Project 3: Report

Christopher Di Giacomo

40133600

10 November 2023

COMP479 2232 D

Dr. Sabine Bergler

Table of Contents

SUBPROJECT 1 ..... **3**

Collection Refinements 3
Naïve Index Construction 4
SPIMI Index Construction 4
Theoretical Performance 5
Statistical Performance 6
Index Characteristics 6

SUBPROJECT 2 ..... **7**

Query Processor 8
Conjunction and Disjunction 9
Query Term Ranking 9
BM25 Ranking 10
K1 and b parameters *11*
Queries 11
Single-Term Queries *11*
Multi-term Queries *12*

DESIGN CHALLENGES AND FINAL THOUGHTS **13**

APPENDIX A **14**

Single-Term Project 2 Test Queries: 14
Single-Term Project 2 Challenge Queries: 15

APPENDIX B **16**

Exact information need queries 16
Query Term Ranking 16
BM25 Ranking 17

Subproject 1

This section covers the different design decisions considered for the implementation of Subproject 1, in *subProject1.py*. It lays out the refinements made from the previous project implementation, as well as details pertaining to the newly introduced SPIMI index construction module(s). These details not only pertain to the actual implementation of the modules, but also their comparison, supplemented by various useful figures.

Collection Refinements

Although in the third iteration of interacting with the Reuters21578 corpus, we still find ourselves debating what truly constitutes a document in this corpus. Upon further inspection of the documents included in the collection, there came about a specific case that was not being account for in the earlier projects. While still extracting with respect to the contents of the <TEXT> tags, we must also track the case where the "type" attribute of the tag is set to "brief". This results in a document, without a <BODY> tag, also including garbage text data within the <TEXT> tags. Thus, within our *getTokens()* (previously named *getAllDocuments()*) we now check for the case where this type attribute is set. Like in the last project, the function composes documents with the title, author, and body. However, this time around, it is only done on the condition that the tags exist to avoid tokenizing garbage terms.

Another refinement made was with regards to the actual handling of the returned tokens. Previously, we stored these tokens as a list for each document, which was then mapped with respect to the NEWID of the document, in a dictionary. Thus, the returned structure was a dictionary containing NEWID – tokens list pairs. However, upon further analysis, it was concluded that the NEWID's also have a one-to-one correspondence with the ordering of the documents in the entire collection. Meaning, document with NEWID 1 was the first document in the collection, while the last was document with NEWID 21578. With this discovery, and the fact that at least the title of a document will be extracted (in the case of a "brief" document), we reverted from extracting the NEWID and mapping it to its respective tokens lists. Instead, since the documents are processed in order, the index in which their tokens list resides will

represent the NEWID for that document. Thus, the function now returns a list of tokens lists, where the index of each tokens list (plus one), corresponds to the NEWID for that document.

Naïve Index Construction

As was included in the last project, the naïve index construction is handled by the *naive()* function. This time around, the only change is the introduction of an optional parameter named *testCount* which is defaulted to a value of *None*. This parameter is set in the main method of the file, when the construction times for both the naïve and spimi functions are compared, for the first 10000 terms/pairs. When set, the function will halt the production of term-docID pairs once the *testCount* threshold has been reached, and will go on to build the index as per usual. Additionally, a *startTime* and *endTime* variable are used to determine the elapsed time during construction, which is then returned by the function along with the actual index. The start timer is set just as the function begins creating term-docID pairs, while the end timer is set just as the function exits the construction phase. Apart from this, the only slight difference to the function is how it creates the term, docID pairs, since it no longer needs to extract them from a dictionary as was explained in the subsection above. The functioning is still identical, it just needed optimizations to deal with the changes to the *getTokens()* function mentioned above. Thus, the function creates term-docID pairs, sorts them, removes exact duplicates, and builds the inverted index.

