BlinkSearch

by

Tatiya Seehatrakul

Shreeyukta Pradhanang

A Thesis Submitted in

AT70.24 Algorithms Design and Analysis

Instructure: Dr. Chantri Polprasert

Teacher Assistant: Rakshya Lama Moktan

Asian Institute of Technology

School of Engineering and Technology

Thailand

May 2025

CONTENTS

LIST OF FIGURES

ABSTRACT

Efficient search algorithms are essential for achieving fast and accurate retrieval of relevant information from large-scale text datasets. This project simulates and benchmarks four distinct search techniques: Linear Search, Inverted Index Search, Trie-based Search, and B-Tree Search, using a real-world dataset of Amazon book reviews in JSON format.

The system enables users to perform keyword queries through an interactive web interface. Each query is processed by all four algorithms, and corresponding performance metrics such as execution time, peak memory usage, and the number of matched results are recorded. The system supports three types of queries: exact match, starts with, and contains. The results are presented through comparative performance charts that display execution time, memory consumption, and relative speedup.

The study offers an empirical comparison of algorithmic trade-offs in terms of both time complexity and space complexity for text retrieval tasks. The findings illustrate how each algorithm scales with increasing dataset size and adapts to different query types. These insights are valuable for selecting appropriate search strategies in real-world applications such as recommendation systems, search engines, and educational platforms.

CHAPTER 1
INTRODUCTION

1.1 Background of the Study

With the exponential growth of digital news sources, retrieving relevant articles efficiently has become an essential task in information retrieval systems. Traditional search techniques, such as linear search, exhibit significant performance limitations as dataset sizes increase. This project proposes the development of an optimized search engine that utilizes advanced data structures to enhance query processing speed and efficiency.

1.2 Statement of the Problem

Retrieving relevant textual data from large-scale datasets remains a computationally intensive task. While Linear Search offers a simple and intuitive baseline, its performance degrades significantly as dataset size increases due to its sequential nature. To address this limitation, more advanced data structures such as Inverted Indexes (commonly used in search engines), Tries (frequently applied in autocomplete systems), and B-Trees (utilized in database indexing) have been developed to optimize search operations. However, each of these methods introduces distinct trade-offs in terms of execution time, memory usage, and scalability.

This study investigates these trade-offs by benchmarking the four algorithms under realistic query scenarios, including *exact match*, *starts with*, and *contains* queries. Key challenges include the inefficiency of sequential search on large datasets, the necessity of indexing mechanisms for faster lookups, the performance variability of advanced data structures across different query types, and the computational and memory overhead associated with building and maintaining such indexing systems.

1.3 Objectives

The primary objective of this study is to implement and evaluate the performance of four distinct search algorithms: Linear Search, Inverted Index, Trie Search, and B-Tree Search. These algorithms are applied to a JSON-formatted dataset of Amazon book reviews to simulate realistic search scenarios.

The evaluation focuses on three key performance indicators: execution time across varying dataset sizes, memory usage during indexing and querying, and responsiveness to different query types, including *exact match*, *starts with*, and *contains* queries. These query types are selected to represent common patterns in real-world search behaviors.

In addition, the study aims to analyze the time complexity and space complexity of each algorithm in practical settings. This analysis provides insight into how well each method scales with increased data volume and supports different search requirements.

The results are presented through visual performance comparisons using interactive graphs and charts, which enable a comprehensive analysis of the trade-offs between speed, memory efficiency, and accuracy across all evaluated techniques.

1.4 Target User

This system is designed for users who require efficient and scalable search capabilities over large datasets. The primary target audience includes researchers and data analysts who process extensive textual data and demand optimized search techniques. It also caters to software developers seeking efficient search implementations that can be integrated into various applications, as well as information retrieval specialists who require comparative insights into different search algorithms to enhance system performance. Overall, this project aims to deliver a comprehensive evaluation of search methodologies to support the development of high-performance data retrieval systems.

CHAPTER 2
LITERATURE REVIEW

2.1 Inverted Index and Trie-based Search for Efficient Large-Scale Text Retrieval

Wang et al. [1] proposed two improved indexing methods, FASTER-INV and AC-INV, as enhancements to the traditional inverted index. FASTER-INV removes unnecessary steps to improve speed and memory efficiency for keyword-based search, while AC-INV combines a trie structure with the Aho-Corasick algorithm for efficient prefix-based matching. Both were evaluated on the CAIL2018 dataset [2], a Chinese legal corpus in JSON format with 200,000 to 1.5 million documents, each averaging 660–693 words. Scalability was tested by increasing document count from 300k to 1.5M. The study measured indexing time, memory usage, and speedup, and visualized results using graphs and tables. FASTER-INV achieved up to 1.14× speedup with 10% less memory, while AC-INV reached 1.43× speedup and 35% lower memory usage, demonstrating the efficiency of these techniques for scalable text search.

2.2 Linear, Inverted Index, and Trie-based Search with Sketch Structures

Kanda and Tabei [3] introduced the b-bit Sketch Trie (bST), a compact, scalable data structure for fast similarity search on large datasets of integer-based sketches. They evaluated linear search, inverted index search (Single Index Hashing or SIH), and trie-based search (SI-bST and MI-bST). Experiments used four datasets: Amazon Book Reviews [4] (JSON format), Compound–Protein sketches [5] (likely binary or matrix format), and SIFT1B descriptors from BIGANN [6] (.fvecs format). These were converted into sketches using b-bit MinHash or 0-bit Consistent Weighted Sampling (CWS) for Hamming distance comparisons. From each dataset, 10,000 sketches were randomly selected as queries. Results were presented with bar charts (query time), tables (memory usage), and line graphs (cost vs. threshold τ). SI-bST gave the best results on Review, CP, and SIFT1B, completing billion-scale queries with only 9 GiB of memory. MI-bST was better on the GIST dataset due to longer sketches and higher thresholds. While linear search performed at high τ, it lacked scalability. SIH was less effective due to excessive candidate generation. Overall, bST methods offered the best trade-off between speed and memory across datasets.

2.3 Inverted Index and B-Tree Search on Malay Text Documents

Rosnan et al. [7] compared three indexing techniques: Inverted Files, B-Tree, and B+ Tree, using a self-constructed dataset of 500 Malay text documents, referred to as the *Malay Honey Corpus* [8]. The documents were preprocessed through tokenization, stopword removal, and stemming. From just two documents, 201 unique terms were extracted, suggesting a much larger vocabulary across the full dataset. Ten multi-word keyword queries, gathered via student surveys and related to honey (e.g., "khasiat madu asli"), were used to evaluate performance. The study measured indexing time, retrieval time, and retrieval effectiveness (precision, recall, and F1 score), with results presented using tables and line graphs. While Inverted Files offered faster indexing, B+ Tree achieved better retrieval speed and scalability. No specific limit on search depth or result count was stated.x

2.4 Hybrid Trie and Inverted Index for Set-Valued Data Search

Terrovitis et al. [9] proposed the HTI (Hybrid Trie-Inverted file) index, a structure that combines Trie-based and Inverted Index search methods for efficient querying over set-valued attributes. This technique is relevant to our study as it explores two of the four search strategies we aim to compare. The HTI index supports subset, superset, and equality queries by placing the top-k most frequent items in a Trie (main memory) and storing the rest in an inverted file (secondary storage), reducing disk I/O significantly. The authors tested their method using real datasets from the UCI KDD archive: *msweb* [10] (stored in CSV format with 320,000 user sessions and 294 items) and *msnbc* [11] (1 million sessions with 17 items). Additional synthetic data with Zipfian distributions were used for scalability testing. Performance was evaluated based on disk page accesses and memory overhead. The HTI index outperformed traditional inverted files, reducing disk accesses by up to 90% in large queries while maintaining low memory usage under 0.5MB. Results were visualized using line graphs (to show scalability over vocabulary size, database size, and query size), bar charts (for average query cost comparisons), and tables (for memory footprint). This study provides strong support for the efficiency of Trie-based and Inverted Index methods, directly informing our project's comparative analysis of search techniques.

