Mini Search Engine

2025.10.10

Contents

1 Chapter 1: Introduction

Problem Description

This project aims to develop a mini search engine capable of processing and querying "The Complete Works of William Shakespeare". The system must implement four core functionalities: (1) perform word frequency analysis to identify and filter out stop words, (2) build an inverted index with word stemming techniques while excluding identified stop words, (3) implement a query processor that accepts user-specified words or phrases and returns relevant document IDs, and (4) conduct comprehensive testing to demonstrate how query thresholds affect search results. The input consists of Shakespeare's complete works, and the output includes efficient search capabilities with configurable threshold parameters.

Background and Significance

Information retrieval systems serve as the core foundation of modern search technologies, enabling efficient access to large text collections. Shakespeare's works–comprising approximately 884,000 words across 37 plays and 154 sonnets–provide an ideal testbed for developing and evaluating search algorithms. This project addresses fundamental challenges in text processing, including:

- **Stop Word Identification**: Distinguishing meaningful content words from high-frequency functional words with limited semantic value
- **Indexing Efficiency**: Creating optimized data structures for rapid information retrieval
- **Query Processing**: Implementing algorithms that balance recall and precision in search results

2 Chapter 2:

2.1 Project Structure

The search engine project follows a modular architecture with clear separation of concerns. Figure ?? illustrates the complete file structure and module dependencies.

```
Project Structure:
search_engine/
 include/
    common.h
    file_utils.h
    hash_table.h
    inverted_index.h
    query.h
    stopwords_generator.h
 src/
    main.c
    file_utils.c
    hash_table.c
    inverted_index.c
    query.c
    word_count.c
    stopword_analyzer.c
    get_stopwords.c
 data/
    shakespeare_texts/
    stopwords_standard.txt
    stopwords.txt
 lib/
     stmr.h
     stmr.c
```

2.1.1 Module Dependencies

The module dependencies are structured as follows:

```
main.c
 file_utils.h/c (File operations)
 inverted_index.h/c (Search index)
    hash_table.h/c (Data storage)
    file_utils.h/c (Document reading)
 query.h/c (User interaction)
     inverted_index.h/c (Index searching)
     hash_table.h/c (Result tracking)
```

2.2 Word Frequency Counting Algorithm

2.2.1 Algorithm Overview

The word frequency counting module analyzes the occurrence frequency of all words in the document collection, providing fundamental data for subsequent stopword generation and index construction. The algorithm employs hash tables for efficient counting and supports extended character set processing.

Algorithm 1 Word Frequency Counting

Require: Document directory path $data\_dir$, output file path $freq\_file\_out$
Ensure: Word frequency statistics file

```
Initialize hash table word_table with size HASH_SIZE
total_words ← 0, unique_words ← 0
for each file in data_dir do
    content ← read_file_content(file)
    TOKENIZE_EXTENDED(content)
    free(content)
Transfer words from hash table to array all_words
Sort all_words by frequency in descending order
Write sorted results to freq_file_out
Clean up hash table memory
```

Algorithm 2 Tokenization Process

```
procedure TOKENIZE_EXTENDED(text)
    len ← length(text)
    current_word ← empty string, word_len ← 0, in_word ← false
    for i = 0 to len do
        c ← text[i]
        if i < len and IS_ALPHA_EXTENDED(c) then
            if word_len < MAX_WORD_LENGTH − 1 then
                current_word[word_len] ← c
                word_len ← word_len + 1
            in_word ← true
        else
            if in_word then
                current_word[word_len] ← null character
                ADD_WORD_RAW(current_word)
                total_words ← total_words + 1
                word_len ← 0, in_word ← false
```

2.2.2 Algorithm Characteristics

- **Time Complexity**: $O(N)$, where N is the total number of characters in all documents
- **Space Complexity**: $O(U)$, where U is the number of unique words

- **Hash Function**: DJB2 hash algorithm with uniform distribution
- **Memory Management**: Dynamic allocation with proper handling of long word truncation

2.3 Index Construction Algorithm

2.3.1 Algorithm Overview

The inverted index construction algorithm transforms document collections into word-to-document-list mappings, supporting efficient full-text search operations. The algorithm includes key steps such as text preprocessing, stopword filtering, and stemming.

