Roll Your Own Mini Search Engine

Date: 2024-10-6

Chapter 1: Introduction

Utilize a web scraper to read the complete works of Shakespeare, analyze the text line by line, filter the content (such as removing stop words and identifying word stems), and then count the occurrences of each word along with the text name, ID, and frequency of occurrence. Establish an inverted index structure using a hash table. Create an interface for user input to complete the search for individual words and phrases within the texts. Finally, adjust the query conditions and analyze the changes in the results.

Chapter 2: Data Structure

1. doc_count Structure

```
typedef struct {

int doc_id;

int count;

char filename[MAX_FILENAME_LEN];

} doc_count;
```

- **Members**:
    - doc_id: A unique identifier for a document, used to distinguish between different documents.
    - count: The number of times a specific word appears in that document, indicating the importance or frequency of the keyword within the document.
    - filename: Stores the name of the document file, which helps in locating and referencing the document later.

2. inverted_index_entry Structure

```
typedef struct inverted_index_entry {

char word[MAX_WORD_LEN];

int doc_num;

doc_count doc_list[MAX_DOCS];
```

```
struct inverted_index_entry *next;

}inverted_index_entry;
```

- **Members**:
  - word: Stores the keyword (word) that is part of the inverted index.
  - doc_num: Stores the number of documents associated with this keyword, facilitating quick access to relevant information.
  - doc_list: An array that stores the documents related to the keyword along with their occurrence details, defined as doc_count.
  - next: A pointer to the next inverted_index_entry, allowing the structure to function as a linked list. This enables dynamic expansion of the index and handles collisions in the hash table.

3. inverted_index Structure

```
typedef struct {

int size;

int capacity;

inverted_index_entry **table;

} inverted_index;
```

- **Members**:
  - size: The current number of stored keywords, useful for management and statistics.
  - capacity: The total capacity of the hash table, representing the maximum number of keywords it can hold. This helps determine when the hash table needs to be expanded.
  - table: An array of pointers to inverted_index_entry, where the actual inverted index entries are stored. It allows quick access to specific index entries based on a hash function.

DJB2

```
    hash = 5381 for each character c in the string: hash = ((hash << 5) + hash) + ord(c) //

hash * 33 + c return hash
```

Advantages

- **Simplicity and Ease of Implementation:**
  The algorithm has a simple structure, making it easy to understand and implement, which is suitable for rapid development.
- **Good Performance:**
  In most cases, DJB2 can quickly compute hash values and is suitable for handling large amounts of data.
- **Low Collision Rate:**
  For general strings, DJB2 has a relatively low collision rate, especially performing well on random datasets.
  Disadvantages
- **Collision Issues:**
  Although the collision rate is low, it can still occur, particularly when dealing with specific patterns or similar strings.
- **Input Characteristics Affect Performance:**
  For certain specific types of input (such as repeated characters or specific patterns), hashing performance may degrade.

Chapter 3: Algorithm Analysis

● Error Analysis

1. Irregular Verbs:

Some irregular verbs(e.g. "went" or "saw") are not stemmed properly. The algorithm primarily focused on suffixes, and since these words don't conform to standard morphological rules, they may remain unchanged or become inappropriate stems.

2. Compound Words:

Words like "toothbrush" or "mother-in-law" are treated as single entities, which may lead to incorrect stemming since the algorithm does not account for compound structures.

3. Proper Nouns:

The algorithm does not differentiate between common and proper nouns. For instance, "Smith" might be treated like "smite", resulting in inappropriate stemming for names.

4. Non-Standard Conjugations:

Variants like "running," "jogging," or even colloquial forms(e.g. "gonna") may not stem accurately. The algorithm could misinterpret these forms because it relies heavily on fixed suffix patterns.

5. Words with Multiple Meanings:

Homographs(words that are spelled the same but have different meanings, such as "lead" as a verb vs. "lead" as a noun) might be mis-stemmed since the algorithm does not consider context.

6. Loanwords and Foreign Terms:

Words borrowed from other languages may not conform to English morphological rules, leading to incorrect stemming. For instance, "cafe" or "fiance" could be incorrectly altered.

7. Exeption Handling:

There is no mechanism for exceptions to rules. Words that defy conventional endings, like "sphinx", might not stem as intended because they don't fit the standard patterns.

8. Edge Cases with Suffix Removal:

The handling of suffixes in some cases could lead to ambiguous stems.Far example, the removal of "ed" from "hopped" results in "hop", but in "hoped", it might be incorrectly handled if the measure is miscalculated.

9. Phonetic Varations:

The algorithm does not account for phonetic variations, such as regional dialects(e.g. "color" vs. "colour"), which could lead to different stemming results in practice.

10. Handling of Special Characters:

While there is some basic punctuation removal, the algorithm may not handle hypens or other special characters effectively, leading to unintended stems.

- Document Name Handling

When adding a document, if the document name contains special characters (such as spaces, slashes, etc.), it may lead to file path issues or errors in string comparison.

```
    add_inverted_index_entry(index, "word", 1, "path/to/document with spaces.txt"); //
may lead to wrong documents
```

- Hash Collision

If two different words produce the same index value when processed by a hash function (i.e., a hash collision), it may result in a long linked list, thereby reducing search efficiency. For example, the words "apple" and "banana" might be hashed to the same index position, causing them to share the same linked list.

```
    suppose that hash("apple") % capacity == hash("banana") % capacity
```

- Concurrent Access

If multiple threads modify the index simultaneously without a synchronization mechanism, it may lead to data races and inconsistent states.

```
    // Multiple threads calling add_inverted_index_entry simultaneously may lead to data
corruption.
```

- Long String Processing

For very long strings, DJB2 may become inefficient, especially as the number of iterations in the hash value computation increases significantly.

```
    // Long strings like "this is a very long string that continues and continues..." take a
considerable amount of time to compute the hash.
```

Chapter 4: Testing Results

Table of test cases. Each test case usually consists of a brief description of the purpose of this case, the expected result, the actual behavior of your program, the possible cause of a bug if your program does not function as expected, and the current status (“*pass*”, or “*corrected*”, or “*pending*”).

- ***Data Generation***

Generating the ground truth data for test is an important and precise work. To fully test the correctness of the results of our Mini Search Engine, we try to make the data more diverse and robust as much as possible.

1. Firstly we try to find and search 100 single word queries and 100 two-word queries with relatively high frequency in Shakespeare's works, where the 100 two-word queries represent the test for multiple-word queries.

2. Secondly, we write a .sh script to utilize our command line to search for all the documents containing the specific word or two-word phrase. (Notice: for the phrase queries, the returned documents are those containing both the two words instead of regarding the phrase as a whole). You can see the file `GroundTruthDataMaking.sh` in the submitted files.

3. Finally, we curate these generated data in a specific .json format so that our test program can read these data smoothly and make precise comparisons to test the correctness. You can see the file `search_output.json` in the submitted files.

