COMP 479 - Project 3

François LaBerge

November 18, 2020

1 Running the Program

Use the -h on "query.py" and "indexer.py".

2 Code Design

2.1 Exectuion time comparison

The indexer for project 2 took 40.60 seconds to complete while the SPIMI indexer in project 3 took 482.56 seconds to run, which is a sizable increase. We improved the execution time by using multithreading.

2.2 Spimi Invert

The following snippet shows our SPIMI invert algorithm, without its helper methods.

```
def spimi_invert(token_stream, max_dictionary_size, block_path):
    count = 0
    token_accumulator = []
    block_names = block_name_generator(block_path)
    for token in token_stream:
        count += 1
        token_accumulator.append(token)
        if count == max_dictionary_size:
            block_name = next(block_names)
            block = build_block(token_accumulator, block_name[1])
            write_to_pickle(block, block_name[0])
            count = 0
            token_accumulator = []

    block_name = next(block_names)
    block: Block = build_block(token_accumulator, block_name[1])
    write_to_pickle(block, block_name[0])

    build_final_index(block_path)
```

2.3 Block merge

The follwing snippet shows the algorithm to merge two SPIMI blocks.

```
def merge(first_block: Block, second_block: Block, out_path: path):
    new_terms = []
```

```
    new_index = {}
    i, j = 0, 0
    while i < len(first_block.sorted_terms) and j < len(second_block.sorted_terms):
        first_term = first_block.sorted_terms[i]
        second_term = second_block.sorted_terms[j]
        if first_term < second_term:
            new_terms.append(first_term)
            new_index[first_term] = first_block.index[first_term]
            i += 1
        elif first_term > second_term:
            new_terms.append(second_term)
            new_index[second_term] = second_block.index[second_term]
            j += 1
        elif first_term == second_term:
            new_terms.append(first_term)
            merged_postings = merge_postings_list(first_block.index[first_term], second_block.index[sec
            new_index[first_term] = merged_postings
            i += 1
            j += 1

    while i < len(first_block.sorted_terms):
        first_term = first_block.sorted_terms[i]
        new_terms.append(first_term)
        new_index[first_term] = first_block.index[first_term]
        i += 1

    while j < len(second_block.sorted_terms):
        second_term = second_block.sorted_terms[j]
        new_terms.append(second_term)
        new_index[second_term] = second_block.index[second_term]
        j += 1

    new_document_lengths = merge_document_lengths(first_block.document_lengths, second_block.document_l
    new_block = Block(new_index, new_document_lengths, new_terms, first_block.name + second_block.name)
    block_path = "{}/{}.pickle".format(out_path, str(first_block.name) + str(second_block.name))
    write_to_pickle(new_block, block_path)
```

This next snippet shows to algorithm to build the final index by merging the blocks. Note that the blocks are actually committed to storage.

```
def build_final_index(path_string: str):
    merge_pass_directory_template = path_string+"/pass{}"
    pass_count = 1
    in_path = Path(path_string)
    out_path = Path(merge_pass_directory_template.format(pass_count))
    while(get_number_of_files_in_directory(in_path)>1):
        merge_blocks_in_directory(in_path, out_path)
        pass_count += 1
        in_path = out_path
        out_path = Path(merge_pass_directory_template.format(pass_count))

    final_index = load_block_from_pickle(get_files_in_directory(in_path)[0])
    write_to_pickle(final_index, path_string+"/final_index.pickle")
```

2.4 Query Algorithms

The following snippet shows how the different types of queries are implemented

```
result = []
if query_mode == QueryMode.RANKED:
        ranked_documents: RankedDocumentQueue = RankedDocumentQueue()
        for term in terms:
                if term in index:
                        for posting in index[term]:
                                df = len(index[term])
                                tf = posting.term_count
                                document_length = document_lengths[posting.doc_id]
                                rank = rank_term(df, tf, collection_size, document_length, average_docu
                                ranked_documents.push(posting, rank)
        result = [posting.doc_id for posting in ranked_documents]
elif query_mode == QueryMode.AND:
        postings_list = []
        first_term = True
        for term in terms:
                if term in index:
                        postings_list = intersect_postings(postings_list, index[term], first_term)
                        first_term = False
        result = [posting.doc_id for posting in postings_list]
elif query_mode == QueryMode.OR:
        postings_list = []
        for term in terms:
                if term in index:
                        postings_list = postings_list + index[term]
        postings_list.sort(key=lambda p: p.term_count, reverse=True)
        result = [posting.doc_id for posting in postings_list]
```

This snippet shows how the rank of document is calculated.

```
def rank_term(df, tf, collection_size, document_length, average_document_length):
    k = 1
    b = 1
    return log(collection_size/df, 10) * ( ((k+1)*tf) / (k*((1-b)+b*(document_length/average_document_l
```

3 Sample Queries

3.1 Single word Query

We use the query “soup”.

