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)
deep learning	0.002076381	0.073177405
artificial intelligence	0.00036296	0.03289399
information retrieval	0.002146084	0.20760942
machine learning	0.002232579	0.095004947
question answering	0.000394112	0.074738486
noun phrases	0.000187595	0.038224785
penn treebank	0.000157805	0.024845402
speech recognition	0.000963448	0.05972555

Continued on next page

Table 1.1 – continued from previous page

Query	Boolean Retrieval (s)	Grep (s)
data mining	0.002722286	0.399311456
computer vision	0.000824453	0.098138264
reinforcement learning	0.001844027	0.03457368
natural language	0.001408135	0.099818529
autoencoder	1.84E-05	0.009986511
ontology	1.96E-05	0.007699669
sentiment analysis	0.001588771	0.035883601
sap	1.98E-05	0.011686215
lstm	2.07E-05	0.009831506
natural language processing	0.002270124	0.108374086
semantic web	0.001088903	0.081617016
mooc	1.91E-05	0.011294304
human computer interaction	0.001732362	0.132038769
eye movement clustering	0.000526751	0.090448985
semantic relations	0.000842205	0.078754137
efficient estimation of word representations in vector space	0.009644045	0.246614077
big data	0.002538403	0.057749055
audio visual fusion	0.000785513	0.055351572
object detection	0.000892734	0.106947388
gfdm	1.99E-05	0.010278152
neural network	0.001670921	0.087610309
generalized extreme value	0.000722306	0.051630851
information geometry	0.001989533	0.17920947
image panorama video	0.001279674	0.119522532
data science	0.002606607	0.349928001
semantic parsing	0.000837145	0.081120548
augmented reality	0.000274849	0.02600902
imbalanced data	0.002644648	0.037697955
recommender system	0.00196627	0.032485175
inverse reinforcement learning mixture	0.000652938	0.047706976
transfer learning	0.001864855	0.043895472
cnn	1.95E-05	0.015674028
dynamic programming segmentation	0.001154723	0.088053054
natural language interface	0.001992324	0.106175066
genetic algorithm	0.001465135	0.039704934

Continued on next page

Table 1.1 – continued from previous page

Query	Boolean Retrieval (s)	Grep (s)
prolog	1.87E-05	0.008161413
contact prediction	0.00046123	0.039166466
wifi malware	0.000140784	0.02533168
nsdi machine learning	1.50E-05	0.011915115
forensics and machine learning	0.01041035	0.060962702
words to speech	0.008488062	0.095173357
information theory	0.002487045	0.196880018
morphology morphological	0.000177094	0.024008949
category theory	0.000675596	0.029710385
graph theory	0.001004156	0.119247975
smart thermostat	0.000186203	0.041641374
exploit vulnerability	0.000346786	0.064524996
reinforcement learning and video game	0.002386957	0.088384748
system health management	0.002440205	0.320264747
spatial multi agent systems	0.000526837	0.097811685
service composition	0.000446342	0.065143825
mobile payment	0.000519869	0.054638268
3 axis gantry	0.000282467	1.432705379
softmax categorization	0.000137604	0.018814235
cost aggregation	0.000580445	0.075017599
chinese dialect	0.000227489	0.02329608
depth camera	0.000342123	0.046755439
mobile tcp traffic analysis	0.001992239	0.077304309
collective learning	0.001842412	0.035306082
robust production planning	0.000903872	0.075795469
memory hierarchy	0.000502684	0.049503957
hashing	2.13E-05	0.009754053
comparable corpora	0.000384697	0.045227857
knowledge graph	0.001206135	0.109404616
social media	0.000588855	0.074684822
deep learning surveillance	0.003074941	0.13482449
cryptography	1.95E-05	0.007068714
parametric max flow	0.000390632	0.07010266
deep reinforcement learning	0.001751655	0.073475152
varying weight grasp	0.000313387	0.047754672