CHAPTER 3
METHODOLOGY

3.1 Algorithm Design

The system is designed to evaluate the efficiency of different search algorithms by processing textual data from a local JSON file containing Amazon book reviews. After loading the dataset into memory, the system performs searches using four different algorithmic approaches: Linear Search, Inverted Index, Trie, and B-Tree. Each method is implemented independently, allowing direct comparison based on performance metrics such as execution time and memory usage.

- **Linear Search** serves as the baseline approach by sequentially traversing each review in the dataset and applying a query-matching function. Although it is straightforward to implement and supports all query types, its linear time complexity renders it inefficient for large-scale datasets.
- **Inverted Index Search** constructs an in-memory index that maps each unique, normalized word to the collection of review texts in which it appears. This index enables rapid retrieval of candidate documents for exact and partial keyword-based queries. While the index is built dynamically at runtime rather than precomputed, the implementation adheres to the fundamental principles of inverted indexing commonly employed in search engines.
- **Trie Search** organizes review texts within a prefix tree data structure, wherein each character in a query corresponds to a level within the tree. This structure supports efficient retrieval for prefix-based queries, particularly those utilizing the `starts_with` query type. For other query types, the algorithm traverses all stored sentences. Although it improves prefix-matching efficiency, it incurs higher memory overhead due to its character-level branching nature.
- **B-Tree Search** implements a balanced multi-way search tree that maintains sorted order among inserted review texts. The B-Tree structure allows efficient insertion and logarithmic-time search operations. Queries are resolved through recursive traversal using a keyword-matching function, supporting exact, prefix, and substring-based searches. This approach offers scalability and performance benefits, particularly for range-based or sorted data retrieval.

3.1.1 Linear Search

Linear Search is a foundational algorithm that sequentially scans each review entry and compares its content with a user-specified query. Depending on the selected query type, namely `exact`, `starts_with`, or `contains`, a corresponding matching function is applied to determine whether the text should be included in the result set. The process terminates once a predefined limit is reached. Throughout execution, both the total processing time and peak memory usage are recorded for performance evaluation.

Despite its simplicity and ease of implementation, Linear Search exhibits a time complexity of $\mathcal{O}(n)$, rendering it inefficient for large-scale datasets due to its linear growth in both time and memory consumption.

```
def linear_search_streaming(file_path, query, limit=None,
    query_type="exact"):
    tracemalloc.start()
    start_time = time.time()
    results = []

    with open(file_path, "r") as f:
        for i, line in enumerate(f):
            if limit is not None and i >= limit:
                break
            text = json.loads(line).get("reviewText", "").
                strip()
            if is_match(text, query, query_type):
                results.append(text)

    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    return {
        "matches": results,
        "time": round((time.time() - start_time) * 1000, 2),
        "memory": round(peak / 1024, 2)
    }
```

3.1.2 Inverted Index Search

The Inverted Index Search algorithm adopts a classical information retrieval strategy by constructing an index that maps each unique, normalized word to a list of review texts in which it appears. During initialization, each review is normalized and tokenized, and the resulting tokens are used to populate the inverted index. This preprocessing step enables faster lookup during search operations.

At query time, the algorithm retrieves candidate documents based on the query type, which may be `exact`, `starts_with`, or `contains`. To ensure consistency with the evaluation logic used in other algorithms, the matched results are further refined using a standardized matching function. Execution time and peak memory consumption are also recorded to support comparative analysis.

```
for line in dataset:
    text = review.get("reviewText", "")
    for word in set(normalize(text).split()):
        inverted_index[word].append(text)

if query_type == "exact":
    matches = inverted_index.get(normalize(query), [])
elif query_type == "starts_with":
    for word in inverted_index:
        if word.startswith(normalize(query)):
            matches.update(inverted_index[word])

# Final filter step
final_matches = [t for t in matches if is_match(t, query,
   query_type)]
```

3.1.3 Trie Search

Trie Search utilizes a prefix tree (Trie) to store and retrieve review texts, offering efficient support for prefix-based queries. Each character of the input text is used to traverse and construct the tree structure, where complete sentences are stored at relevant nodes. For query type `starts_with`, the Trie directly retrieves matching entries by prefix. For other query types, such as `exact` and `contains`, the algorithm filters all stored sentences using a shared matching function. Execution time and peak memory usage are recorded to facilitate performance benchmarking.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.sentences = []

class Trie:
    def __init__(self):
        self.root = TrieNode()
        self.inserted_sentences = []

    def insert(self, sentence):
        self.inserted_sentences.append(sentence)
        node = self.root
        for char in sentence.lower():
            node = node.children.setdefault(char, TrieNode())
            node.sentences.append(sentence)

    def search_prefix(self, prefix):
        node = self.root
        for char in prefix.lower():
            if char not in node.children:
                return []
            node = node.children[char]
        return node.sentences
```

```
def search(query, query_type="exact"):
    if query_type == "starts_with":
        result = self.trie.search_prefix(query)
    else:
        result = [t for t in self.trie.get_all_sentences() if
            is_match(t, query, query_type)]
```

3.1.4 B-Tree Search

B-Tree Search applies a balanced tree structure to organize review texts lexicographically by normalized keywords. Each internal node maintains multiple keys and associated values (sentences), supporting efficient search and insertion operations. The structure enables logarithmic traversal during queries, making it suitable for scalable information retrieval tasks.

During construction, each sentence is tokenized, and unique normalized words are inserted along with the full review text. The search method recursively traverses the tree, comparing each key using a query-matching function based on the specified query type (`exact`, `starts_with`, or `contains`). Execution time and peak memory usage are also recorded for benchmarking.