SPIMI Index Construction

This new module, named *spimi()* receives the same two parameters as the *naive()* function, the tokenized collection and the optional *testCount*. As the naïve function, when the *testCount* parameter is set, the function will halt its processing once this threshold is reached. The difference between the *spimi()* and *naïve()* is what the function is doing when it halts. As mentioned in the naïve index section, the function halts the production of pairs with respect to the threshold. However, given the single-pass nature of the spimi function, it halts the actual construction of the index once the threshold is reached. This function, differently from the naïve index module, also keeps track of term frequencies for the purpose of query processing in sub

project 2. It does so by maintaining a postings list for every term, that is composed of tuples, consisting of docID-term frequency pairs.

To give some insight into the functioning of *spimi()*, it begins by accessing the first list of tokens (document) from the *tokensList* parameter. From this individual list, it then begins by iterating over the stored tokens. It checks if the token is already in the index that it is constructing, where if it is **not**, it simply adds a new entry to the index, where the term is the key and the postings list is a tuple with the docID (index of this tokens list in *tokensList*, plus one), and a frequency of 1. The 1 is because this is the first instance of the term in the collection, every other instance will increment this frequency value. Which leads into the opposite case, where the term is already in the index. The function begins by extracting the postings list of the term into a variable *postingsList.* It then checks if the docID of the last tuple stored in this list matches the docID of the current instance of the term. If yes, the function will overwrite the tuple with a new tuple, containing the same docID but an incremented frequency value. If not, the function will append a new tuple consisting of the current docID and a term frequency of 1. It iterates over all tokens, contained in all lists in the *tokensList,* while maintaining a counter variable to track if the *testCount* threshold has been reached. If the threshold is reached, the construction of the index is halted, and returned. As was stated in the previous subsection, the function uses a *startTime* and *endTime* variable to measure and return the elapsed construction time, along with the index. The start timer is set just before the beginning of the *tokensList* iteration and the end timer lies outside of the whole construction phase.

Theoretical Performance

Due to the nature of SPIMI being a single pass construction algorithm, it is by default the more efficient of the two. This can be derived by simply viewing the natural functioning of the two methods. In the case of the naïve method, there is multiple passes over the collection data, specifically in the production/ sorting of pairs. These steps require the function, in the worst case, to pass over all over the processed data, resulting in a time complexity of O(N log n). Although, in the case of the SPIMI method, things are more efficient since the data is only

passed over one single time. Thus, it doesn't require the use of creating/sorting term-docID pairs, it instead builds the index as it processes the token stream(s). This results in a time complexity of O(n), due to all processing being done in a linear manner, with respect to the size of the collection.

Statistical Performance

In the case of actual performance differences, the SPIMI algorithm takes ≈ 5ms to construct the index, while the naïve algorithm takes ≈ 6.5 ms. So *spimi()* outperforms *naïve()* by ≈ 25% (1.5 ms) on average. This difference is solely due to the fact that the naïve() algorithm takes extra time to produce/sort the pairs while the spimi algorithm starts building the index from the beginning. The difference could in fact most likely be larger, had the SPIMI not taken the extra steps to store postings lists as lists of tuples. However, even with the minimal potential added overhead from the creation/overwriting of tuples, the *spimi()* function is still more efficient. The statistics mentioned above can be seen in the figure below, which is the terminal output generated by the *getStats()* module, in charge of displaying the statistical information.

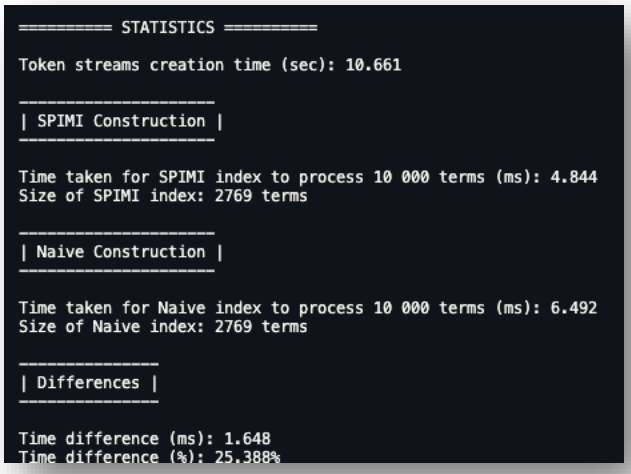

Figure 1: Terminal output for subproject 1, depicting the statistical performance of both index construction methods, as well as their difference.