Continued on next page

Table 1.1 – continued from previous page

Query	Boolean Retrieval (s)	Grep (s)
dirichlet process	0.001026954	0.026898211
word embedding	0.000543121	0.096138892
graph drawing	0.000619701	0.150727797
robust principal component analysis	0.003456626	0.147910936
differential evolution	0.001048134	0.043688406
seq2seq	2.45E-05	0.005744224
document logical structure	0.001488401	0.113610408
duality	2.42E-05	0.010859179
variable neighborhood search	0.001811254	0.081273253
urban public transportation systems	0.002158286	0.083177845
edx coursera	0.000161943	0.006270496
fdir	3.16E-05	0.016543095
cryptography key management	0.001598005	0.05275009
ontology construction	0.000576738	0.045348492
go game	0.00034192	0.565036623
personality trait	0.000170211	0.033202854
sparse learning	0.001992568	0.052878432
directed hypergraph	0.000225351	0.033650648
inventory management	0.00052518	0.023683359
clojure	2.01E-05	0.010627653
ontology semantic web	0.001877339	0.057235072
convolutional neural network time series	0.003067203	0.104519949
AVERAGE	0.001265716	0.091105795
MIN	1.496E-05	0.005744224
MAX	0.01041035	1.432705379

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	0.002015936	0.002127612
artificial intelligence	0.000342353	0.000453457
information retrieval	0.001870757	0.002232468
machine learning	0.00236285	0.002535224
question answering	0.000413711	0.000532311
noun phrases	0.000187531	0.000486422
penn treebank	0.000151572	0.000293176
speech recognition	0.000897459	0.001020809
data mining	0.00241209	0.002932368
computer vision	0.000770576	0.001141071
reinforcement learning	0.001797552	0.001994966
natural language	0.001377456	0.001655945
autoencoder	1.75E-05	0.000144921
ontology	1.92E-05	0.000153305
sentiment analysis	0.001563569	0.001734221
sap	1.01E-05	0.00014532
lstm	1.58E-05	0.000144353
natural language processing	0.002277919	0.002449226
semantic web	0.001085647	0.001493071
mooc	1.36E-05	0.000151719
human computer interaction	0.001829755	0.002392785
eye movement clustering	0.000607311	0.000644985
semantic relations	0.000913796	0.001589647
efficient estimation of word representations in vector space	0.014461107	0.010156779
big data	0.003987899	0.003133677
audio visual fusion	0.000818409	0.000937769

Continued on next page

Table 3.1 – continued from previous page

Query	**Tree-based (s)**	**Hash-based (s)**
object detection	0.000937566	0.001115061
gfdm	1.45E-05	0.000184967
neural network	0.001594811	0.001655942
generalized extreme value	0.000676237	0.000942026
information geometry	0.002036054	0.002537505
image panorama video	0.001391061	0.001422356
data science	0.002354997	0.002813363
semantic parsing	0.000812937	0.00110513
augmented reality	0.000335736	0.00040587
imbalanced data	0.002674254	0.002754419
recommender system	0.001868696	0.002134985
inverse reinforcement learning mixture	0.002203774	0.000759624
transfer learning	0.001916332	0.00205879
cnn	1.18E-05	0.000141099
dynamic programming segmentation	0.001064033	0.001210222
natural language interface	0.001945174	0.002134693
genetic algorithm	0.001540213	0.0017026
prolog	1.73E-05	0.000200268
contact prediction	0.0004857	0.000549271
wifi malware	0.000149135	0.000259886
nsdi machine learning	0.002176696	5.59E-06
forensics and machine learning	0.011745151	0.01133286
words to speech	0.009181028	0.009988671
information theory	0.002418964	0.002632147
morphology morphological	0.000185891	0.000313588
category theory	0.000685118	0.000794601
graph theory	0.000919029	0.001042777
smart thermostat	0.000157191	0.000541155
exploit vulnerability	0.00028646	0.000435263
reinforcement learning and video game	0.010645565	0.002545368
system health management	0.002192316	0.002570057
spatial multi agent systems	0.000393947	0.000635629
service composition	0.000414491	0.000541493
mobile payment	0.000472954	0.000694765
3 axis gantry	0.000251375	0.000409174