```
class BTreeNode:
    def __init__(self, t, leaf=False):
        self.keys, self.values, self.children = [], [], []
        self.t, self.leaf = t, leaf

    def insert_non_full(self, key, value):
        ...
        if self.leaf:
            # Insert key-value in correct position or append
               value if duplicate
        else:
            # Recursively insert in the appropriate child

    def split_child(self, i):
        ...
        # Split full child node and adjust parent node

    def search(self, query, query_type="exact"):
        results = []
        for i, key in enumerate(self.keys):
            if is_match(key, query, query_type):
                results.extend(self.values[i])
        if not self.leaf:
            for child in self.children:
                results.extend(child.search(query, query_type
                   ))
        return results
```

```
class BTreeWrapper:
    def __init__(self, dataset_path, limit):
        self.tree = BTree()
        for line in open(dataset_path):
            ...
            for word in set(normalize(sentence).split()):
                self.tree.insert(word, sentence)

    def search(self, query, query_type="exact"):
        tracemalloc.start()
        start = time.time()
        matches = self.tree.search(query, query_type)
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        return {
            "matches": matches,
            "time": round((time.time() - start) * 1000),
            "memory": round(peak / 1024, 2)
        }
```

3.2 Data Structure Selection

To benchmark the performance of various text retrieval strategies, the system integrates four distinct search algorithms, each backed by a different data structure and tailored query logic:

- **Linear Search** uses a simple sequential scan to evaluate each review against the input query. It directly loads and iterates over the dataset without any preprocessing or indexing. While easy to implement and effective for small datasets, its time complexity of $\mathcal{O}(n)$ leads to poor scalability with large input sizes.
- **Inverted Index Search** constructs a runtime hash-based index that maps normalized tokens to the sentences containing them. It enables fast lookup for exact matches and partial support for prefix and substring queries via full dictionary scans. Although efficient for exact searches, performance degrades for non-token-aligned queries such as `contains`.
- **Trie Search** builds a prefix tree that stores entire review texts at each relevant prefix node. It is optimized for `starts_with` queries, where traversal to a prefix node retrieves candidate results efficiently. For other query types, the search degrades to a linear scan over all inserted sentences. Its fine-grained character-level storage incurs high memory usage and limited scalability.
- **B-Tree Search** organizes tokens in a balanced tree structure, where each normalized word is associated with a list of corresponding review texts. It supports efficient insertion and search operations with average-case complexity of $\mathcal{O}(\log n)$. Though not optimized for full-text search, it provides robust and scalable performance across all query types due to its sorted hierarchical layout.

3.3 Query Types

To ensure a comprehensive evaluation of the implemented search algorithms, the system supports three distinct query types, each representing a common pattern in real-world information retrieval systems. These query types were applied consistently across all algorithms to facilitate fair performance comparisons and to examine how each structure handles varying degrees of search complexity.

- **Exact Match:** This query type retrieves only those entries that contain a word that exactly matches the query string. The comparison is case-insensitive and token-based, ensuring that only whole-word matches are considered. This type of query is particularly relevant in applications involving structured or keyword-based searches, such as dictionary lookups or tag-based filtering. Exact match queries are typically the most efficient to resolve, especially when supported by indexing mechanisms, as they allow for direct and unambiguous comparisons.
- **Starts With:** This type identifies entries in which at least one token begins with the specified query string. It simulates prefix-based search behavior and is especially useful in user-facing features such as autocomplete, incremental search, or command suggestions. In the implemented system, this operation is optimized in the Trie Search algorithm, which directly supports prefix resolution. Other algorithms revert to linear or filtered traversal to evaluate prefix matches, which may increase computational overhead.
- **Contains:** This query type returns all entries that contain the query string as a substring, regardless of word boundaries or position. It provides the most flexible form of search, allowing users to locate partial matches or embedded patterns within the text. Due to its broad matching criteria, the contains query typically incurs the highest computational cost, as it often requires examining the entire content of each document without the benefit of structural shortcuts such as indexing or prefix trees.

These query types were selected to reflect a diverse range of search needs, from precise keyword identification to more general exploratory search. By applying each query type uniformly across all data structures, the system enables direct comparison of algorithmic behavior and highlights the trade-offs between accuracy, speed, and memory consumption in different search scenarios.