Index Characteristics

Both indexes are stored in a JSON file via a small module named *save2json()* , which simply saves each index to a JSON object file, within a subdirectory */indexes.* Note that the elapsed construction times for both indexes do not include this file I/O operation. It's also important to

note that although both modules are different index construction algorithms, they both in the end build the same index. Hence the same index sizes as is displayed above in figure 1. Below in figure 2 can be seen a snippet of the same ordered terms in both indexes, displaying their equivalence.

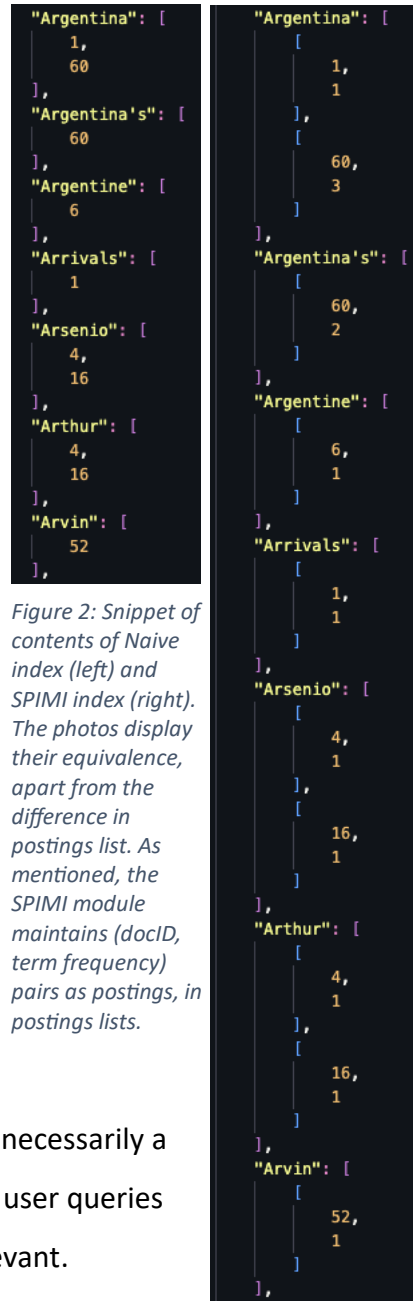

Figure 2: Snippet of contents of Naive index (left) and SPIMI index (right). The photos display their equivalence, apart from the difference in postings list. As mentioned, the SPIMI module maintains (docID, term frequency) pairs as postings, in postings lists.

Furthermore, due to the absence of compression techniques, there are not many equivalence classes permitting for more than one variation of a word to be mapped to a single normalized form. Thus, as can be seen in figure 2 to the right, the terms *Argentina, Argentina's, and Argentine,* are all considered separate terms and thus each use up storage complexity to be stored as such. Whereas, in project 2, the compression techniques like case folding and stemming drastically reduced the dictionary size. This improved the recall, but reduced the precision. However, in the case of this project the precision is heavily emphasized since there is no normalization to any of the vocabulary terms. This leads into the assumption that the user will query with the precise word type that they wish to search, and not use any simplified or root version. While it falls within the guidelines of this project, this is not necessarily a good design choice as recall will be very low, and since the reality of user queries being flawed in some way, the results (if any) may not always be relevant.

Subproject 2

The following section reports on the different aspects encompassing subproject 2, in which the implementation can be found in *subProject2.py*. It covers the overall search engine processing done in the backend, looking at each module involved in the processing of user entered queries.