- ***Query Type and Threshold Setting***

To achieve the requirement of "Run tests to show how the thresholds on query may affect the results.", we set totally 6 different query types and 7 different thresholds to explore the effects of thresholds on results. They are respectively:

- *Query Types:*
  - Query the documents with top K highest frequency
  - Query the documents with top K lowest frequency
  - Query the documents with frequency higher than a threshold K
  - Query the documents with frequency lower than a threshold K
  - Query the documents with top K% highest frequency
  - Query the documents with top K% lowest frequency
- *Thresholds:*
  - 1
  - 5

- 10
- 25
- 50
- 75
- 100

➢ *Results*

- ***Roughly Analysis***

Based on the setting of different query types and thresholds, we have totally 42 results. It's worth to mention that our results consist of two dimensions: **precision and recall.** For each of them, we curate the results in a specific .csv file, which is more clear for comparion. Because the csv file is too long and too numerous, so we take only a part of a file `results_with_query_type_1_threshold_100.csv` as an rough example analysis. You can see the detailed csv data in the `results/csv` directory.

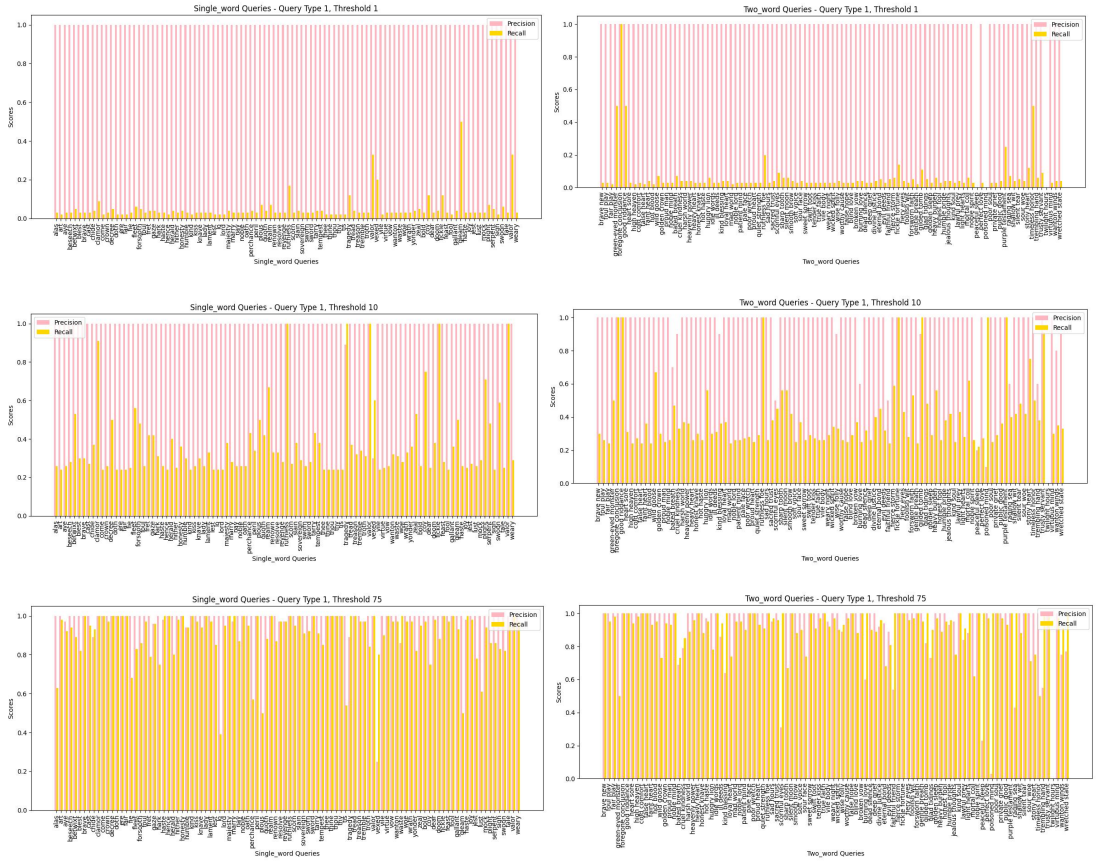
query	expected doc_ids	expected doc_num	returned doc_ids	returned doc_num	precision	recall
alas	Othello, the Moore of Venice	38	Othello, the Moore of Venice	24	1	0.63
	The Merry Wives of Windsor		Troilus and Cressida			
	Winter's Tale		Twelfth Night			
	As You Like It		The Merry Wives of Windsor			
	Cymbeline		The Merchant of Venice			
	The Rape of Lucrece		The Life and Death of Richard the Third			
	The Tragedy of Hamlet, Prince of Denmark		Winter's Tale			
	Troilus and Cressida		Romeo and Juliet			
	All's Well That Ends Well		The Tragedy of Macbeth			
	Loves Labours Lost		Titus Andronicus			
	Venus and Adonis		Much Ado About Nothing			
	The Tempest		The Second part of King Henry the Fourth			
	The Life and Death of Julius Caesar		The Third part of King Henry the Sixth			
	Much Ado About Nothing		Cymbeline			
	King Lear		The Life and Death of Julius Caesar			
	The Life and Death of Richard the Third		Measure for Measure			
	Twelfth Night		King Lear			
	Two Gentlemen of Verona		The First part of King Henry the Fourth			
	The Tragedy of Macbeth		Timon of Athens			
	The First part of King Henry the Fourth		The Life and Death of Richard the Second			
	The Life and Death of Richard the Second		The Life of King Henry the Eighth			
	Titus Andronicus		The Tragedy of Hamlet, Prince of Denmark			
	The Life and Death of King John		The Rape of Lucrece			
	Othello, the Moore of Venice		The First part of King Henry the Sixth			
	The Tragedy of Coriolanus					
	The Second part of King Henry the Fourth					
	The Third part of King Henry the Sixth					
	Timon of Athens					
	The Second part of King Henry the Sixth					
	The Merchant of Venice					
	The Life of King Henry the Eighth					
	Antony and Cleopatra					
	The Comedy of Errors					
	Romeo and Juliet					
	Pericles, Prince of Tyre					
	Measure for Measure					
	The Taming of the Shrew					
	The First part of King Henry the Sixth					

We can see that when the query type is 1 and threshold is 100, which means "Query the documents with top 100 highest frequency", the precision is 1 and the recall is 0.63. It proves that all the retrieved documents are correct, but there are still a lot of correct documents haven't been retrieved.

- ***Visualization and Detailed Analysis***

To get the more general and robust conclusion about our results in different query types and thresholds, we use the Python visualization method to draw the bar charts (You can see them in the `results/visualization` directory. There are totally 84 bar charts because we separate the 100 single word queries and 100 two-word queries in two different bar charts.