Result from project 2

```
{
        "soup": [
                6338,
                9646,
                10162,
                11004,
```

```
                16852
]
}
```


Result from project 3


```
[
        16852,
        9646,
        6338
]
```


We can see that the data compression in project 1 makes the list of result longer, but that the ranking in project 3 changed the ordering of the results.

3.2 BM25 Queries

We use the query "nuclear weapons race"


```
[
   12794,
   21498,
   6892,
   19944,
   12729,
   14754,
   12552,
   3004,
   11157,
   19289,
   12531,
   18864,
   155,
   16081,
   15730,
   14536,
   13246,
   11672,
   4861,
   12087,
   10901,
   10301,
   12056,
   10166,
   2935,
   2745,
   7675,
   5724,
   5130,
   10949,
   10988,
   4469,
   4393,
```

```
3525,
10626,
4393,
3291,
21453,
10211,
4124,
2734,
21241,
9616,
20060,
2733,
4046,
19836,
18432,
368,
17859,
17793,
2656,
8686,
17668,
2730,
3998,
8237,
8022,
16229,
16081,
15771,
15203,
1574,
7896,
3404,
14829,
7793,
14406,
3291,
7492,
13035,
2280,
2009,
20860,
12550,
12513,
6778,
3014,
12544,
12354,
6444,
12134,
11296,
10501,
```

```
5514,
9348,
11423,
4875,
1756,
3464,
3404,
1574,
1290,
9685,
20392,
9465,
19776,
9192,
17684,
17017,
16905,
3936,
3837,
15706,
15187,
15041,
7532,
7427,
21547,
6143,
12744,
3062,
563,
5262,
9607,
11717,
11487,
10301,
9760,
20760,
195,
6576,
7901,
8073,
15390,
13257,
12907,
17490,
12181,
5690,
7525,
2759,
19909,
8601,
16894,
```

```
  13225,
  16562,
  11487,
  9607,
  17416,
  17721,
  1895,
  17497,
  4049
]
```


3.3 AND query

We use the query "nuclear weapons".


```
[
  1574,
  3291,
  3404,
  4393,
  9607,
  10301,
  11487,
  16081
]
```


3.4 OR query

We reuse the query "nuclear weapons race".


```
[
  3404,
  12552,
  12794,
  16081,
  13246,
  11157,
  1574,
  14406,
  17793,
  20060,
  16081,
  19289,
  368,
  563,
  1756,
  2745,
  3004,
  3062,
  3291,
  3837,
  5130,
```

```
8022,
8686,
9760,
10988,
11487,
12729,
15041,
15187,
17668,
17684,
17859,
19836,
20392,
21241,
1290,
11672,
19944,
155,
195,
1895,
2009,
2280,
2656,
2730,
2733,
2734,
2935,
3014,
3936,
3998,
4046,
4049,
4124,
4393,
4861,
5262,
5514,
6444,
6576,
6778,
6892,
7427,
7492,
7532,
7793,
7896,
7901,
8073,
8237,
8601,
9192,
```

```
9465,
9607,
9616,
9685,
10211,
10301,
10626,
10949,
11423,
11717,
12056,
12087,
12134,
12181,
12354,
12531,
12544,
12550,
12744,
12907,
13035,
13225,
13257,
14829,
15203,
15390,
15706,
15771,
16229,
16894,
16905,
17017,
17416,
17721,
18432,
19776,
19909,
20760,
21453,
1574,
2759,
3291,
3404,
3464,
3525,
4393,
4469,
4875,
5690,
5724,
7525,
```

```
    7675,
    9348,
    9607,
    10166,
    10301,
    10501,
    10901,
    11296,
    11487,
    14536,
    15730,
    17497,
    18864,
    6143,
    12513,
    14754,
    16562,
    17490,
    20860,
    21498,
    21547
]
```


4 Sample Runs

4.1 Query: Democrats' welfare and healthcare reform policies

Some results are too long to display see the associated file for the full result

BM25 see "sampleruns/1_bm25.json".

AND see "sampleruns/1_and.json".

OR see "sampleruns/1_or.json".

For this query, BM25 and OR return over 14000 results, but AND returns 0. However, the highest ranked document of BM25 doesn't to be more relevant to the information need than a lowly ranked document.

4.2 Query: Drug company bankruptcies

BM25 see "sampleruns/2_bm25.json"

AND see "sampleruns/2_and.json".

OR see "sampleruns/2_or.json"

Again, BM25 and OR return very large lists for this query and AND returns none. The BM25 seem fairly relevant, but it makes it obvious the system ignores the syntactic meaning of its queries and documents.

4.3 Query: George Bush

BM25 see "sampleruns/3_bm25.json".

AND see “sampleruns/3_and.json”.

OR see “sampleruns/3_or.json”.

The results of these queries are much smaller. The top ranked results of BM25 seem somewhat relevant.