Query Processor

The main module in the spotlight for this subproject is *queryTest()*, a newly updated version of the previously seen processing function from project 2. Firstly, there are many more parameters in play.

- ***query*:** The actual query submitted by the user for searching. It is tokenized in order to normalize it to the format of the index vocabulary terms.
- ***spimiIndex*** and ***naiveIndex*:** Both are optional parameters, set to *None* by default, and are initialized with the actual indexes required for searching, at run-time.
- ***collection***: This parameter is the original list of token lists (*tokensList* from subproject 1). It is used to get the collection size for the purpose of BM25.
- ***operation*:** Yet another optional parameter, however this is defaulted to '*OR*' in the case where a Boolean operator is not specified, the disjunction is assumed.
- ***queryTermRanking*** and ***bm25*** : These are both optional Boolean parameters, defaulted to *False*. When set, they are what flag the ranking of results, depending on which ranking parameter is set to *True.*

With all of the above parameters, the module works as follows:

1) Tokenizes the query into *queryTerms*
2) For both the *naiveIndex* and *spimiIndex* parameters, if they are initialized, the function:
    a. Determines if the query is single-term by verifying the length of *queryTerms.* In the case of single-term queries, it queries the index and returns the result.
    b. In the case of multi-term queries:
        i. The function first iterates through the *queryTerms,* appending their postings lists to a variable called *postingsLists.*
        ii. Then, depending on the *operator* (OR, AND) the function processes the query using either the *conjunction()* module or the *disjunction()* module.
        iii. Returned postings list from either *conjunction()* or *disjunction()* is stored in a variable *result*.

iv. Depending on the values of either *queryTermRank* and *bm25*, the result is ranked accordingly, by being passed as a parameter to the appropriate ranking function.
v. Result is displayed

Conjunction and Disjunction

The *conjunction()* and *disjunction()* modules simply take the *postingsLists* variable as a parameter, perform the required operations and return the result. The conjunction module uses an auxiliary *intersect()* module to perform the intersection between two postings lists, and all intermediary results are managed by the parent function. The disjunction module is also passed the *queryTermRank* parameter taken directly from the *queryTest()* parameters, in which it calls *queryTermRank()* to perform the ranking depending on the value of the parameter. Further details of this ranking module will be covered in a later subsection. The reason the function is passed directly to the *disjunction()* module is due to the project guidelines specifying the nature of the queries needed for this subproject. It mentions the need for an *OR* multi-term query to be ranked via keyword ranking. Thus, the ranking condition is handled nested within the *disjunction()* function.

Also worth mentioning, is the *convertPostingsLists()* module which is used exclusively with the spimi index. Since the spimi index carries postings lists composed of tuples instead of single docID values, both the *conjunction()* and *disjunction()* modules would require optimizations to be able to handle both postings list styles from the spimi and naïve indexes. In order to mitigate overcomplicating these two modules, the *convertPostingsLists()* function is called on the *postingsLists* from within the *queryTest()* function, in order to normalize them before being sent to *conjunction()* or *disjunction()*.

Query Term Ranking

The ranking with respect to the occurence of query terms in the documents is handled by the *queryTermRank()* module. It simply creates a dictionary called *docFreq,* and iterates through the

docIDs of the postings list parameter, adding it to the *docFreq* dictionary with a value of 1 on the first occurrence, and then incrementing the value after every occurrence of the docID in the postings list. In the end, the resulting dictionary is sorted in descending value order, and return as a list of docID-score tuples.