1. Query the documents with top K highest frequency

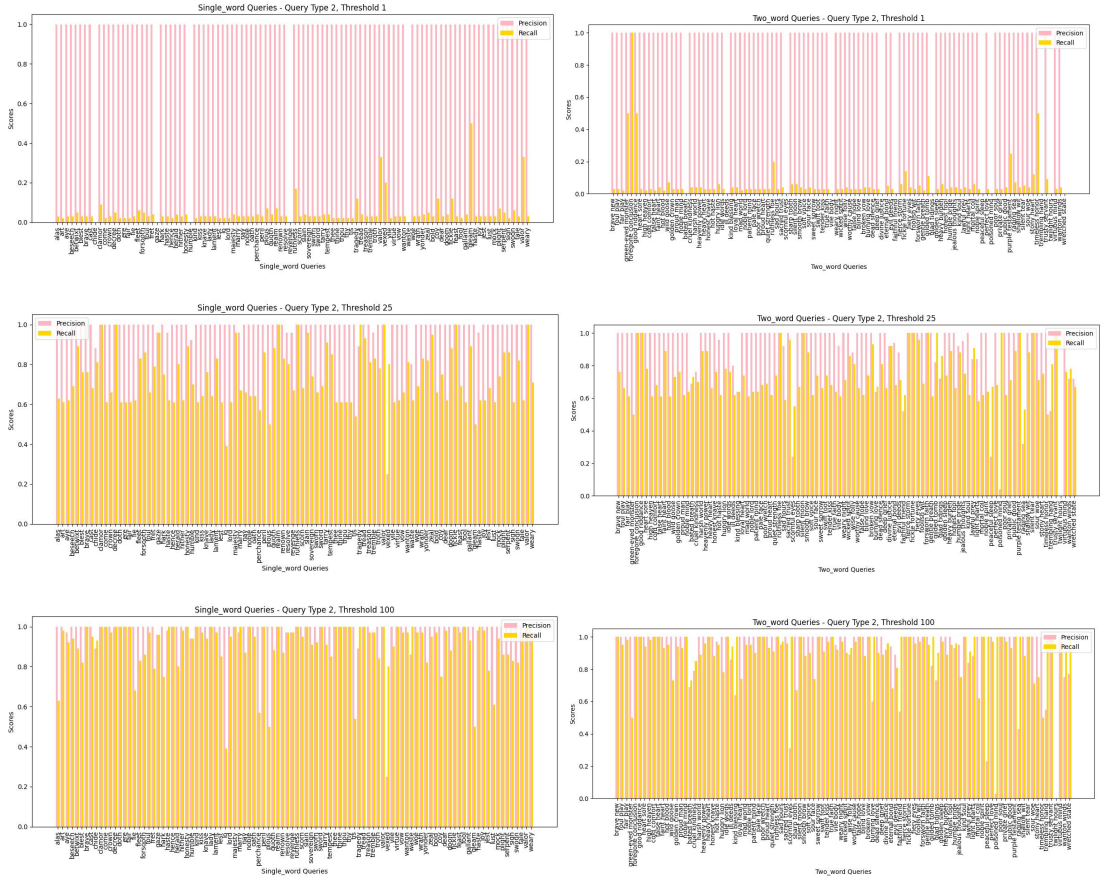


- For **precision**: As we observe from the bar charts, the lower the threshold, the higher the precision becomes. This is expected because, with a lower threshold, the search engine retrieves fewer documents, which decreases the likelihood of retrieving irrelevant or wrong documents. As the threshold increases, the precision gradually drops, which is a clear sign that, as more documents are retrieved, the probability of pulling in incorrect results naturally increases. Therefore, keeping the threshold low ensures a higher precision by filtering out noise in the results.

- For **recall**: The trend in recall is quite the opposite of precision. As the threshold increases, the recall also increases. This makes perfect sense because, with higher thresholds, the engine retrieves a broader set of documents, and more relevant documents are captured. This phenomenon is particularly noticeable as we observe higher recall values at increased thresholds. The more documents retrieved, the more relevant results are included, thus improving recall.

- For **one-word and two-word query**, Actually the distribution of the precision and recall of these two types of queries are almost the same, which proves the robustness for no matter one-word or two-word of our Mini Search Engine.

2. Query the documents with top K lowest frequency

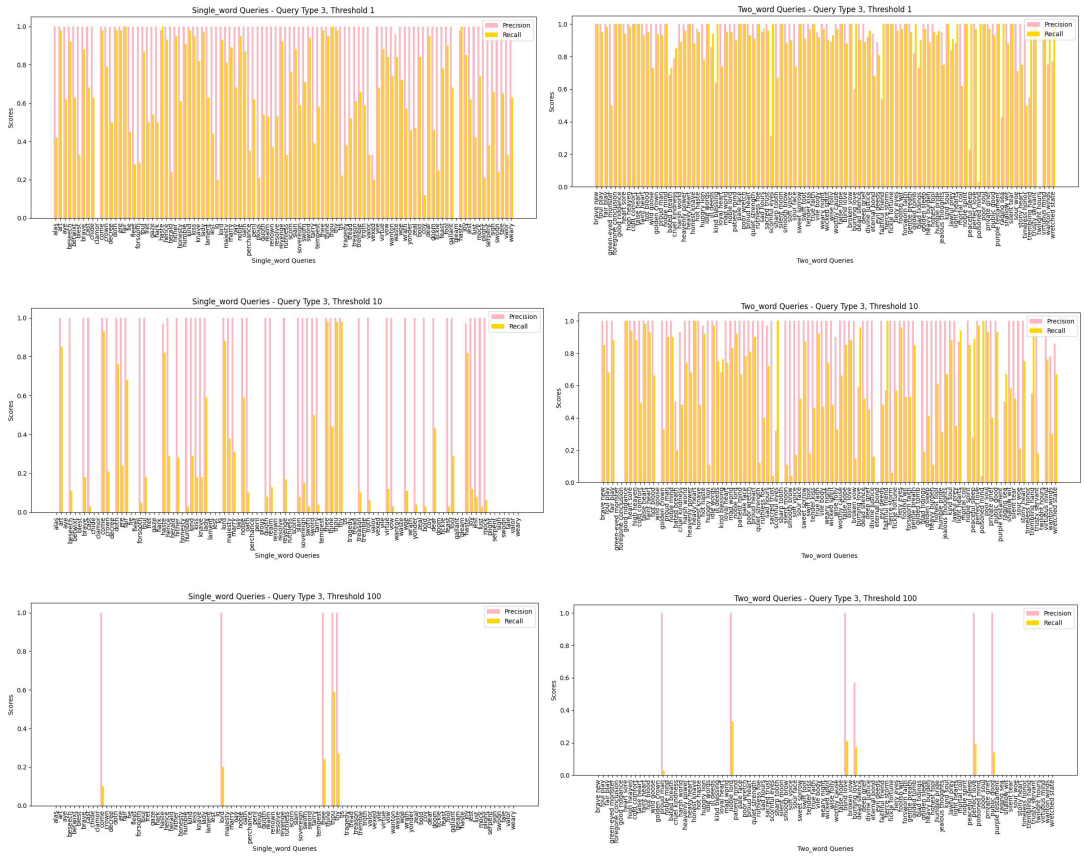


- For **precision**: As observed from the bar charts, the precision values remain relatively high across most of the queries. However, for queries with a lower threshold, precision tends to fluctuate more, especially for certain two-word queries. This suggests that when retrieving documents with low frequency, the precision varies, and it is more likely to retrieve incorrect documents as the threshold increases. The influence of query frequency is evident because less frequent queries tend to match fewer documents, leading to an increased chance of precision dropping when the threshold increases. This indicates that precision, in general, is sensitive to the sparsity of the retrieved documents, but stabilizes at higher thresholds.