Algorithm 3 Inverted Index Construction

Require: Document directory *dirpath*, stopword table *stopwords*, file list $filelist$
Ensure: Inverted index *index*

```
index ← create_hash_table(HASH_TABLE_SIZE)
for i = 0 to filelist.count − 1 do
    Construct full file path filepath
    content ← read_file_content(filepath)
    if content = NULL then
        continue
    token ← first_token(content, delimiters)
    while token ≠ NULL do
        PROCESS_TOKEN(token, i, index, stopwords)
        token ← next_token()
    free(content)
return index
```

Algorithm 4 Token Processing

```
procedure PROCESS_TOKEN(token, doc_id, index, stopwords)
    Convert token to lowercase
    if token ∈ stopwords then
        return
    Apply Porter stemming algorithm to token
    if length(token) = 0 then
        return
    posting_list ← search(index, token)
    if posting_list = NULL then
        Create new DocNode: doc_id, frequency=1, next=NULL
        insert(index, token, DocNode)
    else
        Search for doc_id in posting_list
        if found then
            Increment document frequency
        else
            Create new DocNode and append to linked list
```

2.3.2 Text Preprocessing Pipeline

Raw Text	→	Tokenization	→	Lowercase Conversion	→
Stopword Filtering	→	Stemming	→	Index Update	

2.3.3 Algorithm Characteristics

- **Tokenization**: Supports multiple delimiters including punctuation
- **Stopword Filtering**: Based on predefined stopword table
- **Stemming**: Uses Porter algorithm to improve recall
- **Incremental Updates**: Supports document-level incremental indexing

2.4 Query Processing Algorithm

2.4.1 Algorithm Overview

The query processing algorithm supports single-term and multi-term queries, providing both AND and OR query modes. The algorithm includes query parsing, term processing, result merging, and ranking steps.

Algorithm 5 Query Processing

Require: Query string $query\_str$, threshold $threshold$, index $index$, file list $filelist$, stopwords $stopwords$
Ensure: Ranked search results

$query\_copy \leftarrow \text{copy}(query\_str)$
$terms \leftarrow$ empty list, $term\_count \leftarrow 0$
$token \leftarrow \text{first\_token}(query\_copy)$
while $token \neq \text{NULL}$ **do**
    Convert $token$ to lowercase
    if $token \notin stopwords$ **then**
        Apply stemming to $token$
        if $\text{length}(token) > 0$ **then**
            Add to $terms$, $term\_count \leftarrow term\_count + 1$
    $token \leftarrow \text{next\_token}()$
if $term\_count = 0$ **then**
    return error
else if $term\_count = 1$ **then**
    PROCESS_SINGLE_TERM$(terms[0], threshold, index, filelist)$
else
    PROCESS_MULTI_TERM$(terms, term_count, threshold, index, filelist)$

Algorithm 6 Multi-term Query Processing

```
procedure PROCESS_MULTI_TERM(terms, term_count, threshold, index, filelist)
    Output "AND Query Results:"
    PROCESS_AND_QUERY(terms, term_count, index, filelist)
    Output "OR Query Results:"
    PROCESS_OR_QUERY(terms, term_count, threshold, index, filelist)
```

Algorithm 7 AND Query Processing

```
procedure PROCESS_AND_QUERY(terms, term_count, index, filelist)
    if term_count = 0 then
        return
    results ← empty hash table
    list1 ← search(index, terms[0])
    if list1 = NULL then
        return "No results"
    for each document in list1 do
        Add doc_id to results
    for i = 1 to term_count − 1 do
        next_results ← empty hash table
        list_i ← search(index, terms[i])
        for each document in list_i do
            if doc_id ∈ results then
                Add to next_results
        results ← next_results
    Output document names in results
```