BM25 Ranking

In the case of *bm25* ranking being set to *True* in the *queryTest()* function, the *bm25()* function is invoked. Note that it can only be invoked when querying the SPIMI index, since of the two (SPIMI and Naïve) it is the only index that maintains term frequency. The ranking is performed on the resulting set of docIDs, returned by the Boolean operation functions *conjunction()* or *disjunction()*. However, it requires a few other parameters as well:

1- The query terms (tokenized query)
2- The spimi index
3- The collection (tokenized collection stored as list of token lists)

The function begins by obtaining the values needed for the constant parameters used in the function such as N, L_{ave}, and k1 and b which are obtained at runtime. Once it has these values, it begins iterating over every query term, appending their postings lists to a variable called *postingsLists,* retrieving the document frequency (*df*) by simply getting the length of the term's postings list, as well as the inverse document frequency (*idf*) by performing $idf = \log(N/df)$. The function then finds all docIDs within *postingsLists* that also appears in the result set of docIDs which was passed as a parameter. If so, it goes ahead to calculate the remainder of the BM25 equation, computing the score for this docID, term pair. The function maintains docIDs as keys in a local *rankedResults* dictionary, for which it increases the value (score) of every docID in the dictionary, whenever a new score is calculated for it with respect to the query term in which it is adjoined to. If the score for all docIDs are 0.0, the function returns *None*, if not, then it returns a sorted list of the docID – score tuples.

K1 and b parameters

In the context of the BM25 ranking function, the `k1` and `b` parameters serve as dials to fine-tune the impact of term frequency and document length within the scoring algorithm. For the reuters21578 collection, which is known to include brief documents comprising solely of titles, the `b` parameter has been found to be optimal close to the upper limit of its range (0<b<1). This decision was informed by the unique composition of the dataset, where some documents are 'brief' types, containing only titles. While these documents might be relevant, they potentially lack the comprehensive context provided by longer documents that include both titles and bodies. Thus, a `b` value leaning towards 1 tips the balance in favor of more extended documents by penalizing the scores of shorter ones.

On the other hand, `k1`, which is adjustable during runtime, has been found to be optimal at values close 1.0. This recognizes the importance of term frequency but also guards against overvaluing it. Repetition of a term indeed suggests topical relevance, but beyond a point, it doesn't necessarily equate to a document's capacity to fulfill an information need. For instance, multiple mentions of "drug" might not signal a document's centrality to the concept if the context is not about drugs directly but perhaps about an entity like the FDA.

The choice of these optimal values—1.0 for `k1` and 0.75 for `b`—is based on the corpus's structure and composition. These can be adjusted in real-time, allowing users to refine the search experience based on their particular search needs.

Queries

The following subsection details the different queries used for the development and testing of subproject 2.

Single-Term Queries

The single term queries used, are both the test and challenge queries from the previous project. The test queries include: *U.S.A. , it's, multi-billion.* The challenge queries include: *pineapple, Chrysler, Bundesbank.* All single term queries on the SPIMI index, return the exact same results in

comparison to the Naïve index from project 2, as can be seen in Appendix A. This is due to them being equivalent indexes, apart from the inclusion of docID-term frequency tuples in the SPIMI index. Also, apart from that slight structural difference, they differ only in the nature of their construction methods, as is elaborated on in the subsection on index characteristics.

Multi-term Queries

The resulting logs for the multi-term queries can be found in Appendix B. For developmental purposes, the Boolean operators were tested on some of the single term queries from the previous section, to ensure the proper handling of Boolean operations. As can be seen, the disjunction between *it's* and *pineapple* returns a postings list size of 150 which is correct since their single term queries return 149 documents, and 1 document, respectively. Conversely, their conjunction should return the single document from *pineapple*'s postings list, which it in fact does.

With respect to the three multi-term test queries, they were run based on the three information needs presented in the project guidelines. For the first multi-term test query, we are required to perform a multi-keyword query on subproject 1, with a conjunction between all the terms. The second multi-term query was a disjunction, with query term ranking, and the third was simply a BM25 ranking, with any Boolean operation. As can be seen from the logs, when running the exact information need as a query for any of the test queries, there are often no results (except for *George Bush)*. This is because the more query terms present in a conjunction, the smaller the result set becomes due to increased specificity. Thus, the adapted queries based on the information needs perform much better as queries, actually yielding docIDs in their returns. Furthermore, the lack of compression results in the lack of equivalence classes for terms, as well the lack of recall. Meaning, term variations are not associated through a normalized root form, and thus the retrieve the postings for any of these terms, the query must explicitly match their syntax.