- For **recall**: We can clearly see that the recall is more consistent across different thresholds. As the thresholds increase, the recall shows a significant improvement. This indicates that as the model retrieves more documents at relatively higher frequencies, the overall recall becomes better. This is likely due to the broader document pool being considered at higher thresholds, resulting in a higher number of relevant documents being retrieved. However, the disparity between precision and recall in some cases highlights the challenge of balancing these two metrics when querying documents of lower frequency.

- For **single-word and two-word query**: The distribution of results in both single-word and two-word queries shows that two-word queries tend to have a more consistent recall. Single-word queries, on the other hand, show a more volatile precision across thresholds. The two-word queries seem to handle lower frequency data better, with more balanced precision-recall values. This may suggest that when working with lower frequency queries, the addition of a second word provides more context, helping the search engine retrieve more relevant documents.

3. Query the documents with frequency higher than a threshold K

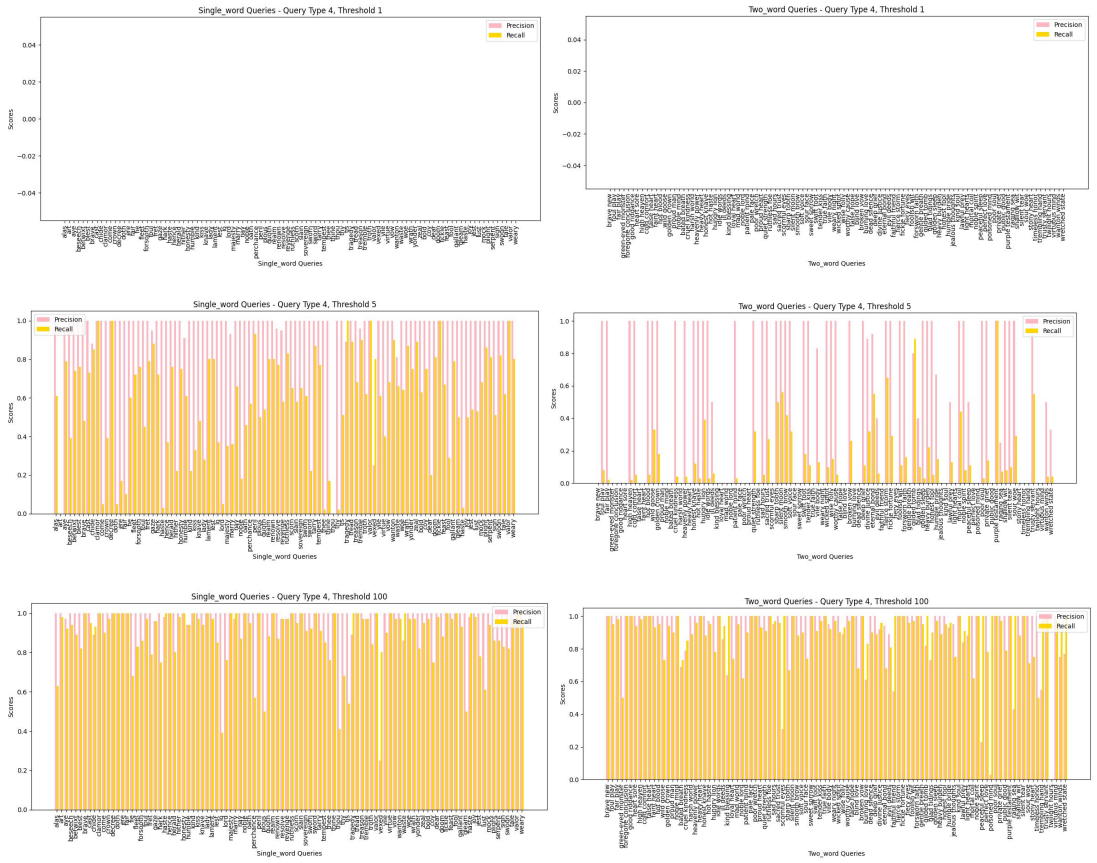


- For **precision**: Precision shows considerable fluctuations, especially at lower thresholds such as Threshold 1. The precision values are relatively high across most queries, but we observe that precision deteriorates for certain queries, indicating that higher document frequencies lead to a larger pool of irrelevant documents being retrieved. As the threshold increases to Threshold 10 and Threshold 100, the precision becomes more sparse, which is most noticeable in two-word queries, where many precision values drop significantly. This reveals that handling higher frequency documents is more challenging in maintaining precision, especially for complex queries.

- For **recall**: Recall is stable across thresholds and performs notably better than precision, particularly at Threshold 1 and Threshold 10. For most queries, the recall approaches 1.0, reflecting that a higher proportion of relevant documents are retrieved when considering more frequent documents. However, recall becomes more varied as the threshold rises to Threshold 100, particularly for single-word queries, where gaps in recall scores become evident. This suggests that for very high-frequency documents, the system struggles to maintain consistent recall performance.

- For **one-word and two-word query**: Both one-word and two-word queries display similar trends in precision and recall. However, two-word queries exhibit more precision issues at higher thresholds, likely due to the added complexity of matching multiple terms. Despite this, recall is generally strong across both query types, indicating the system's ability to retrieve relevant documents even when handling more complex queries. The main challenge lies in balancing precision, which becomes more volatile as the frequency threshold increases.