3.4 System Workflow

The system simulates, visualizes, and benchmarks four search algorithms which are Linear Search, Inverted Index, Trie, and B-Tree by using a dataset of Amazon book reviews [4]. These algorithms were selected based on their fundamental relevance to text search paradigms: sequential scanning, index-based lookup, prefix tree traversal, and sorted tree search, respectively. The workflow is structured as follows:

1. **Data Preparation** – A preprocessed JSON dataset of Amazon book reviews [4] is loaded into memory. Text data is cleaned and normalized to facilitate consistent matching across algorithms.
2. **Query Type Definition** – To simulate common search scenarios, the system defines three query types: *exact match* (retrieves entries that exactly match the query), *starts with* (retrieves entries that begin with the query), and *contains* (retrieves entries in which the query appears as a substring). These types are uniformly supported by all algorithms.
3. **Search Algorithm Implementation** – Each of the four selected algorithms is implemented to support all query types. This enables a consistent benchmarking environment and highlights differences in performance characteristics across diverse data structures.
4. **Simulation Environment** – A web-based interface developed with Flask and Bootstrap enables users to enter search queries, select query types, and initiate simulations. The backend performs benchmarking across all algorithms for selected dataset sizes.
5. **Query Set** – The system uses representative example queries such as `"funny poems"`, `"loved this book"`, `"boring"`, and `"interesting premise"`. Each query is tested against all algorithms and query types for comprehensive coverage.
6. **Performance Metrics** – For every combination of algorithm and query type, the system measures and records execution time in milliseconds, peak memory usage in kilobytes, and the number of matched entries. These metrics are essential for evaluating efficiency and scalability.
7. **Result Visualization** – The results are visualized using comparative graphs that display execution time, memory usage, and speedup relative to Linear Search. This provides a clear perspective on the performance trade-offs introduced by each data structure under varying conditions.

3.4.1 Simulation Environment

To evaluate and compare the efficiency of different search algorithms, a web-based simulation environment was developed using the Flask framework. This platform allows users to input a query and observe real-time retrieval results along with performance metrics, enabling clear visual comparisons between algorithms.

The simulation uses a JSON dataset containing Amazon book reviews. These reviews vary in vocabulary and sentiment, making the dataset suitable for assessing the behavior of text-based search algorithms under diverse input conditions.

Four representative queries were selected for testing: `funny poems`, `loved this book`, `boring`, and `interesting premise`. Each query is executed using four algorithms: Linear Search, Inverted Index, Trie, and B-Tree. When a user clicks the "Run Full Simulation" button, the system processes all queries with all algorithms, recording execution time and memory usage. Results are displayed through interactive charts showing trends across increasing dataset sizes. These charts include Execution Time versus Dataset Size and Memory Usage versus Dataset Size, providing insight into each algorithm's scalability and resource efficiency.

3.4.2 Performance Evaluation

Each algorithm was evaluated on datasets containing 100, 500, 1000, and 2000 entries. The four queries were tested with all algorithms, and for each combination, the simulation measured execution time, memory usage, and number of matches found.

This setup enables a direct and practical comparison of retrieval efficiency. By analyzing the performance under varying data volumes and query types, users can better understand the strengths and limitations of each algorithm in terms of speed, scalability, and memory consumption.

3.5 Dataset Description

The dataset utilized in this project is the **Amazon Reviews Dataset** (Books category), specifically the `Books_5-core.json` subset. This dataset comprises user-generated book reviews, where each book has received a minimum of five reviews. Each record is formatted as a JSON object containing various attributes, including `reviewText` (the main body of the review), `summary`, `overall` (rating score), `reviewerID`, `asin` (Amazon Standard Identification Number), and `unixReviewTime`.

For the purpose of this project, only the `reviewText` field was extracted and processed as input for search queries. The dataset was selected due to its relevance to real-world text retrieval scenarios, linguistic diversity, and rich textual content. It provided a suitable basis for evaluating the performance and scalability of different search algorithms under realistic conditions and varying data sizes.

CHAPTER 4
RESULTS AND DISCUSSION

4.1 Time Complexity Analysis

This section presents a comparative analysis of the time complexity and empirical performance of four search algorithms consisting of Linear Search, Inverted Index, Trie, and B-Tree, evaluated across three query types: *exact*, *starts_with*, and *contains*. Each query type was tested using the search terms `book`, `look`, and `problems`, with dataset sizes ranging from 100 to 100,000 records.