In analyzing the ranked retrieval, the query term rank works as expected. Note that, this ranking is restricted to disjunctions since conjunctions only return docIDs of those documents containing all the query terms. Ranking a conjunction by query term would result in all documents containing the same score. The BM25 ranking also seems to work as expected, with the best-fit documents often returned at the top of the returns. This is most often the case with k1 and b parameters set at around 1, and at 0.75, respectively. Variations in their values bring about a different ranking of the end results.

Design Challenges and Final Thoughts

The challenges brought about by this project are numerous. In the beginning, the refinements made to the collection brought about the need for in depth consideration when setting k1 and b values for BM25 ranking. On this ranking technique, it also birthed the challenge of figuring out how to collect the required parameters needed, more specifically with regards to the term frequencies. These design choices to the structure of *spimi()* lead to the need for auxiliary functions to ensure adaptability for the differently structured postings lists. Also implementing an engine-like experience also presented its challenges, to retrieve parameters at run time and conditionally execute certain functions give specific circumstances. This project also depicted the importance of equivalence classes and compression, where without, the recall of a system can be drastically reduced. Although challenges, they shed light on the different factors to be considered when designing a search engine, from the corpus data to the ranking techniques. This details not only complex nature of these systems, but the theoretical and practical decisions that are to be made.

Appendix A

Single-Term Project 2 Test Queries:

```
====== NAIVE INDEX ======
QUERY: 'U.S.A.'
OPERATION: AND
POSTINGS LIST SIZE: 27
POSTINGS LIST: [834, 1671, 1702, 3181, 3322, 3327, 3486, 6517, 7365, 7868, 8873,
8939, 11403, 13061, 14541, 14742, 15111, 15607, 16173, 16558, 16581, 16991,
17102, 17563, 19101, 19664, 20693]

====== SPIMI INDEX ======
QUERY: 'U.S.A.'
OPERATION: AND
POSTINGS LIST SIZE: 27
POSTINGS LIST: [(834, 1), (1671, 2), (1702, 1), (3181, 1), (3322, 1), (3327, 1),
(3486, 1), (6517, 2), (7365, 1), (7868, 1), (8873, 1), (8939, 1), (11403, 1),
(13061, 1), (14541, 1), (14742, 1), (15111, 1), (15607, 1), (16173, 1), (16558,
1), (16581, 1), (16991, 1), (17102, 1), (17563, 1), (19101, 1), (19664, 1),
(20693, 1)]


====== NAIVE INDEX ======
QUERY: 'it's'
OPERATION: AND
POSTINGS LIST SIZE: 149
POSTINGS LIST: [748, 1272, 1357, 1674, 1682, 1777, 1836, 2064, 2156, 2351, 2618,
2924, 3446, 3534, 3551, …
====== SPIMI INDEX ======
QUERY: 'it's'
OPERATION: AND
POSTINGS LIST SIZE: 149
POSTINGS LIST: [(748, 1), (1272, 1), (1357, 1), (1674, 1), (1682, 1), (1777, 1),
(1836, 2), (2064, 1), (2156, 1), (2351, 2), (2618, 1), (2924, 1), (3446, 1),
(3534, 2), (3551, 1), …

====== NAIVE INDEX ======
QUERY: 'multi-billion'
OPERATION: AND
POSTINGS LIST SIZE: 6
POSTINGS LIST: [2118, 7273, 9788, 12756, 14769, 17598]

====== SPIMI INDEX ======
QUERY: 'multi-billion'
OPERATION: AND
```

```
POSTINGS LIST SIZE: 6
POSTINGS LIST: [(2118, 1), (7273, 1), (9788, 1), (12756, 1), (14769, 1), (17598,
1)]
```