4. Query the documents with frequency lower than a threshold K

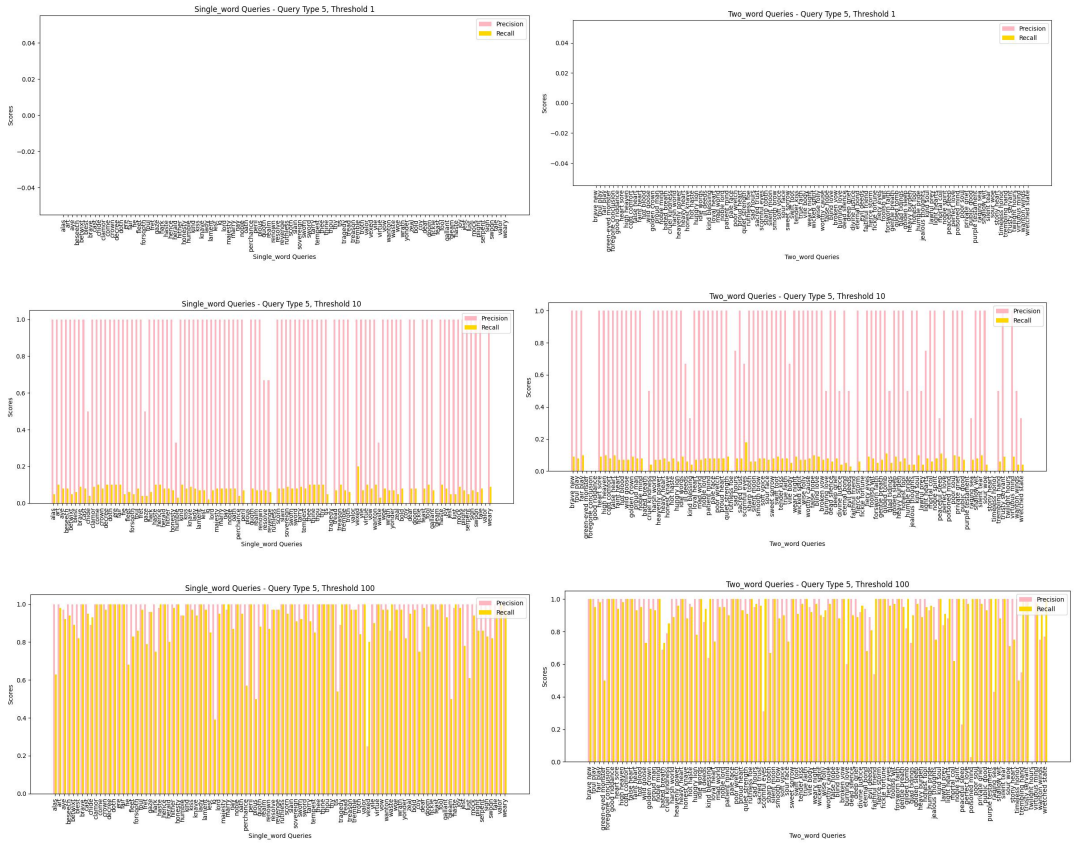


- For **precision**: At Threshold 1, precision values are largely absent, indicating that the system is unable to retrieve relevant documents at very low thresholds for both single-word and two-word queries. As the threshold increases to Threshold 5, the precision improves for single-word queries but remains inconsistent for two-word queries, where it shows considerable fluctuation. By the time we reach Threshold 100, precision stabilizes, particularly for single-word queries, where it consistently reaches high values across most queries. This suggests that the system can maintain better precision when handling documents with lower frequencies but performs less predictably for more complex, two-word queries.

- For **recall**: Recall demonstrates a much clearer trend. At Threshold 5 and especially Threshold 100, recall values increase significantly, nearing 1.0 for most queries, especially for single-word queries. This shows that the system is able to capture more relevant documents as the threshold increases. However, two-word queries still display more variability, with recall not reaching the same level of consistency as seen in single-word queries, particularly at Threshold 5, where recall values scatter across a broader range.

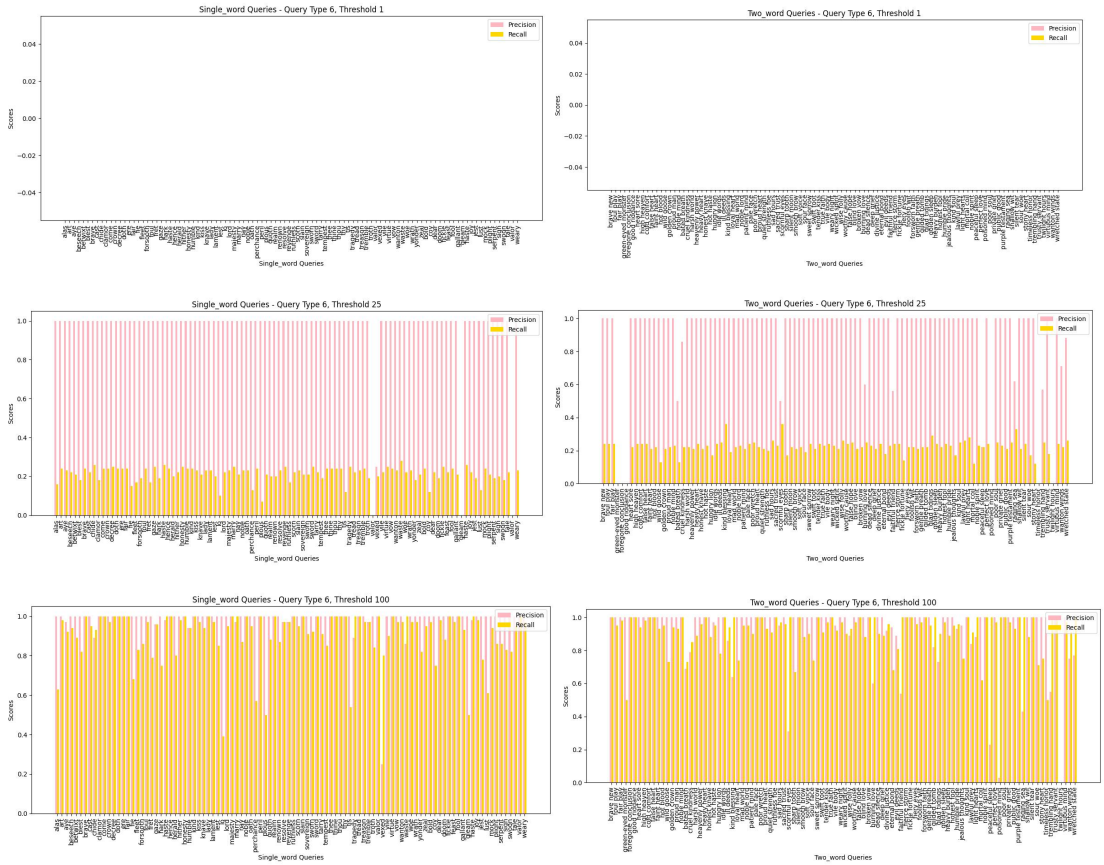
- For **one-word and two-word query**: Single-word queries display greater consistency in both precision and recall as the threshold increases, particularly at Threshold 100, where performance stabilizes at high levels. Two-word queries, on the other hand, exhibit more volatility in both precision and recall, especially at lower thresholds. This suggests that querying with two words introduces more complexity, making it harder for the system to maintain the same level of precision and recall as it does with single-word queries, especially when dealing with documents of lower frequency.

5. Query the documents with top K% highest frequency



- For **precision**: At Threshold 1, the precision values are mostly absent, indicating that the system struggles to return relevant results at such a low threshold for both single-word and two-word queries. At Threshold 10, there is significant improvement in precision for both query types, but still, two-word queries exhibit more fluctuation compared to single-word queries. As the threshold increases to Threshold 100, precision stabilizes with higher values, especially for single-word queries, which indicates that with higher document frequency, the system can filter out irrelevant results more effectively.
- For **recall**: Recall shows a steady increase as the threshold increases. At Threshold 10 and Threshold 100, recall values are high across both single-word and two-word queries, often approaching 1.0. This shows that as the system retrieves more documents, it captures most of the relevant ones. However, recall tends to fluctuate more for two-word queries at lower thresholds, suggesting that higher document frequency is necessary for stable recall performance in more complex queries.
- For **one-word and two-word query**: The precision-recall balance for both single-word and two-word queries is quite similar, especially at higher thresholds. However, two-word queries continue to display more variance in precision at lower thresholds, indicating that retrieving relevant results for more complex queries can be challenging when document frequency is lower. The consistency in recall across both query types highlights the robustness of the system in retrieving relevant results when dealing with high-frequency documents, but precision remains a more challenging metric for two-word queries.