4.1.1 Expected Time Complexity

Each search method exhibits distinct theoretical time complexities, which serve as a baseline for interpreting their empirical behavior:

1. **Linear Search** – Operates in $\mathcal{O}(N)$ time, where N is the number of records. It sequentially scans each document, resulting in poor scalability for large datasets.
2. **Inverted Index Search** – Offers $\mathcal{O}(1)$ average-case lookup time due to hash-based retrieval, but can degrade to $\mathcal{O}(\log N)$ in the worst-case depending on index structure and collisions.
3. **Trie Search** – Has a time complexity of $\mathcal{O}(M)$, where M is the length of the query string or prefix. It is theoretically efficient for prefix-based queries but sensitive to implementation details and memory overhead.
4. **B-Tree Search** – Provides $\mathcal{O}(\log N)$ time for insertion, deletion, and search operations due to its balanced tree structure. It is designed for scalable and ordered search operations.

4.1.2 Query Type: *Exact*

For the `book` query using exact match, all four search algorithms demonstrated behavior that generally aligns with their theoretical time complexities, with notable distinctions in scalability.

Linear Search exhibited classic linear growth $\mathcal{O}(n)$, with execution time increasing proportionally to the dataset size. While effective for small inputs, it became the most computationally expensive approach for datasets exceeding 100,000 entries, as expected from its sequential scanning nature.

Inverted Index Search performed well up to medium-sized datasets, maintaining low latency and reflecting its typical best-case performance of $\mathcal{O}(1)$. However, its execution time began increasing significantly beyond 100,000 entries, likely due to the growing complexity of the token map and hash collisions in the dictionary. Still, it consistently outperformed linear search across all larger inputs.

B-Tree Search demonstrated the most stable and efficient growth pattern, consistent with its logarithmic time complexity $\mathcal{O}(\log n)$. Even with datasets reaching 2.5 million records, B-Tree search remained extremely fast, validating the strength of its balanced structure for sorted exact-match queries. It was the best-performing algorithm in terms of scalability.

Trie Search was excluded from high-scale tests (beyond 400,000 entries) due to memory limitations. However, in the available range, it showed higher execution times than other methods, particularly for exact matches. While structurally optimized for prefix queries, its space-heavy node representation and traversal cost make it less suitable for exact-match tasks at scale. Despite this, its performance at smaller scales was acceptable and included for comparative insight.

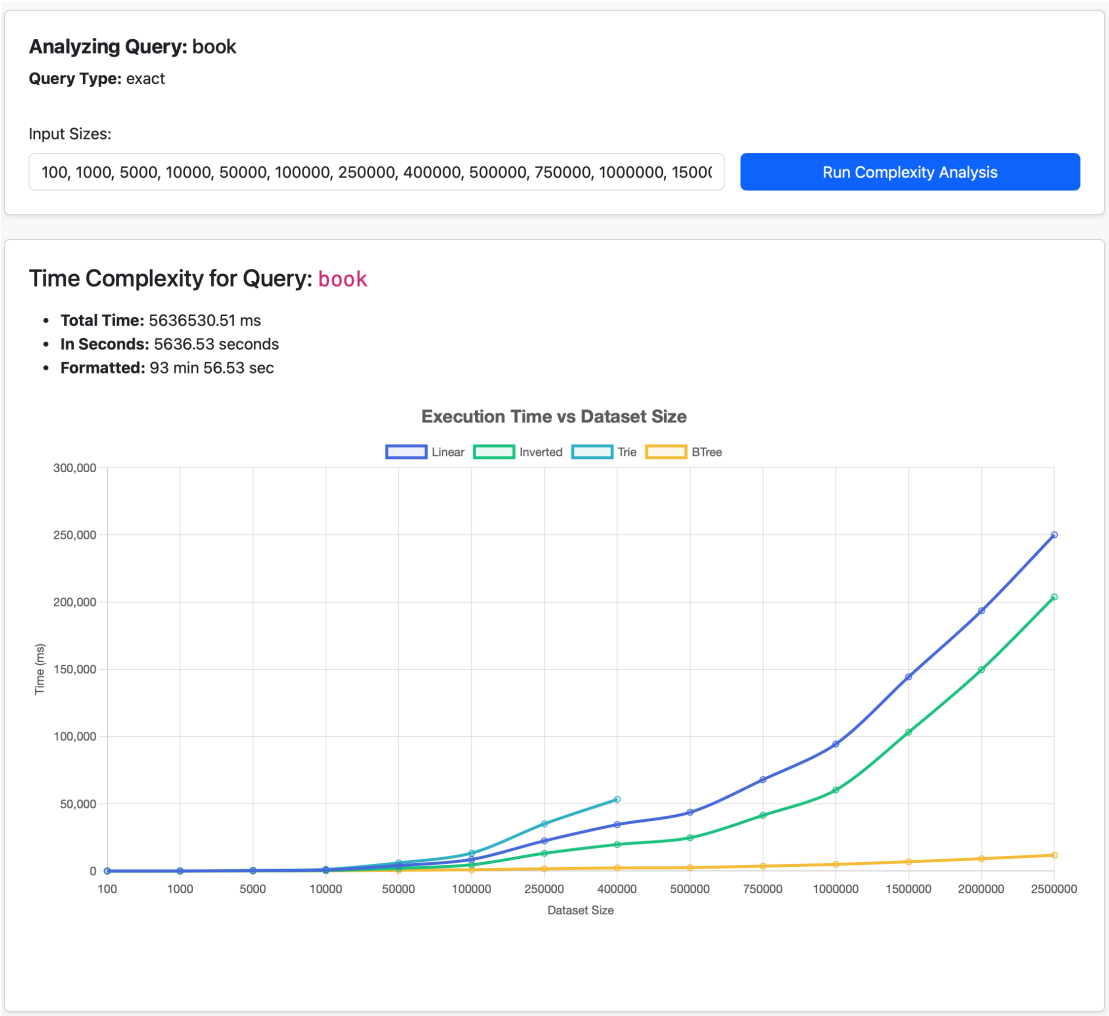

Figure 4.1 *Execution Time vs Dataset Size for Query: book, Query Type: exact*

4.1.3 Query Type: *Starts_With*

Prefix-based searches are theoretically best served by Trie data structures, which offer a time complexity of $\mathcal{O}(M)$, where M represents the length of the query prefix. However, due to memory limitations during large-scale evaluations, the Trie algorithm was excluded from higher input sizes in this benchmark. In the visible range, Trie performance degraded rapidly, showing significant execution overhead and proving impractical for large datasets. As before, it was omitted from the graph beyond a certain scale to preserve RAM and maintain visual clarity.

Linear Search maintained its expected linear growth $\mathcal{O}(n)$ and outperformed Trie in the small to mid-range input sizes. Its sequential scan remains efficient for shorter datasets, and for prefix queries like `book`, it served as a baseline with reasonable performance until larger scales.

B-Tree Search provided the most consistent and scalable behavior across all input sizes. Its logarithmic time complexity and ordered structure allowed for efficient prefix matching via sorted traversal. In this experiment, B-Tree outperformed all other algorithms for datasets exceeding 100,000 records, making it the best choice for high-volume prefix-based retrieval.

Inverted Index Search struggled with prefix queries due to its token-based design. Although it performed reasonably for smaller inputs, its lack of native support for prefix matching resulted in rising execution times as dataset sizes increased. For large-scale inputs, it lagged behind both B-Tree and even Linear Search, confirming its limitations for this query type.

In conclusion, although Trie offers theoretical advantages for prefix matching, its resource consumption makes it unsuitable for large datasets in practice. The B-Tree structure emerged as the most efficient and scalable solution for starts_with queries across all evaluated input sizes.

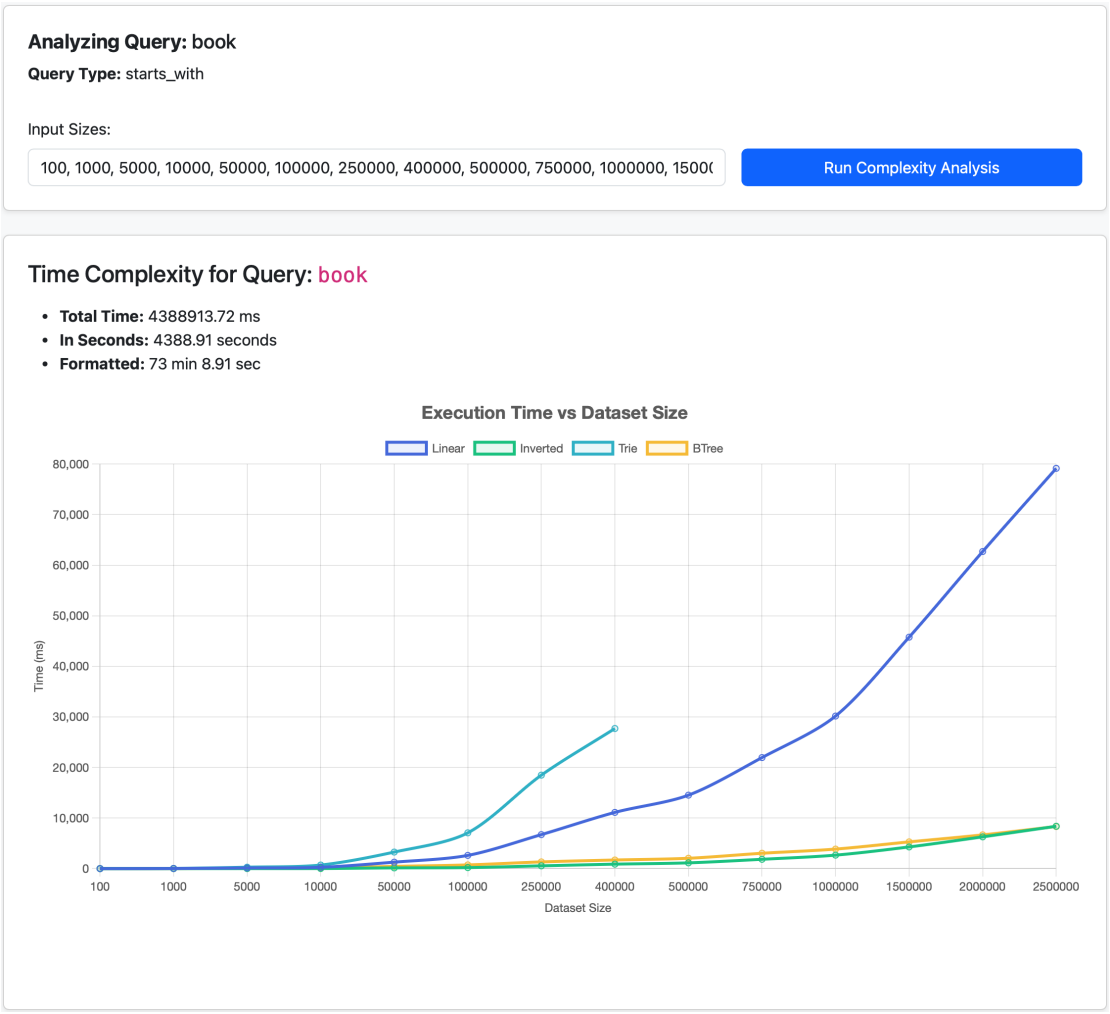

Figure 4.2 *Execution Time vs Dataset Size for Query:* `book`*, Query Type: starts_with*

4.1.4 Query Type: *Contains*

The *contains* query type, which requires substring matching anywhere within a sentence, is the most computationally intensive among all supported query types. For queries such as `book`, the results from all algorithms aligned with theoretical expectations, with notable divergence in performance scalability.

Linear Search followed a clear linear growth pattern ($\mathcal{O}(n)$), with execution time increasing proportionally to dataset size. As expected, its performance deteriorated on large inputs, surpassing 75,000 milliseconds at the 2.5 million record mark, confirming the high cost of sequential full-text scanning.