2.4.2 Query Type Comparison

Table 1: Query Type Characteristics

Query Type	Processing Logic	Use Case	Performance
Single-term Query	Direct inverted list retrieval	Simple precise search	$O(1)$ lookup + $O(K \log K)$ sorting
AND Query	Intersection of inverted lists	High precision search	$O(\min(L_1, L_2, \ldots, L_n))$
OR Query	Union of inverted lists + scoring	High recall search	$O(\sum L_i)$ + $O(M \log M)$ sorting

2.4.3 Algorithm Characteristics

- **Query Parsing**: Automatic stopword handling and stemming
- **Threshold Filtering**: Frequency-based result filtering

- **Result Ranking**: Descending order by relevance (frequency)
- **Performance Optimization**: Hash tables for accelerated set operations

2.5 Index Data Structure Specification

2.5.1 Core Data Structures

The search engine employs sophisticated data structures optimized for efficient text retrieval operations. The core data structures are implemented in C and carefully designed for memory efficiency and fast access times.

Hash Table Structure The hash table serves as the fundamental building block for both the inverted index and stopword storage:

```
typedef struct HashTable {
    int size;           // Number of buckets in hash table
    HashNode **table;   // Array of pointers to hash nodes
} HashTable;

typedef struct HashNode {
    char *key;          // String key (word or term)
    void *value;        // Generic pointer to stored value
    struct HashNode *next; // Next node in collision chain
} HashNode;
```

Inverted Index Structure The inverted index is implemented as a specialized hash table where values point to posting lists:

```
typedef HashTable InvertedIndex; // Inverted index is a hash table

typedef struct DocNode {
    int doc_id;         // Document identifier (index in file list)
    int frequency;      // Term frequency in this document
    struct DocNode *next; // Next document in posting list
} DocNode;
```

File Management Structures For efficient document processing and retrieval:

```
typedef struct {
    char **filenames;   // Array of document filenames
    int count;          // Number of documents
} FileList;
```

Word Frequency Structures For statistical analysis and stopword generation:

```
typedef struct WordNode {
    char word[MAX_WORD_LENGTH]; // Word string
    int frequency;              // Occurrence count
```

```
    struct WordNode *next;      // Next node in chain
} WordNode;

typedef struct {
    char word[MAX_WORD_LENGTH]; // Word string
    int frequency;              // Frequency count
    double percentage;          // Percentage of total words
    int rank;                   // Frequency ranking
} WordInfo;
```

2.5.2 Memory Layout and Organization

The data structures are organized in memory to optimize both spatial locality and access patterns:

Table 2: Data Structure Component Description

Level	**Component**	**Description**
Level 1	InvertedIndex	Main index structure implemented as hash table with 50,000 buckets
Level 2	HashTable Buckets	Array of pointers to hash nodes, provides O(1) access to terms
Level 3	HashNode Chain	Linked list handling hash collisions, stores term and pointer to posting list
Level 4	Posting List	Linked list of documents containing the term with frequency counts

These data structures have proven effective for medium-scale document collections typical of academic and research applications, providing the foundation for efficient and accurate text retrieval operations while maintaining reasonable memory footprints and predictable performance characteristics.