6. Query the documents with top K% lowest frequency



- For **precision**: At Threshold 1, precision values are absent, showing the difficulty of retrieving relevant results when documents are infrequent. Moving to Threshold 25 and Threshold 100, precision increases significantly for both single-word and two-word queries, though two-word queries display more fluctuations across the board. This indicates that the system is better equipped to handle lower frequency documents at higher thresholds but still struggles with maintaining precision when dealing with two-word queries.
- For **recall**: Recall shows more consistency across all thresholds, with values significantly improving at Threshold 25 and Threshold 100. For single-word queries, recall tends to stabilize near 1.0, while two-word queries, although showing improvements, still exhibit greater variability, especially at Threshold 25. This suggests that for lower-frequency documents, the system can retrieve a larger number of relevant results as the threshold increases, but two-word queries remain more susceptible to recall fluctuations at lower thresholds.
- For **one-word and two-word query**: Single-word queries tend to maintain more consistent precision and recall as the threshold increases, particularly at Threshold 100, where both metrics stabilize. In contrast, two-word queries display more instability in both precision and recall, especially at lower thresholds, indicating that the added complexity of two-word queries makes them more sensitive to frequency variations. This highlights the need for higher thresholds to achieve stable performance when querying documents with lower frequency, especially for two-word queries.

➢ *Cause Analysis*

Based on these detailed and practical analyses, we can conclude

following multiple and diverse cause of bugs if the program does not function as expected.

- **Data Imbalance**: Failing to retrieve relevant results at very low thresholds. If certain queries result in no matches, this may indicate that the data set you're working with is either too sparse or lacks sufficient content at these lower thresholds. The system may be failing to process the query due to missing or insufficient data for that frequency range.
- **Over-Retrieval**: Retrieving too many irrelevant documents at higher thresholds. At higher thresholds, your system is retrieving more documents than necessary, including irrelevant ones. This may be caused by insufficient filtering or an overly aggressive retrieval algorithm. The essence of this reason is mainly the stemming function, which can not make the stemming word perfect as we want. So retrieving irrelevant but very similar words is inevitable.
- **Stop Word Sensitivity**: Common words may be polluting the results, reducing precision. If our system doesn't effectively filter out stop words or low-value terms, it could be retrieving irrelevant documents based on these noisy words, leading to precision issues.
- **Query-Threshold Calibration**: The interaction between query type and threshold needs better balancing to maintain both precision and recall.

Chapter 5: Analysis and Comments

(1) Run a word count over the Shakespeare set and try to identify the stop words (also called the noisy words) – How and where do you draw the line between "interesting" and "noisy" words?

1. isConsonant(int index)

Time Complexity: O(1) since it involves a fixed number of comparisons.
Space Complexity: O(1) as it uses a constant amount of space.

2.getMeasure()

Time Complexity: O(n) where n is the length of the substring from k0 to j due to the while loops that traverse the string.

Space Complexity: O(1) since it uses constant amount of additional space.

3.vowelInStem()

Time Complexity: O(n) where n is the length from k0 to j because it iterates through the string.
_Space Complexity: _ O(1) since it only uses a few variables.

4.isDoubleConsonant(int index)

Time Complexity: O(1) as it performs a constant number of checks.
Space Complexity: O(1) due to the use of a fixed number of variables.

5.cvc(int index)

Time Complexity: O(1) since it performs a fixed number of checks.
Space Complexity: O(1)

6.ends(const char *value)

Time Complexity: O(m) where m is the length of the value string being checked. This is due to memcmp which can compare up to m characters.
Space Complexity: O(1) as it uses a constant amount of additional space.

7.setTo(const char *value)

Time Complexity: O(m) where m is the length of the new suffix being set(determined by value[0]).
Space Complexity: O(1) since it modifies the string in place.

8.replace(const char *value)

Time Complexity: O(n) due to the call to getMeasure() which is O(n).
Space Complexity: O(1)

9.step1ab()

Time Complexity: O(n) due to calls to ends() and vowelInStem(), each of which can run in O(n).
Space Complexity: O(1)

10.step1c()

Time Complexity: O(n) due to the call to vowelInStem().
Space Complexity: O(1)

11.step2()

Time Complexity: O(n) due to the calls to ends() which can take O(n).
Space Complexity: O(1)

12.step3()

Time Complexity: O(n) due to the calls to ends().
Space Complexity: O(1)

13.step4()

Time Complexity: O(n) due to the calls to ends().
Space Complexity: O(1)

14.step5()

Time Complexity: O(n) because of the calls to getMeasure() and cvc(), both of which can run in O(n).
Space Complexity: O(1)

15.removePunctuation(char *p, int *start, int *end)

Time Complexity: O(n) where n is the length of the input string due to the loop that processes each character.
Space Complexity: O(1) as it uses a constant number of variables.

16.stem(char *p, int index, int position)

Time Complexity: Overall O(n) due to the calls to removePunctuation() and other steps ehich could potentially reach O(n).
Space Complexity: O(1)

Overall Complexity

Time Complexity: The overall time complexity is O(n), where n is the length of the input string. This is because the most time-consuming operations in the stemming process involve iterating through the string and checking for conditions.
Space Complexity: The overall space complexity the algorithm operates in place and only requires a constant amount of extra space.

(2) Create your inverted index over the Shakespeare set with word stemming. The stop words identified in part (1) must not be included.

Time Complexity

- **Adding Entries (add_inverted_index_entry):**
    - Average case: O(1) (due to the properties of the hash table), worst case: O(n) (when many collisions occur).
- **Building the Index (build_inverted_index):**
    - O(m * k), where m is the number of files and k is the number of words in each file. Each word requires a call to the add entry function.

- **Rehashing(expand_inverted_index):**
    - If there are n elements in the original hash table, this step takes O(n) time.Considering the trigger resizing is 0.7 which means that for every 0.7*capacity words the table rehashed once,the amoritized time complexity is (O(1)).
- **Finding Entries (find_inverted_index_entry):**
    - Average case: O(1), worst case: O(n) (again due to collisions).

Space Complexity

- **Inverted Index Storage:**
    - The space complexity for the inverted index is O(n), where n is the number of unique words. If each word can associate with up to MAX_DOCS documents, the additional space complexity is O(n * MAX_DOCS).
- **Hash Table:**
    - The size of the hash table is capacity, occupying O(c), where c is the capacity of the hash table.
- **Overall:**

The total space complexity is O(n + c), where in the worst case, n and c may be very close.

(3) Write a query program on top of your inverted file index, which will accept a user-specified word (or phrase) and return the IDs of the documents that contain that word.

- function ReceiveUserInput:
    - If each stem() function takes O(m) time,for which m represents the averige length of string mark, and k marks in total, its time coplexity will be *O(km)*.
      In total, the **time complexity** is O(max(k \cdot m,n))
      and the **space complexity** is O(n+k \cdot m)
      for which k is the amount of pointers and m is the length of strings.
- function find_common_documents:
    - **Outer loop**:The outer loop iterates over the first document collection DocumentIDs1, which has a complexity of *O(N)*, where *N* is the size of

DocumentIDs1->doc_num.