Continued on next page

Table 3.1 – continued from previous page

Query	**Tree-based (s)**	**Hash-based (s)**
softmax categorization	0.000151556	0.00025922
cost aggregation	0.00046818	0.000632617
chinese dialect	0.000219028	0.00033232
depth camera	0.000337709	0.000475206
mobile tcp traffic analysis	0.001984245	0.002431105
collective learning	0.00214619	0.002229211
robust production planning	0.000951791	0.001216526
memory hierarchy	0.000527748	0.000682917
hashing	1.45E-05	0.000164138
comparable corpora	0.000374081	0.000515217
knowledge graph	0.001258891	0.001396035
social media	0.000660485	0.000797325
deep learning surveillance	0.00233272	0.002309795
cryptography	1.93E-05	0.000158614
parametric max flow	0.000318857	0.000477557
deep reinforcement learning	0.001789317	0.001796842
varying weight grasp	0.000223605	0.000423075
dirichlet process	0.00079407	0.000968338
word embedding	0.000489516	0.000667503
graph drawing	0.000530583	0.000709483
robust principal component analysis	0.002709386	0.002929446
differential evolution	0.000475448	0.00059829
seq2seq	2.80E-05	0.000265468
document logical structure	0.001264863	0.001423847
duality	2.22E-05	0.000163899
variable neighborhood search	0.001054538	0.001222996
urban public transportation systems	0.002036844	0.002143483
edx coursera	7.42E-05	0.000212624
fdir	1.25E-05	0.000148627
cryptography key management	0.001296221	0.001347929
ontology construction	0.000477762	0.00065661
go game	0.000264025	0.000414852
personality trait	0.000153167	0.000287198
sparse learning	0.002101295	0.002148624
directed hypergraph	0.000228292	0.000355442

Continued on next page

Table 3.1 – continued from previous page

Query	**Tree-based (s)**	**Hash-based (s)**
inventory management	0.000514344	0.000639552
clojure	2.38E-05	0.000282544
ontology semantic web	0.001647483	0.001712864
convolutional neural network time series	0.003114527	0.002906362
AVERAGE	0.001414691	0.001417525
MIN	1.009E-05	5.595E-06
MAX	0.014461107	0.01133286

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

Tree-based implementations demonstrate faster retrieval times but consume more memory compared to hash-based counterparts.Conversely, hash-based implementations offer more memory-efficient solutions, providing constant-time access to elements. This efficiency makes them suitable for large datasets and scenarios with limited memory resources.Ultimately, the choice between tree-based and hash-based implementations depends on specific application requirements, including performance objectives and memory constraints. Both approaches have their merits, and selecting the optimal solution entails considering trade-offs between speed and memory consumption.

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	0.367552988	0.042116001
in*e	0.511680376	0.059597619
ret*val	0.385330903	0.001231875
mac*	0.465039057	0.648745376
*nswering	0.366529495	0.325682398
*ses	0.46778136	0.350090067
tree*	0.423355557	0.451063456
sp*h	0.435653721	0.006775305
m*g	0.426165196	0.035763865
v*n	0.581672655	0.026784873
rein*	0.389962551	0.448592688
lan*ge	0.406765221	0.002580433
auto*	0.396046176	0.426587162
on*y	0.750358131	0.019578048
senti*	0.906941451	0.416241126
s*p	2.053164299	0.029849403
l*m	0.426892472	0.013630402
nat*l	0.38992654	0.014650192
we*	0.374066327	0.424280252
mo*c	0.36084923	0.010850308
com*er	0.440311659	0.010368085
cl*g	0.366703364	0.02020781
s*ic	0.370765435	0.034204028
re*s	0.376054906	0.052412186
*ata	0.371681315	0.324759663
f*sion	0.440658545	0.011039596
o*ject	0.368796265	0.008779438
g*dm	0.447450835	0.007946573
n*ural	0.37214277	0.012361606
e*treme	0.385467435	0.012301251
AVERAGE	0.498559077	0.145067417
MIN	0.36084923	0.001231875
MAX	2.053164299	0.648745376
TOTAL	14.82576624	4.249071085

Table 4.1: Time - Permuterm vs Trie

4.5 Memory Comparision Report

Memory(in MiB) using Permuterm	Memory(in MiB) using Trie
274.5	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 queries 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 higher than that of retrieval using tries. This could be because of the fact that tries are an optimal way to prefix search through words when compared to checking permuterms in a dictionary and it's mapping to the actual word and checking the postings.

With respect to the memory however, we observe that trie occupies a lot of memory especially due to multiple node data structures and the fact that we need two trees for this retrieval.

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"])
```