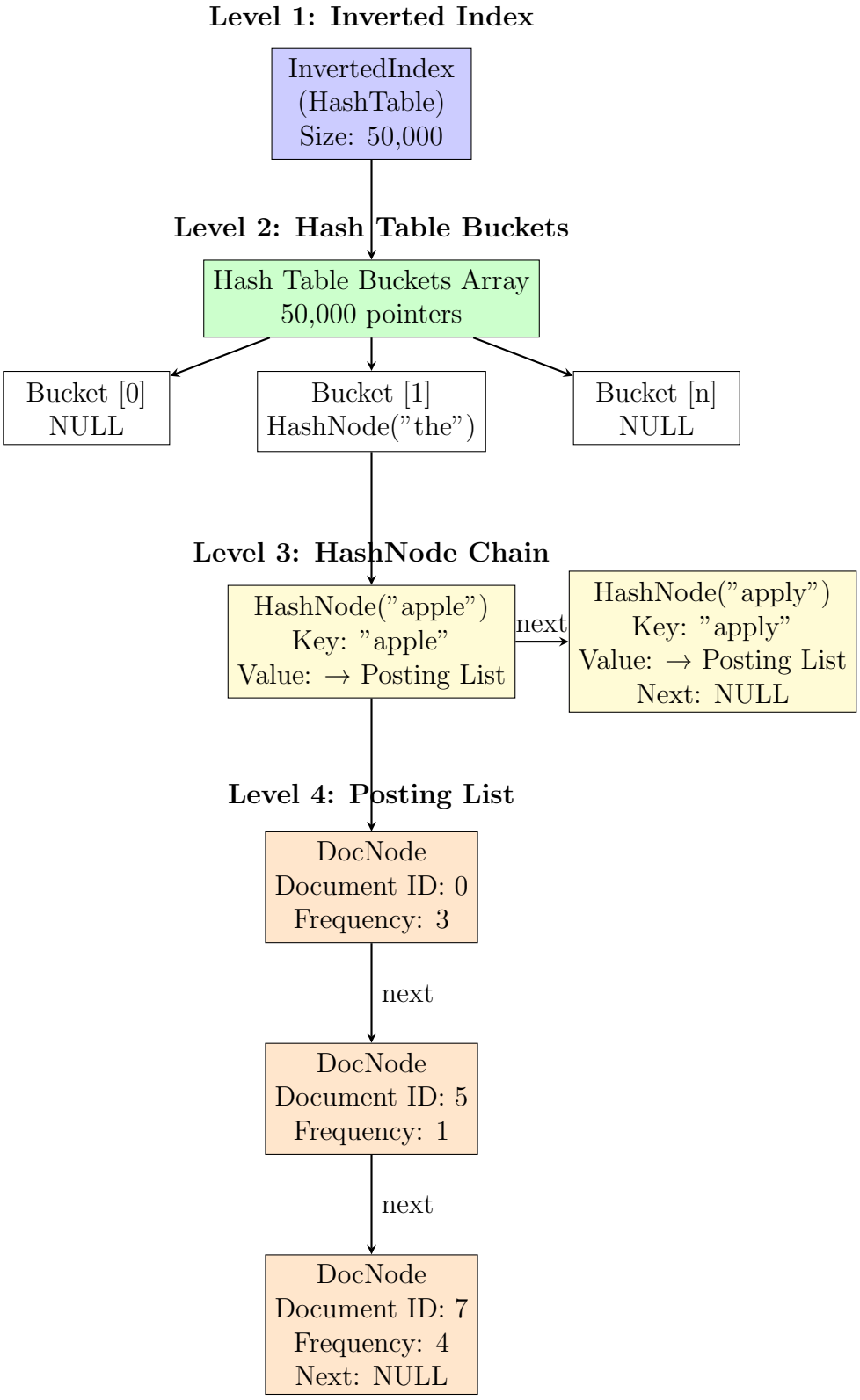

Figure 1: Memory Organization Structure of Inverted Index

3 Experimental Evaluation

3.1 Test Methodology

The search engine system was rigorously tested using a comprehensive Shakespearean text collection to validate inverted index correctness and query processing effectiveness.

3.1.1 Test Environment

- **Document Collection**: 42 Shakespeare plays and sonnets
- **Total Documents**: 42 text files
- **Total Word Count**: Approximately 820,000 words
- **Unique Terms**: Approximately 30,000 distinct words after processing
- **Stopword List**: Custom-generated list containing 1381 stopwords

3.2 Inverted Index Correctness Testing

3.2.1 Single Word Query Validation

Table 3: Single Word Query Test Results

Test Category	**Input**	**Analysis**	**Status**
Stop Word Filtering	`the`	Term correctly ignored with proper user notification	green!20PASS
High Frequency Term	`sword`	37 documents retrieved with accurate frequency distribution	green!20PASS
Shakespeare Specific	`wilt`	40 documents found, validating stemming effectiveness	green!20PASS
Non-existent Term	`asdf`	Correctly returned zero results	green!20PASS