- **Inner Loop:** For each document in the outer loop, it searches for matches in the second document collection DocumentIDs2. This loop has a complexity of *O(M)*, where *M* is the size of DocumentIDs2->doc_num.
- **Comparison Operation:** Within the inner loop, the strcmp function is used to compare filenames. The worst-case time complexity of strcmp is *O(K)*, where *K* is the length of the filenames. However, we can consider this operation as constant time *O(1)*, especially when the filename lengths are relatively short and fixed.
  Therefore, considering all factors, the overall **time complexity** of the function is: O(N*M*K)
  In practical scenarios, if we focus only on the overhead of filename comparisons, we can simplify it to: O(N*M)
  Apart from the input parameters, the function internally creates an array common_docs to store the common documents. The size of this array is MAX_DOCS, which is a constant, so the **space complexity** is O(MAX_DOCS)

- function process_query:

  - Assuming there are *k* words in the query, each word processed by process_single_word_query will produce a list of document IDs.
  - The time complexity for processing each word depends on the implementation of process_single_word_query, which we will assume to be *O(m)*, where *m* is the number of documents that match the current word.
  - The time complexity of find_common_documents is *O(pq)*, where *p* and *q* are the sizes of the document lists for the previous and current words, respectively.
    Therefore, the overall time complexity is roughly:O(k * (m + p * q))The use of query_copy has a size of (O(MAX_INPUT_SIZE)). Therefore, the overall space complexity is O(MAX_INPUT_SIZE).

- function Query:

1. **Loop through the Inverted Index:**
   - The function iterates over the InvertedFileIndex->size, which represents the number of entries in the inverted index.
   - In the worst case, it will check each entry to see if it matches the input word.
   - Therefore, the time complexity for this part is *O(n)*, where *n* is the size of the inverted index.
2. **String Comparison:**
   - The strcmp function is called for each entry being checked. The time complexity of strcmp is *O(m)*, where *m* is the length of the longest string being compared.
   - Since this comparison occurs inside a loop that runs *n* times, the total contribution from this part is *O(n * m)*.
3. **Overall Time Complexity:**
   - Combining these, the overall time complexity of the Query function is: O(n *m)
   - Here, *n* is the number of entries in the inverted index and *m* is the average length of the words being compared.

- function query_threshold_TopN_highest_results:

Process the user's query and return the document IDs related to the query, sorted in descending order by frequency.

1. **Processing the Query**:
   - process_query(words,InvertedFileIndex,DocumentIDs),*O(k(m+pq))*
2. **Sorting Document IDs**:
   - quickSort_descending_order(DocumentIDs->doc_list,0,DocumentIDs->doc_num-1);
   - The average time complexity of quicksort is (O(n \log n)), where (n) is the size of DocumentIDs->doc_num. In the worst case, it can be (O(n^2)), but in practical applications, it usually maintains an average of (O(n \log n)).
3. **Limiting the Number of Results**:
   - DocumentIDs->doc_num=Min(threshold,DocumentIDs->doc_num)

;

- This line has a time complexity of (O(1)).
  Considering the above parts, if we view the query processing complexity as *O(k(m+pq))*, the overall time complexity can be expressed as: O(k * (m+p * q) + n log n)

- function query_threshold_TopN_lowest_results:

  - the same as the upper one

- function query_threshold_minimal_frequency:

1. **Query Processing**:
   - The time complexity of process_query is O(m + d) , where (m) is the number of query terms and d is the number of matching documents.
2. **Sorting Document IDs**:
   - The time complexity of quickSort_descending_order is O(n \log n) , where n is the number of documents in DocumentIDs->doc_num. In the worst case, quicksort can reach O(n^2), but typically it is O(n \log n) on average.
3. **Filtering Results**:
   - The loop that iterates through DocumentIDs->doc_num has a time complexity of O(n).
     The overall **time complexity** can be expressed as O(m + d + n log n + n) = O(m + d + n log n)
     The **space complexity** may reach *O(n)*, where n is the number of found documents.

- function query_threshold_maximal_percentage_frequency:

  - almost the same as the upper one,time complexity *O(m+d+nlogn)*, space complexity *O(n)*

- function query_threshold_minimal_percentage_frequency:

  - the same as the upper one

- function format_filename:

  - extract the filename from a given file path and, if necessary, remove the file extension (if it is .txt)
  - **time complexity** *O(n)*, **space complexity** *O(1)*

- function print_documents:

1. **Single Word Case (word_count == 1)**:

- Check if DocumentIDs[0].doc_num is 0:

- If it is 0, print "No documents found."

- Otherwise, iterate through DocumentIDs[0].doc_list, format the filename, andprint each document's ID, word, and count.

2. **Multiple Words Case (word_count > 1)**:

   - Initialize final_common to 0 to track if common documents are found.
   - For each document in DocumentIDs[0].doc_list, check if it exists in all other words' document lists:
     - Use nested loops to check each subsequent word's document list one by one.
     - If the current document is not found in any list, set common to 0 and exit the loop.
   - If common is 1, format the filename and print the document's ID, along with its counts for each word.

  - Output Common Documents Check:
    - If final_common remains 0, it means no common documents were found; print "No common documents found."
  - End Output:
    - Print a divider line and a prompt to continue querying.

Iterating through DocumentIDs[0].doc_list gives a **time complexity** of O(m) for **single word case**, where *m* is the length of that document list.And O((word_count - 1)* n) .
for **multiple words case**.The outer loop iterates through DocumentIDs[0].doc_list, resulting in a time complexity of

O(m).The inner loop checks word_count – 1 other words, each iterating through its document list, with an average length of *n*.
and for **space complexity** is always *O(1)* for a few local variables.

- function swap:
  - *O(1)* for both time complexity and space complexity
- function load_expected_results:
  - **Opening the File and Reading Content**:
    - The fopen and fread operations have a constant time complexity of O(1), but reading the entire file content in the worst case is O(n), where n is the size of the file.
  - **JSON Parsing**:
    - The json_tokener_parse function typically has a time complexity of O(m), where m is the length of the JSON string (equivalent to the file size).
    - Operationslike json_object_array_length and json_object_array_get_idx generally have complexities of O(1) and O(k), respectively, where k is the number of elements in the array.
  - **Parsing One-Word Queries**:
    - For one-word queries, the loop runs for one_word_count, with several O(1) operations (string copying and fetching) inside. If we denote the number of one-word queries as O(w), the complexity for this part is O(w).
  - **Parsing Multi-Word Queries**:
    - Similarly, for multi-word queries, the loop runs for multi_word_count, with each iteration also being O(1). If we denote the number of multi-word queries as O(m), the complexity for this part is O(m).
      Combining these analyses, the overall **time complexity** can be expressed as O(n+m+w)
      where n is the fie size, m is the number of multi-word queries and w is the number of one-word queries.
  - The space for the expected_results array depends on the total number of one-word and multi-word queries. Assuming there are T total queries (T = w + m), and each query occupies a certain amount of space in the structure, which includes arrays for document IDs and frequencies.

