
=============================================================
     8    114.1 MiB    114.1 MiB           1   @memory_profile
     9                                         def tolerant_retrieval():
    10    114.1 MiB      0.0 MiB           1       tolerant_profile = time_profile.Profile()
    11
    12                                             # Open the file and search for the term
    13    114.1 MiB      0.0 MiB           2       with open("./s2/s2_wildcard_boolean.json") as f:
    14    114.1 MiB      0.0 MiB           1           queries = json.load(f)
    15
    16    211.8 MiB     97.7 MiB           1       postings, term_freq = load_index_in_memory("./s2/")
    17    211.8 MiB      0.0 MiB           1       threshold = 2
    18
    19    340.7 MiB    128.9 MiB           1       start_trie, end_trie = generate_trie_indices(term_freq=term_freq)
    20
    21    340.7 MiB      0.0 MiB           1       initial_time = 0
    22    340.7 MiB      0.0 MiB           1       times = []
    23
    24    358.2 MiB    -73.7 MiB          31       for query_ in tqdm(queries["queries"]):
    25    358.2 MiB    -70.0 MiB          30           logging.info(f"Query: {query_}")
    26
    27    358.2 MiB    -70.0 MiB          30           tolerant_profile.enable()
    28
    29    358.2 MiB    -70.0 MiB          30           words = query_["query"].split()
    30    358.2 MiB    -70.0 MiB          30           final_doc_list = list()
    31
    32    359.8 MiB   -235.0 MiB          95           for w in words:
    33    359.8 MiB   -170.0 MiB          65               if "*" in w:
    34    359.8 MiB   -112.0 MiB          42                   logging.info(f"::> Wildcard Term: {w}")
    35                                                         # Split the term at the *
    36    359.8 MiB   -112.0 MiB          42                   first_half = w.split("*")[0]
    37    359.8 MiB   -112.0 MiB          42                   second_half = w.split("*")[1]
    38                                                         # This will return all the words that start with the first half
    39    359.8 MiB   -109.9 MiB          42                   first_half_results = start_trie.starts_with(first_half)
```

```
35     40                                                  # This will
   return reverse of all the words that end with the second half
36     41    356.9 MiB   -150.1 MiB          42
   second_half_results = end_trie.starts_with(second_half[::-1])
37     42                                                  # reverse all the
    words in the second half results
38     43    359.7 MiB -1454308.6 MiB      788551
   second_half_results = [i[::-1] for i in second_half_results]
39     44
40     45    359.8 MiB   -211.2 MiB          84                    wildcard_words =
   list(
41     46    359.8 MiB   -102.3 MiB          42                        set(
   first_half_results).intersection(set(second_half_results))
42     47                                                        )
43     48    359.8 MiB   -112.4 MiB          42                    wildcard_docs =
   set()
44     49    359.8 MiB -18966.0 MiB       13762                    for word in
   wildcard_words:
45     50    359.8 MiB -18853.6 MiB       13720                        wildcard_docs
    = wildcard_docs.union(set(postings[word]))
46     51                                                        # logging.info(f
   "::> Docs: {wildcard_docs}")
47     52    359.8 MiB   -112.4 MiB          42                    final_doc_list.
   append(wildcard_docs)
48     53
49     54                                                  else:
50     55    359.8 MiB    -58.1 MiB          23                    logging.info(f"
   ::> Non Wildcard Term: {w}")
51     56    359.8 MiB    -52.2 MiB          23                    docs = set()
52     57    359.8 MiB -3947860.4 MiB     1696480                      for term in
   postings:
53     58                                                            if (
54     59    359.8 MiB -3947844.9 MiB     1696457
   edit_distance(w, term) <= threshold
55     60    359.8 MiB  -1743.9 MiB         755                            and abs(
   len(w) - len(term)) <= 1
56     61                                                            ):
57     62    359.8 MiB  -2050.9 MiB         912                            logging.
   info(
58     63    359.8 MiB  -1020.9 MiB         456                                f"::>
    Non Wildcard Term that's closer to this: {term}"
59     64                                                            )
60     65                                                            # logging
   .info(f"::> Docs: {postings[term][0]}")
61     66    359.8 MiB  -1020.9 MiB         456                            docs =
   docs.union(set(postings[term]))
62     67                                                        # logging.info(f
   "::> Docs: {docs}")
```

```
63     68    359.8 MiB    -52.5 MiB          23                    final_doc_list.
   append(docs)
64     69
65     70    358.2 MiB    -93.4 MiB          30            result = set(
   final_doc_list[0])
66     71    358.2 MiB   -159.7 MiB          65            for i in range(1, len(
   final_doc_list)):
67     72    358.2 MiB    -85.8 MiB          35                result = result.
   intersection(final_doc_list[i])
68     73
69     74    358.2 MiB    -73.9 MiB          30            result = list(result)
70     75
71     76    358.2 MiB    -73.9 MiB          30            tolerant_profile.disable
   ()
72     77
73     78    358.2 MiB    -73.9 MiB          30            stats = pstats.Stats(
   tolerant_profile)
74     79    358.2 MiB    -73.9 MiB          30            current_iteration = stats
   .total_tt - initial_time
75     80    358.2 MiB    -73.9 MiB          30            initial_time = stats.
   total_tt
76     81
77     82    358.2 MiB    -73.9 MiB          30            times.append([query_,
   current_iteration])
78     83
79     84    358.2 MiB   -147.8 MiB          60            logging.info(
80     85    358.2 MiB    -73.9 MiB          30                f"Profiled {
   current_iteration} seconds at {index} for {tree_based_search.__name__}"
81     86                                                  )
82     87
83     88    354.8 MiB     -3.4 MiB           1       pd.DataFrame(times).to_csv("
   experiment5.csv", index=False, header=["Query", "Time"])
```

Memory(in MiB) usage for Tolerant Retrieval
354.2

Table 5.1: Memory Usage