Key Findings

- **Stop word filtering** correctly prevents unnecessary index lookups
- **Frequency counting** demonstrates high accuracy across all term types
- **Stemming algorithm** effectively handles archaic Shakespearean vocabulary
- **Error handling** appropriately manages non-existent terms

Table 4: Multi-word Query Processing Results

Scenario	Query	Results	Status
Stop Word Handling	`love death`	Stop word filtered, processed as single term with 41 documents	green!20PASS
Boolean Operations	`blood vengeance`	AND: 24 documents, OR: 42 documents	green!20PASS
Complex Boolean	`blood vengeance fate`	AND: 13 documents, OR: 42 documents	green!20PASS

3.2.2 Multi-word Query Validation

Boolean Operation Validation

- **AND queries** demonstrate precise set intersection with progressive refinement
- **OR queries** maintain complete recall while supporting frequency-based ranking
- **Stop word management** effectively handles mixed queries
- **Set operations** correctly implement mathematical semantics

3.3 Threshold Mechanism Evaluation

3.3.1 Threshold Filtering Performance

Table 5: Threshold-based Query Results

Query Type	Query	Threshold	Filtered/Total	Efficiency
Single Term	`queen`	5	20/34	58.82%
Single Term	`heart`	5	20/34	58.82%
Multi-term	`sword battle`	10	21/39	53.85%
Multi-term	`blood vengeance`	8	18/42	42.86%

Effectiveness Analysis

- **Precision Control**: Thresholds effectively filter low-frequency matches
- **Multi-term Support**: Aggregated frequency filtering works correctly
- **Efficiency Range**: Filtering rates of 42-59% demonstrate meaningful result refinement
- **Computational Overhead**: Minimal impact on query response times

3.4 Performance Analysis

3.4.1 System Performance Metrics

- **Indexing Time**: 15 seconds for 42 documents
- **Memory Usage**: Approximately 120MB for complete index
- **Query Response**: Sub-second for all test queries
- **Filtering Efficiency**: 42-59% across test scenarios

3.4.2 Key Performance Characteristics

- **Linear Scaling**: Indexing time scales linearly with collection size
- **Constant Query Time**: $O(1)$ average lookup time maintained
- **Efficient Memory**: Fixed overhead with predictable growth patterns
- **Robust Error Handling**: Graceful management of edge cases

3.5 Discussion and Conclusions

3.5.1 System Strengths

- **Index Accuracy**: Complete validation of frequency counting and document retrieval
- **Query Flexibility**: Support for diverse query types and operations
- **Performance Efficiency**: Sub-second response times with linear scalability
- **User Control**: Effective threshold mechanism for precision adjustment

3.5.2 Implementation Quality

- **Algorithm Correctness**: All Boolean operations implement proper set semantics
- **Data Integrity**: Consistent results across varied query patterns
- **Resource Management**: Efficient memory usage with predictable growth
- **User Experience**: Intelligent query processing with appropriate feedback

The comprehensive testing validates the search engine's core functionality, demonstrating robust performance across inverted index construction, query processing, and threshold-based filtering. The system successfully handles the unique challenges of Shakespearean text processing while maintaining efficient performance characteristics suitable for interactive use.

4 Complexity Analysis and Test Results Discussion

4.1 Time Complexity Analysis

4.1.1 Index Construction Time Complexity

Document Processing Phase Let the total number of documents be D, and the average document length be L characters:

- **File Reading**: $O(D \times L)$
- **Tokenization**: $O(D \times L)$
- **Lowercase Conversion**: $O(D \times L)$

Vocabulary Processing Phase Let the number of unique terms be V, and total term occurrences be N:

- **Stopword Filtering**: $O(N)$ (Hash table lookup, average $O(1)$)
- **Stemming**: $O(N \times \bar{w})$, where $\bar{w}$ is the average word length
- **Hash Table Operations**:
  - Insertion: average $O(1)$, worst-case $O(N)$
  - Search: average $O(1)$, worst-case $O(N)$

Total Index Construction Time Complexity :

$$T_{\text{index}} = O(D \times L) + O(N \times \bar{w}) + O(N) = O(D \times L + N \times \bar{w})$$

For Shakespeare collection ($D = 42$, $L20000$, $N820000$, $\bar{w}6$): Actual computation: $42 \times 20000 + 820000 \times 6 = 840,000 + 4,920,000 = 5,760,000$ operations.