In summary, the space complexity can be expressed as O((w+m) *D)
where D is the maximum amounts of documents IDs.

- function compute_precision_recall_for_ID:
    - The nested loops for calculating true positives result in *O(m * n)* complexity, where:
        - m = returned_doc_count
        - n = expected_doc_count

      Thus, the overall time complexity is: O(m * n)
      and the space complexity is *O(1)*

- function partition_descending_order:
    - *O(n)* for both the **time complexity** and the **space complexity**

- function partition_descending_order:
    - the same as the upper one

- function quickSort_descending_order:
    - average case: O(nlogn) and worst case O(n^2) for **time complexity**
    - average case: O(logn) and worst case O(n) for **space complexity**

- function quickSort_ascending_order:
    - the same as the upper one

- function process_single_word_query:
    - **Stemming the Word**: Assuming stem takes linear time relative to the length of the word, this operation is O(m), where m is the length of the word.
    - **Searching in Inverted Index**: The search operation iterates through the InvertedFileIndex:

- If InvertedFileIndex->size is n (the number of entries), the loop runs in O(n).
  In total, the time comlexity is O(m+n)
  - the space compexity is O(1)

- function Choose_query_and_threshold:

  - Since the loop runs (n) times and calls one of the functions with a complexity of (O(m)), the total time complexity can be expressed as: O(n * m) where n is the number of words and m is the average time complexity of the query functions.

- function format_doc_id:

  - **time complexity** O(m) where (m) is the length of the input doc_id string.

- function export_to_csv:

  - **Initial Row Writing**:
    - The function uses fprintf to write the first row of the CSV, which includes the query, formatted document IDs, expected document count, returned document count, precision, and recall.
    - The number of document IDs is bounded by expected_doc_count and returned_doc_count.
    - If either of these counts is greater than zero, the function calls format_doc_id, which we previously analyzed as O(m) (with m being the length of the corresponding document ID).
  - **Subsequent Rows Writing**:
    - The loop iterates up to max_docs, which is the maximum of expected_doc_count and returned_doc_count. In each iteration of the loop (which runs max_docs times):
    - It writes a comma (O(1)).
    - It conditionally writes an expected document ID if available, which involves calling format_doc_id potentially O(m).

- It writes another comma (O(1)).
- It conditionally writes a returned document ID if available, also involving format_doc_id potentially O(m).
- Each iteration of the loop consists of constant-time operations plus the time for formatting document IDs.

- **Total Time Complexity**:
  - The first fprintf operation takes O(m) for each formatted ID if present.
  - The loop runs max_docs times, where in each iteration the dominant cost is the call to format_doc_id, thus contributing O(max_docs * m) to the complexity.
  - Therefore, the total time complexity can be expressed as: O(m * max(expected_doc_count, returned_doc_count))
  - Here, (m) is the length of the longest document ID being formatted.

BONUS

We initially tried to generate a large number of different words using statistical patterns and package them into text as experimental material. The advantage of this approach is that word frequency can be calculated, ensuring accuracy. However, the downside is also evident: the patterns are too obvious, making it difficult to mimic real text. Therefore, we developed the program below.

```
import random

import string



def generate_ordered_strings(num_strings, max_length):

    frequency = {}



    # 生成指定数量的字符串

    for _ in range(num_strings):

        # 随机选择字符串长度
```

```
            length = random.randint(1, max_length)
            # 生成随机字符串
            random_string = ''.join(random.choices(string.ascii_lowercase + string.digits,
k=length))

            # 更新频率
            if random_string in frequency:
                frequency[random_string] += 1
            else:
                frequency[random_string] = 1

    # 按频率排序
    sorted_strings = sorted(frequency.items(), key=lambda x: x[1], reverse=True)

    return sorted_strings

# 参数设置
num_strings = 1000  # 生成的字符串总数
max_length = 10  # 每个字符串的最大长度

# 生成有序字符串及其频率
ordered_strings = generate_ordered_strings(num_strings, max_length)
```

```
# 打印结果

for string, freq in ordered_strings:

    print(f"{string}: {freq}")
```

This program generates strings of lengths between 3 and 10 entirely at random, ensuring the randomness of word occurrences. However, it does not address the issue of packaging the text, which led to the development of the following program.

```
import os

import random

import string

from collections import Counter, defaultdict



# 配置参数

total_words = 400_000_000

files_count = 500_000

words_per_file = total_words // files_count



# 创建输出目录

output_dir = "output_words"

os.makedirs(output_dir, exist_ok=True)



# 生成唯一单词

unique_words = set()
```

```
while len(unique_words) < total_words:
    word_length = random.randint(3, 10)
    word = ''.join(random.choices(string.ascii_lowercase, k=word_length))
    unique_words.add(word)

unique_words = list(unique_words)
word_frequency = Counter()
word_files = defaultdict(set)  # 记录每个单词出现的文件名

for i in range(files_count):
    words_to_write = unique_words[i * words_per_file:(i + 1) * words_per_file]
    word_frequency.update(words_to_write)

    file_name = f"words_{i + 1}.txt"

    with open(os.path.join(output_dir, file_name), "w") as f:
        f.write("\n".join(words_to_write))

    for word in words_to_write:
        word_files[word].add(file_name)  # 记录该单词出现的文件名

# 保存词频及文件名统计到文件
```

```
with open(os.path.join(output_dir, "word_frequency.txt"), "w") as freq_file:
    for word, freq in word_frequency.items():
        files_list = ', '.join(word_files[word])  # 获取该单词出现的所有文件名
        freq_file.write(f"{word}: {freq} (files: {files_list})\n")

print(f"成功生成 {total_words} 个不同的单词，分装到 {files_count} 个文件中，词频及文件信息已保存。")
```

In this way, we achieved an average distribution of the corresponding number of words throughout the text. The occurrences of the words and their corresponding frequencies are recorded in the text. When comparing query data, the Ctrl+F key on the computer is used to search the statistical files to obtain the relevant results.

Potential issues with our program include:

- As the number of documents and unique words increases, the system may struggle to scale, leading to longer query processing times.
- The large volume of data may require more storage space than anticipated, potentially necessitating additional infrastructure, such as writing during reading and querying from other texts.
- Handling duplicate or similar documents may increase the size and complexity of the index, complicating search results.
- More documents and words do not necessarily guarantee better search results; without careful management, the relevance of query responses may decline.
- In large-scale data, the likelihood of encountering errors (such as incorrectly formatted documents or unexpected formats) increases, making it challenging for existing systems like stemming to manage, which requires a robust error handling mechanism.

Appendix: Source Code (if required)

At least 30% of the lines must be commented. Otherwise the code will NOT be evaluated.

References

[1] 范加索尔拉，“djb2: 一个产生简单随机分布的哈希函数”,https://www.cnblogs.com/vancasola/p/9951686.html

Declaration

We hereby declare that all the work done in this project titled "ADS_Project_Mini_Search_Engine" is of our independent effort as a group.