Single-Term Project 2 Challenge Queries:

```
====== NAIVE INDEX ======
QUERY: 'pineapple'
OPERATION: AND
POSTINGS LIST SIZE: 1
POSTINGS LIST: [4630]

====== SPIMI INDEX ======
QUERY: 'pineapple'
OPERATION: AND
POSTINGS LIST SIZE: 1
POSTINGS LIST: [(4630, 1)]

====== NAIVE INDEX ======
QUERY: 'Chrysler'
OPERATION: AND
POSTINGS LIST SIZE: 105
POSTINGS LIST: [627, 678, 1114, 1121, 1214, 1408, 1455, 1539, 1599, 1682, 1733,
1736, 1817, 1854, 2198, …

====== SPIMI INDEX ======
QUERY: 'Chrysler'
OPERATION: AND
POSTINGS LIST SIZE: 105
POSTINGS LIST: [(627, 7), (678, 3), (1114, 2), (1121, 2), (1214, 2), (1408, 1),
(1455, 2), (1539, 4), (1599, 1), (1682, 2), (1733, 7), (1736, 3), (1817, 1),
(1854, 2), (2198, 4), …

====== NAIVE INDEX ======
QUERY: 'Bundesbank'
OPERATION: AND
POSTINGS LIST SIZE: 163
POSTINGS LIST: [278, 926, 942, 950, 1540, 1959, 1969, 1971, 2110, 2197, 2662,
2686, 2979, 3020, 3514, …

====== SPIMI INDEX ======
QUERY: 'Bundesbank'
OPERATION: AND
POSTINGS LIST SIZE: 163
```

```
POSTINGS LIST: [(278, 5), (926, 5), (942, 5), (950, 2), (1540, 11), (1959, 2),
(1969, 1), (1971, 1), (2110, 1), (2197, 2), (2662, 4), (2686, 6), (2979, 1),
(3020, 6), (3514, 6), …
```

Appendix B

Exact information need queries

```
====== NAIVE INDEX ======
QUERY: 'Democrats' welfare healthcare reform policies' - Your query returned no
results with operation: AND.

====== SPIMI INDEX ======
QUERY: 'Democrats' welfare healthcare reform policies' - Your query returned no
results with operation: AND.


====== NAIVE INDEX ======
QUERY: 'Drug company bankruptcies' - Your query returned no results with
operation: AND.

====== SPIMI INDEX ======
QUERY: 'Drug company bankruptcies' - Your query returned no results with
operation: AND.

====== NAIVE INDEX ======
QUERY: 'George Bush'
OPERATION: AND
RANKING: None
POSTINGS LIST SIZE: 12
POSTINGS LIST: [854, 965, 2796, 3560, 4008, 5405, 7525, 8593, 16780, 20719,
20860, 20891]
```