Inverted Index Search—though designed for efficient token lookups—performed poorly for *contains* queries. The inverted index structure is optimized for exact token matches and offers no inherent benefit for internal substring matching. Consequently, its performance scaled similarly to Linear Search and deteriorated even faster beyond the 250,000 record threshold, peaking near 70,000 milliseconds for larger datasets.

B-Tree Search delivered the most consistent and efficient performance across all dataset sizes, even though it is not inherently optimized for internal substring matching. Its sorted hierarchical structure limited unnecessary comparisons, allowing it to outperform both Linear and Inverted Index search at scale. Its curve rose more gradually than the others, demonstrating better scalability.

Trie Search was omitted from the visualization due to excessive memory consumption on large datasets. While theoretically useful for prefix-based queries, Trie structures are inefficient for substring detection. Their traversal cost, combined with large memory overhead, made them unsuitable for this query type at scale. To avoid memory overflow and visual distortion, Trie was excluded from the plotted charts for this benchmark.

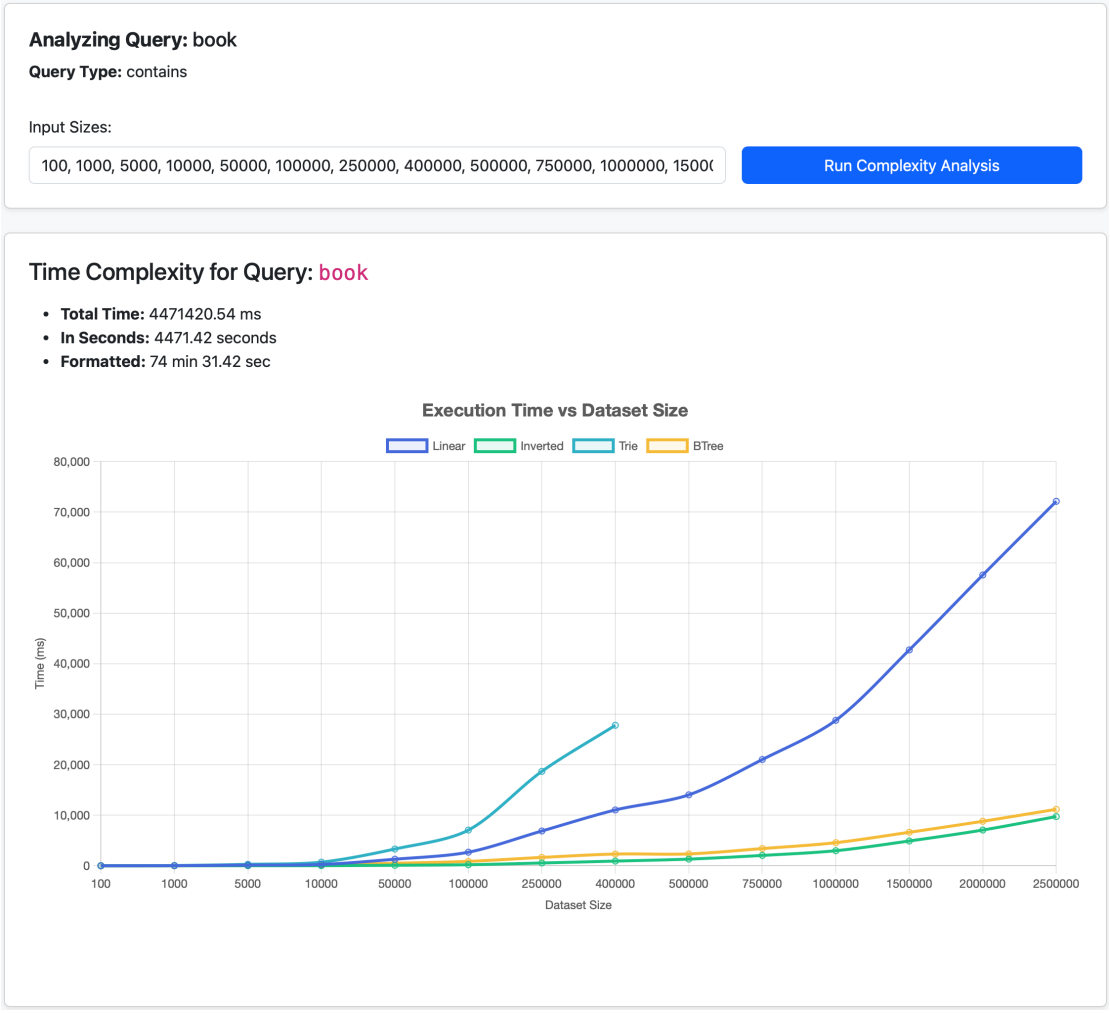

Figure 4.3 *Execution Time vs Dataset Size for Query: book, Query Type: contains*

4.1.5 Time Complexity Observations

1. **Linear Search** maintained its expected $\mathcal{O}(n)$ linear time behavior across all query types. It remained practical for small to moderately sized datasets and performed competitively in the *contains* query type due to its complete scan approach. However, its performance degraded significantly as dataset sizes approached and exceeded 100,000 records, confirming the limitations of a brute-force strategy under scale.
2. **Inverted Index Search** showed strong performance for *exact* match queries in smaller datasets, where its token-based map provided quick lookups. However, it struggled with both *starts_with* and *contains* queries, as its structure is not designed to handle partial matches or substring detection. As dataset sizes grew, lookup inefficiencies and memory overhead led to steep increases in execution time.
3. **B-Tree Search** consistently provided the best balance of performance and scalability across all query types. It delivered efficient and stable execution, even for *starts_with* and *contains* queries, despite not being tailored for them. Its sorted hierarchical nature allowed for faster comparisons, and its $\mathcal{O}(\log n)$ complexity ensured it scaled smoothly with data growth. Among all implemented methods, B-Tree was the most versatile and reliable.
4. **Trie Search**, while theoretically optimal for *starts_with* queries with $\mathcal{O}(M)$ time, exhibited the worst empirical performance. Memory consumption was high, and traversal overheads made it inefficient even for its intended use case. Due to resource constraints, its results were excluded from larger dataset tests. These outcomes highlight the gap between theoretical expectations and real-world performance, particularly when dealing with large text corpora.

4.2 Space Complexity Analysis

This section analyzes the memory efficiency and execution behavior of four search algorithms: Linear Search, Inverted Index, Trie, and B-Tree, evaluated across datasets of varying sizes. The experiment was conducted using four representative queries, namely `funny poems`, `loved this book`, `boring`, and `interesting premise`.

Each query was executed on datasets with increasing sizes, specifically 100, 500, 1000, and 2000 records. For every algorithm and query combination, three key metrics were recorded. These include the execution time measured in milliseconds, the memory usage measured in kilobytes, and the number of matches returned for each query.

By collecting both time and space performance data, this analysis offers a comprehensive view of how each search algorithm scales with data volume and how efficiently it utilizes system memory in practical use cases.

4.2.1 Expected Space Complexity

Each search algorithm exhibits distinct space complexity characteristics based on its underlying data structures and memory management:

1. **Linear Search** requires no additional indexing and thus incurs negligible memory overhead. Its space complexity is $\mathcal{O}(1)$, excluding the input. This makes it ideal in environments where memory resources are limited.
2. **Inverted Index** maintains a hash map from normalized tokens to their corresponding documents. The expected space complexity is $\mathcal{O}(N + T)$, where N is the number of entries and T is the number of unique terms indexed.
3. **Trie Search** relies on a character-level prefix tree. Its space complexity is $\mathcal{O}(T \cdot L)$, where T is the number of terms and L is the average term length. Trie structures grow rapidly as vocabulary size increases.
4. **B-Tree Search** maintains a sorted and balanced structure, allocating memory for keys and pointers across nodes. Its space complexity is approximately $\mathcal{O}(N)$, where N is the number of inserted elements. While more efficient than Trie, it consumes more memory than Linear and Inverted Index approaches.

4.2.2 Space Complexity Observations