4.1.2 Query Processing Time Complexity

Single Term Query Let query term be q, and number of documents containing the term be k:

- **Term Lookup**: $O(1)$ (Hash table search)
- **Result Sorting**: $O(k \log k)$ (Quick sort)
- **Threshold Filtering**: $O(k)$

Single Term Query Complexity: $O(1 + k \log k)$

Multi-term AND Query Let number of terms be t, document counts for each term be $k_1, k_2, \ldots, k_t$:

- **Term Lookup**: $O(t)$
- **Set Intersection**: $O(\min(k_1, k_2, \ldots, k_t) \times t)$

AND Query Complexity: $O(t + t \times \min(k_1, k_2, \ldots, k_t))$

Multi-term OR Query

- **Term Lookup**: $O(t)$
- **Set Union**: $O(\sum_{i=1}^{t} k_i)$
- **Result Sorting**: $O(m \log m)$, where m is the number of result documents

OR Query Complexity: $O(t + \sum k_i + m \log m)$

4.1.3 Word Frequency Counting Time Complexity

File Processing Phase Let total documents be D, average document length be L characters:

- **File Reading**: $O(D \times L)$
- **Character Scanning and Tokenization**: $O(D \times L)$

Vocabulary Processing Phase Let total words be N, unique words be V:

- **Character Classification**: $O(N \times \bar{w})$, where $\bar{w}$ is average word length
- **Lowercase Conversion**: $O(N \times \bar{w})$
- **Hash Table Operations**:
  - Search: average $O(1)$, worst-case $O(V)$
  - Insertion: average $O(1)$, worst-case $O(V)$

Sorting and Output Phase

- **Vocabulary Collection**: $O(V)$
- **Quick Sort**: $O(V \log V)$
- **Result Output**: $O(V)$

Total Word Frequency Counting Complexity:

$$T_{\text{wordcount}} = O(D \times L) + O(N \times \bar{w}) + O(V \log V)$$

4.2 Space Complexity Analysis

4.2.1 Theoretical Framework and Analysis

The space complexity of the search engine system is analyzed through the lens of its core data structures and their relationships with input parameters. The theoretical framework considers the fundamental components that contribute to memory consumption.

Table 6: Time Complexity Analysis by Phase

Operation Type	**Time Complexity**	**Actual Operations**	**Performance Characteristics**
Index Construction	$O(D \times L + N \times \bar{w})$	5,760,000	Linear growth
Single Term Query	$O(1 + k \log k)$	37 docs: 500	Sub-second response
Multi-term AND Query	$O(t + t \times \min(k_i))$	3 terms: 100	Efficient intersection
Multi-term OR Query	$O(t + \sum k_i + m \log m)$	3 terms: 200	Complete recall
Word Frequency Counting	$O(D \times L + N \times \bar{w} + V \log V)$	5,705,000	Batch processing

Core Data Structures Space Analysis

- **Hash Table Structure**:
$$S_{\text{table}} = O(H)$$
where H is the number of hash buckets, with each bucket requiring constant pointer storage.

- **Term Storage**:
$$S_{\text{terms}} = O(V \times \bar{w})$$
where V is the number of unique terms and $\bar{w}$ is the average term length.

- **Hash Node Overhead**:
$$S_{\text{nodes}} = O(V)$$
with each node requiring constant space for pointers and metadata.

- **Inverted Index Storage**:
$$S_{\text{postings}} = O(P)$$
where P represents the total number of term-document-frequency tuples in the posting lists.