Query Term Ranking

```
====== NAIVE INDEX ======
QUERY: 'Democrats reform'
OPERATION: OR
RANKING: Query Term Ranking
POSTINGS LIST SIZE: 286
POSTINGS LIST: [(2132, 2), (9774, 2), (15369, 2), (18731, 2), (18878, 2), (19660,
2), (18, 1), (28, 1), (32, 1), (54, 1), (55, 1), (61, 1), (219, 1), (226, 1),
(234, 1), …

====== SPIMI INDEX ======
QUERY: 'Democrats reform'
OPERATION: OR
RANKING: Query Term Ranking
POSTINGS LIST SIZE: 286
```

```
POSTINGS LIST: [(2132, 2), (9774, 2), (15369, 2), (18731, 2), (18878, 2), (19660,
2), (18, 1), (28, 1), (32, 1), (54, 1), (55, 1), (61, 1), (219, 1), (226, 1),
(234, 1), …


====== NAIVE INDEX ======
QUERY: 'drug company bankruptcy'
OPERATION: OR
RANKING: Query Term Ranking
POSTINGS LIST SIZE: 4935
POSTINGS LIST: [(2127, 3), (3091, 3), (192, 2), (375, 2), (451, 2), (696, 2),
(730, 2), (902, 2), (1269, 2), (1391, 2), (1860, 2), (1891, 2), (1991, 2), (1998,
2), (2036, 2), …

====== SPIMI INDEX ======
QUERY: 'drug company bankruptcy'
OPERATION: OR
RANKING: Query Term Ranking
POSTINGS LIST SIZE: 4935
POSTINGS LIST: [(2127, 3), (3091, 3), (192, 2), (375, 2), (451, 2), (696, 2),
(730, 2), (902, 2), (1269, 2), (1391, 2), (1860, 2), (1891, 2), (1991, 2), (1998,
2), (2036, 2), …


====== NAIVE INDEX ======
QUERY: 'George Bush'
OPERATION: OR
RANKING: Query Term Ranking
POSTINGS LIST SIZE: 140
POSTINGS LIST: [(854, 2), (965, 2), (2796, 2), (3560, 2), (4008, 2), (5405, 2),
(7525, 2), (8593, 2), (16780, 2), (20719, 2), (20860, 2), (20891, 2), (28, 1),
(43, 1), …

====== SPIMI INDEX ======
QUERY: 'George Bush'
OPERATION: OR
RANKING: Query Term Ranking
POSTINGS LIST SIZE: 140
POSTINGS LIST: [(854, 2), (965, 2), (2796, 2), (3560, 2), (4008, 2), (5405, 2),
(7525, 2), (8593, 2), (16780, 2), (20719, 2), (20860, 2), (20891, 2), (28, 1),
(43, 1), …
```

BM25 Ranking

```
====== NAIVE INDEX ======
QUERY: 'Democrats reform'
OPERATION: AND
RANKING: None
POSTINGS LIST SIZE: 6
POSTINGS LIST: [2132, 9774, 15369, 18731, 18878, 19660]

====== SPIMI INDEX ======
Enter your k1 value (k1 >= 0): 1
```

```
Enter your b value (0 <= b <= 1): .75
QUERY: 'Democrats reform'
OPERATION: AND
RANKING: BM25
POSTINGS LIST SIZE: 6
POSTINGS LIST: [(19660, 16.332542207235964), (18731, 16.32483479930248), (18878,
14.14110532791878), (2132, 14.026105986364097), (9774, 11.718115800687656),
(15369, 8.058009221290343)]
====== NAIVE INDEX ======
QUERY: 'drug bankruptcy'
OPERATION: AND
RANKING: None
POSTINGS LIST SIZE: 2
POSTINGS LIST: [2127, 3091]

====== SPIMI INDEX ======
Enter your k1 value (k1 >= 0): 1
Enter your b value (0 <= b <= 1): .75
QUERY: 'drug bankruptcy'
OPERATION: AND
RANKING: BM25
POSTINGS LIST SIZE: 2
POSTINGS LIST: [(3091, 14.996104430218104), (2127, 7.28016768542342)]


====== SPIMI INDEX ======
Enter your k1 value (k1 >= 0): 1
Enter your b value (0 <= b <= 1): .75
QUERY: 'George Bush'
OPERATION: AND
RANKING: BM25
POSTINGS LIST SIZE: 12
POSTINGS LIST: [(7525, 20.055101003864856), (4008, 18.158554129889595), (20719,
17.345725219013254), (20891, 14.133981750410236), (16780, 14.10978959976576),
(854, 13.907038836958256), (3560, 13.18275321833339), (2796, 12.759583008796202),
(965, 12.668105090715867), (5405, 12.391464474047364), (8593,
11.557844595114926), (20860, 10.999784745408308)]
```