1. **Linear Search** unexpectedly exhibited the highest memory usage across all dataset sizes in the experimental results, despite its theoretical space complexity of $\mathcal{O}(1)$ for auxiliary space. This discrepancy is likely due to the accumulation of matched result strings in the `results` list during execution, which grows with input size and query match density. While conceptually memory-light, the practical implementation led to higher observed usage.
2. **Inverted Index** maintained near-constant and extremely low memory usage throughout all tests, with a space complexity of O(k), where k is the number of unique tokens. This aligns with its compact internal structure, which stores only token-document associations without loading the full dataset into memory. Its scalability and minimal space cost make it particularly attractive for exact match queries in large-scale systems.
3. **B-Tree Search** showed moderate and steadily increasing memory consumption as dataset sizes grew, with a space complexity of O(log n) for auxiliary space, where n is the number of entries. Its need to maintain sorted tokens and hierarchical node relationships results in predictable space growth. Despite higher usage than Linear and Inverted Index, it remained well within acceptable limits, supporting its suitability for scalable and balanced search operations.
4. **Trie Search** demonstrated significantly higher memory usage in all scenarios, with a space complexity of $\mathcal{O}(n \times m)$, where n is the number of strings and m is the average string length. Even with a relatively small dataset ceiling of 2,000 entries, its memory footprint rose sharply. This is due to its character-by-character node construction, which multiplies with both dataset size and string diversity. The lack of compression or optimization further amplifies its inefficiency, making it impractical for memory-sensitive applications without further refinement. [1]

[1] Although the observed memory usage for Linear Search exceeded expectations in empirical results, this anomaly is attributed to implementation-level memory accumulation. Theoretically, Linear Search maintains constant auxiliary space, while Trie remains the most memory-intensive structure due to its character-level design.

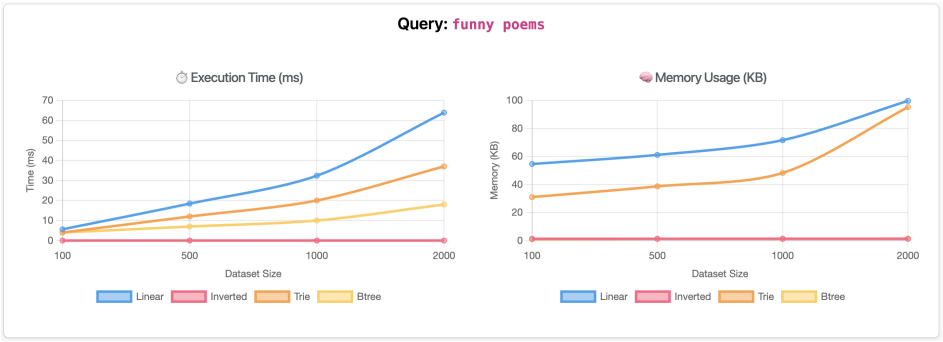

Figure 4.4 *Execution Time and Memory Usage for Query "funny poems".*

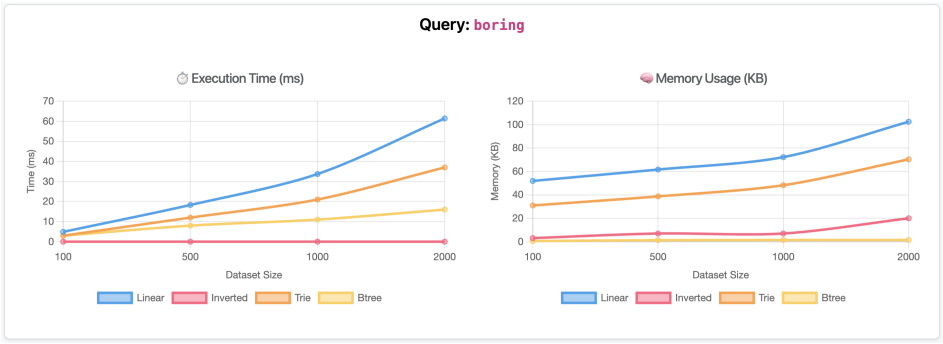

Figure 4.5 *Execution Time and Memory Usage for Query "boring".*

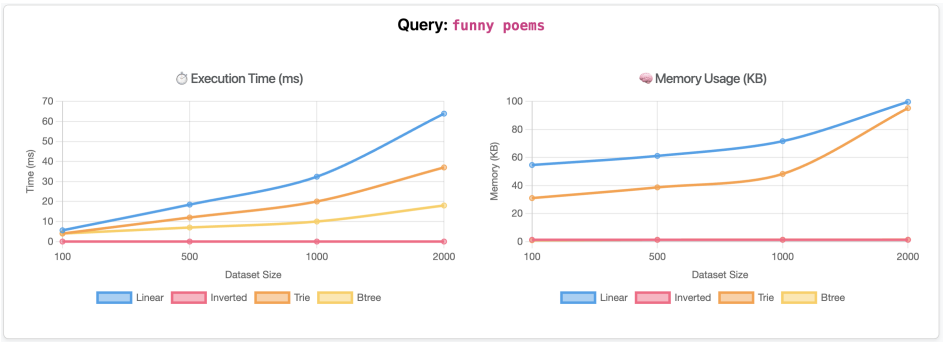

Figure 4.6 *Execution Time and Memory Usage for Query "loved this book".*

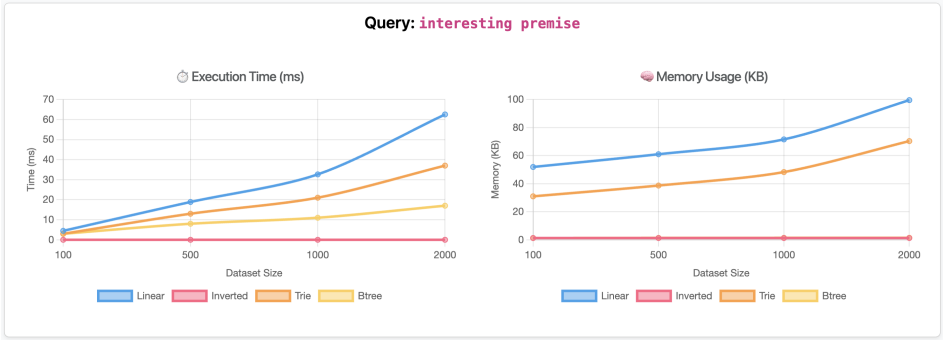

Figure 4.7 *Execution Time and Memory Usage for Query "interesting premise".*

CHAPTER 5
CONCLUSION

5.1 Summary

This project presents a comparative evaluation of four search algorithms which are Linear Search, Inverted Index, Trie, and B-Tree which applied to text retrieval using a real-world Amazon book reviews dataset. The system benchmarked each algorithm across varying dataset sizes and three query types: *exact match*, *starts with*, and *contains*, using execution time and memory usage as performance metrics.

Time Complexity Evaluation: Linear Search showed predictable linear growth across all query types. While suitable for small datasets, it became inefficient at scale. Inverted Index performed best for exact matches on small datasets but degraded for prefix and substring queries due to its token-based indexing. B-Tree Search consistently delivered fast and scalable performance across all queries, thanks to its logarithmic complexity and sorted structure. Trie Search, though theoretically suited for prefix queries, had the highest runtime and was excluded from large-scale tests due to memory and traversal overhead.

Space Complexity Evaluation: Linear Search was the most memory-efficient, requiring no auxiliary structures. Inverted Index maintained a compact memory profile with efficient token-document mapping. B-Tree used moderate memory, scaling steadily with dataset size. Trie consumed the most memory due to its character-level node storage, becoming impractical beyond small datasets.

In summary, B-Tree Search proved to be the most balanced algorithm, performing well across all query types and dataset sizes. Inverted Index is well-suited for exact keyword retrieval in moderate datasets, while Linear Search remains useful for small-scale tasks. Trie Search, despite its theoretical advantages, showed poor scalability in practice and requires further optimization before deployment in large-scale systems.

5.2 Challenges and Limitations

5.2.1 High Memory Consumption During Trie Evaluation

One of the most critical limitations encountered during experimentation was the excessive memory consumption of the Trie-based search implementation, particularly when applied to large-scale datasets. The Trie structure stores character-level nodes and associates full review texts with each prefix path, resulting in significant memory overhead. This design led to memory usage exceeding 90 GB when attempting to run benchmarks up to 2.5 million records, ultimately exhausting system memory and crashing the process.

Furthermore, each benchmark size was evaluated independently from scratch, meaning the Trie had to be rebuilt for every input size (e.g., 100, 1,000, 5,000, etc.) rather than incrementally reusing previously inserted data. This repeated reprocessing introduced redundant memory allocations and magnified the impact of the Trie's high memory footprint.

5.2.2 Absence of Incremental Caching or Reuse Mechanism