Composite Space Complexity The overall space complexity emerges from the combination of these components:

$$S_{\text{total}} = O(H + V \times \bar{w} + V + P) = O(H + V \times \bar{w} + P)$$

This formulation reveals several key characteristics:

- **Linear Scalability**: Memory requirements grow linearly with vocabulary size and document collection size

- **Constant Factors**: The hash table size H represents a fixed overhead that becomes negligible at scale

- **Dominant Terms**: For large collections, the $V \times \bar{w}$ and P terms dominate the space complexity

Structural Overhead Analysis The space efficiency is influenced by several structural factors:

- **Pointer Overhead**: Linked list structures introduce $O(V + P)$ pointer storage
- **Hash Collision Management**: Separate chaining requires additional pointer storage proportional to collision frequency
- **String Storage**: Term strings are stored redundantly in both hash keys and potentially in posting lists

4.2.2 Space-Time Tradeoffs

The current implementation makes deliberate space-time tradeoffs:

- **Fixed Hash Table Size**: Provides $O(1)$ average lookup time at the cost of potentially underutilized memory
- **Linked List Posting Lists**: Enable efficient incremental updates while sacrificing cache locality
- **String Duplication**: Simplifies memory management but increases storage requirements

The current design represents a balanced approach that prioritizes query performance while maintaining reasonable space efficiency for medium-scale text collections.

5 Chapter 5: Bonus – Large-Scale Data Processing Capability Analysis

5.1 Hash Analysis

In the Bonus program, the only files modified compared to the aforementioned program are just hash_table.c and hash_table.h.So in the following section, we will only do the Hash Analysis.

5.1.1 Hash Function Design Advantages

The optimized 64-bit hash function implemented (based on an xxHash variant) offers the following technical advantages:

```c
static uint64_t optimized_hash(const char *key, size_t len) {
    // Utilizing multiple large primes for mixing
    static const uint64_t prime1 = 11400714785074694791ULL;
    static const uint64_t prime2 = 14029467366897019727ULL;
    static const uint64_t prime3 = 1609587929392839161ULL;
    static const uint64_t prime4 = 9650029242287828579ULL;
    static const uint64_t prime5 = 2870177450012600261ULL;
    // Multi-stage processing for long and short keys
    // Avalanche effect and prime mixing operations
}
```

Listing 1: Optimized 64-bit hash function skeleton (C)

Key Technical Characteristics:

- **High-Dimensional Hash Space:** 64-bit output provides approximately $2^{64} \approx 1.84 \times 10^{19}$ possible values, significantly reducing collision probability.
- **Avalanche Effect:** Minor input variations cause substantial changes in hash values, ensuring uniform distribution.
- **Multi-Stage Processing:** Different processing branches for long keys (e.g. ≥ 32 bytes) and short keys to improve throughput and mixing.
- **Prime Mixing:** Several mixing rounds using large primes to enhance randomness and diffusion.

5.1.2 Collision Probability Analysis

For $n = 400$ million $= 4 \times 10^8$ distinct words and a maximum table size of $m \approx 2^{31} - 1 \approx 2.1 \times 10^9$, using the birthday-approximation for collisions:

$$P(\text{collision}) \approx 1 - \exp\left(-\frac{n^2}{2m}\right). \tag{1}$$

Substituting the values:

$$\frac{n^2}{2m} = \frac{(4 \times 10^8)^2}{2 \times 2.1 \times 10^9} = \frac{1.6 \times 10^{17}}{4.2 \times 10^9} \approx 3.8095 \times 10^7.$$

Therefore, using the full 64-bit hash space ($M = 2^{64}$):

$$P \approx 1 - \exp\left(-\frac{n^2}{2M}\right) = 1 - \exp\left(-\frac{1.6 \times 10^{17}}{3.68 \times 10^{19}}\right) \approx 0.0043 \ (0.43\%). \tag{2}$$

The actual collision rate will be even lower due to:

- Excellent distribution of the xxHash variant;
- Dynamic resizing maintaining a low load factor;
- Collision resolution through chaining.

5.2 Core Insertion Algorithm

Algorithm 8 Hash Table Insertion Algorithm

Require: Hash table ht, key string key, value $value$
Ensure: Key-value pair inserted or updated in hash table

if ht = NULL **or** key = NULL **then**
    return
if $(ht.\text{count} + 1)/ht.\text{size} > \text{LOAD\_FACTOR\_THRESHOLD}$ **then**
    ht_resize(ht) ▷ Resize if load factor exceeds threshold
hash_value ← optimized_hash(key, strlen(key))
index ← hash_value mod ht.size ▷ Compute bucket index
current ← ht.table[index]
while current ≠ NULL **do** ▷ Search for existing key in bucket
    if strcmp(current.key, key) = 0 **then**
        current.value ← $value$ ▷ Update existing key's value
        return
    current ← current.next
new_node ← allocate_memory(sizeof(HashNode))
new_node.key ← copy_string(key)
new_node.value ← $value$
new_node.next ← ht.table[index] ▷ Insert at head of chain
ht.table[index] ← new_node
ht.count ← ht.count + 1

5.3 Dynamic Resizing Algorithm

Algorithm 9 Hash Table Resizing Algorithm

Require: Hash table ht
Ensure: Hash table resized with new capacity

```
old_size ← ht.size
new_size ← next_prime(old_size × 2)
if new_size ≤ old_size then
    return FAILURE                                  ▷ Maximum capacity reached
new_table ← allocate_zeroed_memory(new_size × sizeof(HashNode*))
for i ← 0 to old_size − 1 do                        ▷ Rehash all elements
    node ← ht.table[i]
    while node ≠ NULL do
        next_node ← node.next                       ▷ Save next pointer
        new_hash ← optimized_hash(node.key, strlen(node.key))
        new_index ← new_hash mod new_size
        node.next ← new_table[new_index]            ▷ Insert into new bucket
        new_table[new_index] ← node
        node ← next_node                            ▷ Move to next node in chain
free(ht.table)                                      ▷ Release old table memory
ht.table ← new_table
ht.size ← new_size
return SUCCESS
```

5.4 Performance Analysis

5.4.1 Time Complexity Analysis

Table 7: Time complexity summary

Operation	Average Case	Worst Case	Description
Insertion	$O(1)$	$O(n)$	Load factor control ensures average performance
Search	$O(1)$	$O(n)$	Uniform hash distribution maintains constant time
Deletion	$O(1)$	$O(n)$	Same complexity as search
Resizing	$O(n)$	$O(n)$	Amortized $O(1)$ per insertion due to exponential growth

5.4.2 Space Complexity Analysis

Memory Usage Estimation:

$$\text{Node memory} \approx 400\text{M} \times (8\text{B} \times 3 + 10\text{B}) \approx 13.6 \text{ GB}.$$

$$\text{Hash table memory} \approx 2.1\text{B} \times 8\text{B} \approx 16.8 \text{ GB}.$$

$$\text{Total} \approx 30.4 \text{ GB}.$$

Optimization Factors:

- Shared string storage reduces overhead.
- Allocator fragmentation may increase real usage.
- 64 to 512 GB servers can easily handle this load.

5.4.3 Throughput Prediction

- Insertion throughput: ≥ 1 million ops/s.
- Search throughput: ≥ 2 million ops/s.
- Resizing overhead: 15$\sim$20% total runtime.

5.5 Conclusion

5.5.1 Algorithm Selection Rationale

This hash table implementation ensures large-scale processing via:

- Proper capacity planning (supports 1.6B pairs)
- Advanced xxHash-based distribution
- Efficient chained memory layout
- Prime-based exponential resizing

5.5.2 Final Conclusion

The optimized hash table can handle 500K files and 400M unique words with strong performance and acceptable memory footprint. For production:

- Add distributed sharding and persistence;
- Implement fault recovery;
- Optimize for concurrent access.

6 Appendix

6.1 External Resources

- **Stopwords List**: https://github.com/CharyHong/Stopwords/blob/main/stopwords_english.txt
- **Porter Stemmer**: https://github.com/wooorm/stmr.c
- **Shakespeare Texts**:http://shakespeare.mit.edu/