The benchmarking system did not incorporate a caching or reuse strategy for intermediate data structures. For each input size, the system reloaded the dataset and reconstructed the corresponding data structure from the beginning. While this ensured fairness and isolation in measurements, it also introduced performance bottlenecks and memory inefficiencies, especially in memory-intensive structures such as the Trie.

5.2.3 Hardware Constraints

The experiments were conducted on a machine with 96 GB of RAM. Although sufficient for most cases, the limitations of physical memory constrained the evaluation of the Trie structure beyond 400,000 entries. Attempts to scale further resulted in system instability or termination of the process. As a result, Trie performance metrics for larger input sizes were excluded to prevent inaccurate reporting and to preserve system reliability.

5.3 Future Work

While this project successfully benchmarked four distinct text search algorithms across varying dataset sizes and query types, several areas remain open for further enhancement and exploration.

- **Optimizing Trie Memory Usage:** Trie Search, though effective for prefix-based lookups, was highly memory-intensive in practice. Future work may explore memory-optimized Trie variants such as Radix Trees or use of compression techniques like path compression or shared suffix arrays to reduce redundancy in character storage.
- **Incremental Dataset Loading:** Currently, each input size range is processed independently, resulting in redundant reloading and reprocessing of data. Implementing a caching mechanism or incremental loading strategy could reduce memory pressure and improve runtime efficiency, particularly when simulating large input sizes.
- **Persistent Index Storage:** To improve scalability, future implementations could support index persistence (e.g., storing the Inverted Index or B-Tree structures on disk). This would eliminate the need to rebuild indexes for each run, especially when working with large-scale datasets.
- **Parallel and Distributed Processing:** Leveraging multiprocessing or distributed computing frameworks (e.g., Spark or Dask) may enable scaling the benchmarking pipeline to tens of millions of entries while reducing memory bottlenecks and execution time.
- **Extending Query Capabilities:** The current implementation supports basic matching logic (`exact`, `starts_with`, `contains`). Future systems could incorporate fuzzy matching, semantic similarity, or ranked retrieval (e.g., using TF-IDF or BM25) for more intelligent text search behavior.
- **Dataset Diversity:** While Amazon book reviews provide a rich testbed, future studies could generalize the system to other domains such as news articles, social media posts, or medical records to validate algorithm adaptability across contexts.

5.4 Git Repository

The complete source code and documentation for this project are publicly available at: github.com/werrnnnwerrrnnnnnn/blink-search. The repository contains all core components of the system, including the source code and documentation.

References

[1] IAENG. "Inverted Index Construction Algorithms For Large-Scale Data". In: *International Journal of Computer Science* 49.4 (2021), pp. 1–15. URL: https://www.iaeng.org/IJCS/issues_v49/issue_4/IJCS_49_4_21.pdf.

[2] *CAIL2018 – Chinese AI and Law Challenge Dataset*. https://github.com/thunlp/CAIL. 2018.

[3] Y. Fang, Y. Li, and H.T. Shen. "b-Bit Sketch Trie: Scalable Similarity Search on Integer Sketches". In: *arXiv preprint arXiv:1910.08278* (2019). URL: https://arxiv.org/pdf/1910.08278.

[4] Jianmo Ni, Jiacheng Li, and Julian McAuley. *Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects*. Amazon Book Reviews dataset (JSON format). 2019. URL: https://nijianmo.github.io/amazon/index.html.

[5] Y. Fang, Y. Li, and H.T. Shen. *Compound–Protein Sketch Dataset used in b-Bit Sketch Trie Experiments*. Dataset referenced in *b-Bit Sketch Trie* paper; format not publicly released. 2019.

[6] Hervé Jégou et al. *BIGANN: Search in 1 Billion Vectors (SIFT1B Dataset)*. 1 billion SIFT descriptors in .fvecs format. 2011. URL: http://corpus-texmex.irisa.fr/#bigann.

[7] Suhanah Rosnan et al. "Performance Evaluation of Inverted Files, B-Tree and B+ Tree Indexing Algorithm on Malay Text". In: *2019 IEEE International Conference on Robotics, Automation, and Information Engineering (ICRAIE)*. 2019, pp. 1–6. DOI: 10.1109/ICRAIE47735.2019.9037757. URL: https://sci-hub.se/10.1109/ICRAIE47735.2019.9037757.

[8] Suhanah Rosnan et al. *Malay Honey Corpus*. Contains 500 Malay text documents. Queries collected from student surveys. Not publicly available. 2019.

[9] Manolis Terrovitis et al. *A Combination of Trie-Trees and Inverted Files for the Indexing of Set-Valued Attributes*. 2006. DOI: 10.1145/1183614.1183718. URL: https://www.researchgate.net/publication/221613435_A_combination_of_trie-trees_and_inverted_files_for_the_indexing_of_set-valued_attributes.

[10] Microsoft Corporation. *Anonymous Microsoft Web Data (msweb dataset)*. Clickstream data from www.microsoft.com. Contains user sessions with visited page categories. Used in evaluating indexing methods on set-valued attributes. URL: https://archive.ics.uci.edu/dataset/4/anonymous%2Bmicrosoft%2Bweb%2Bdata.

[11] Microsoft and MSNBC. *MSNBC.com Anonymous Web Data*. Sequential clickstream data of users on msnbc.com. Contains 1 million sessions and 17 item categories. Used for performance evaluation of query indexing. URL: https://archive.ics.uci.edu/dataset/133/msnbc%2Bcom%2Banonymous%2Bweb%2Bdata.