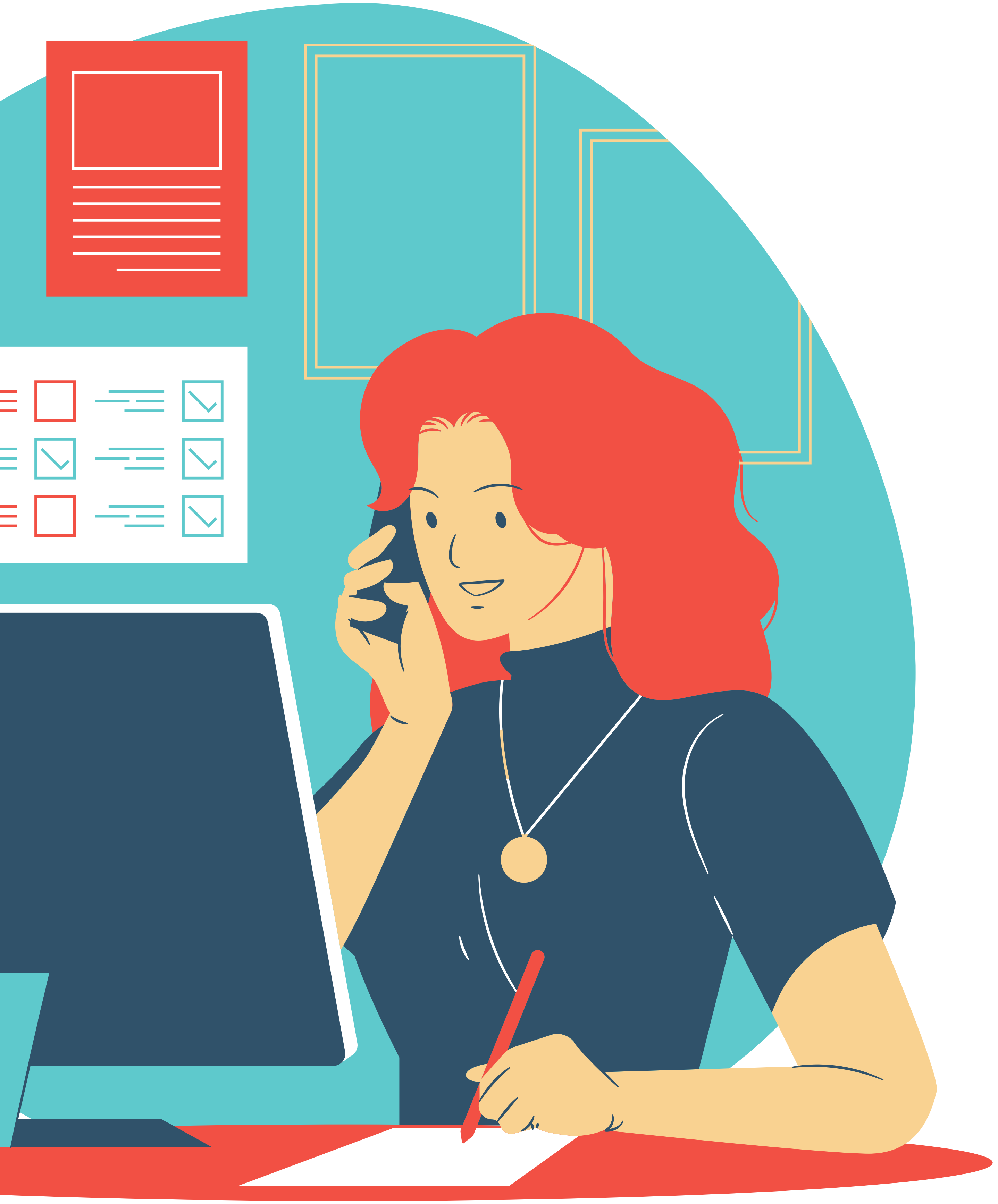
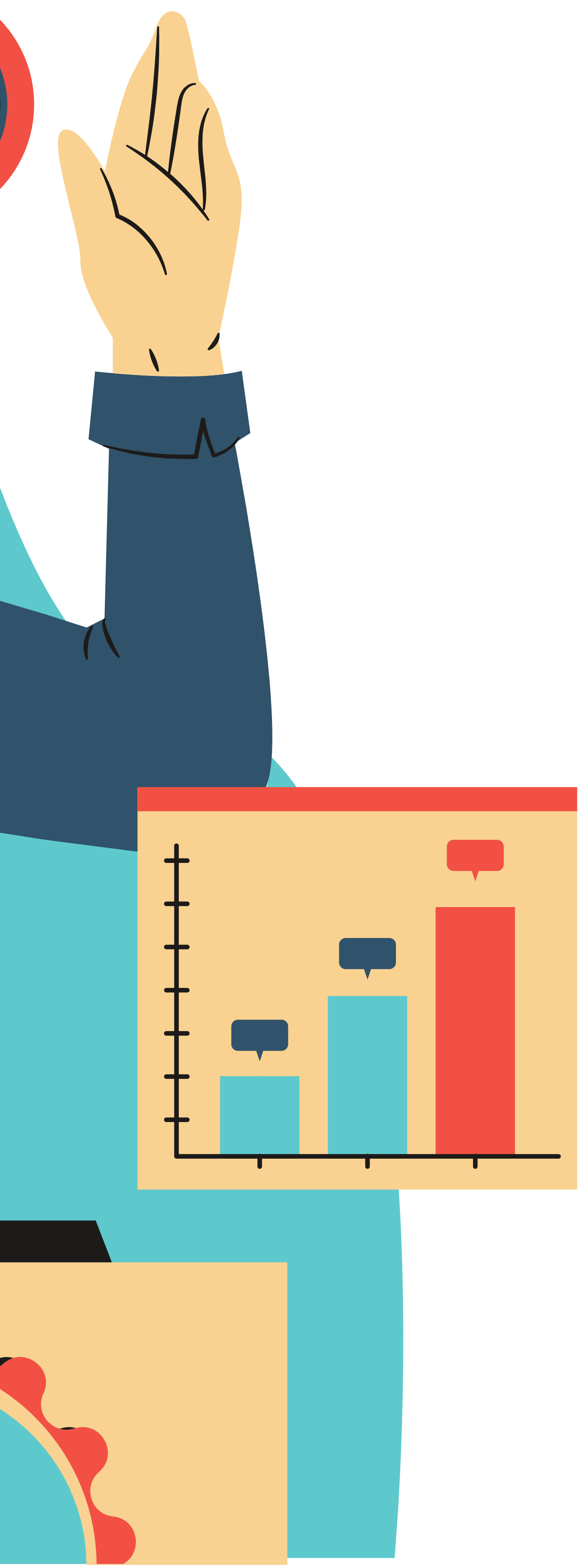


BLINKSEARCH

Benchmarking Text Search Algorithms

- Tatiya Seehatrakul
- Shreeyukta Pradhanang
- Group Name : HeHe

TABLE OF CONTENTS



01 Introduction

PROBLEM STATEMENT

- Large datasets make simple text search (Linear Search) inefficient.
- Indexing is essential for fast lookups, but various methods exist.
- Modern applications demand efficient prefix-based search (e.g., autocomplete).
- Choosing an indexing structure involves trade-offs between speed, memory, and scalability.

OBJECTIVES

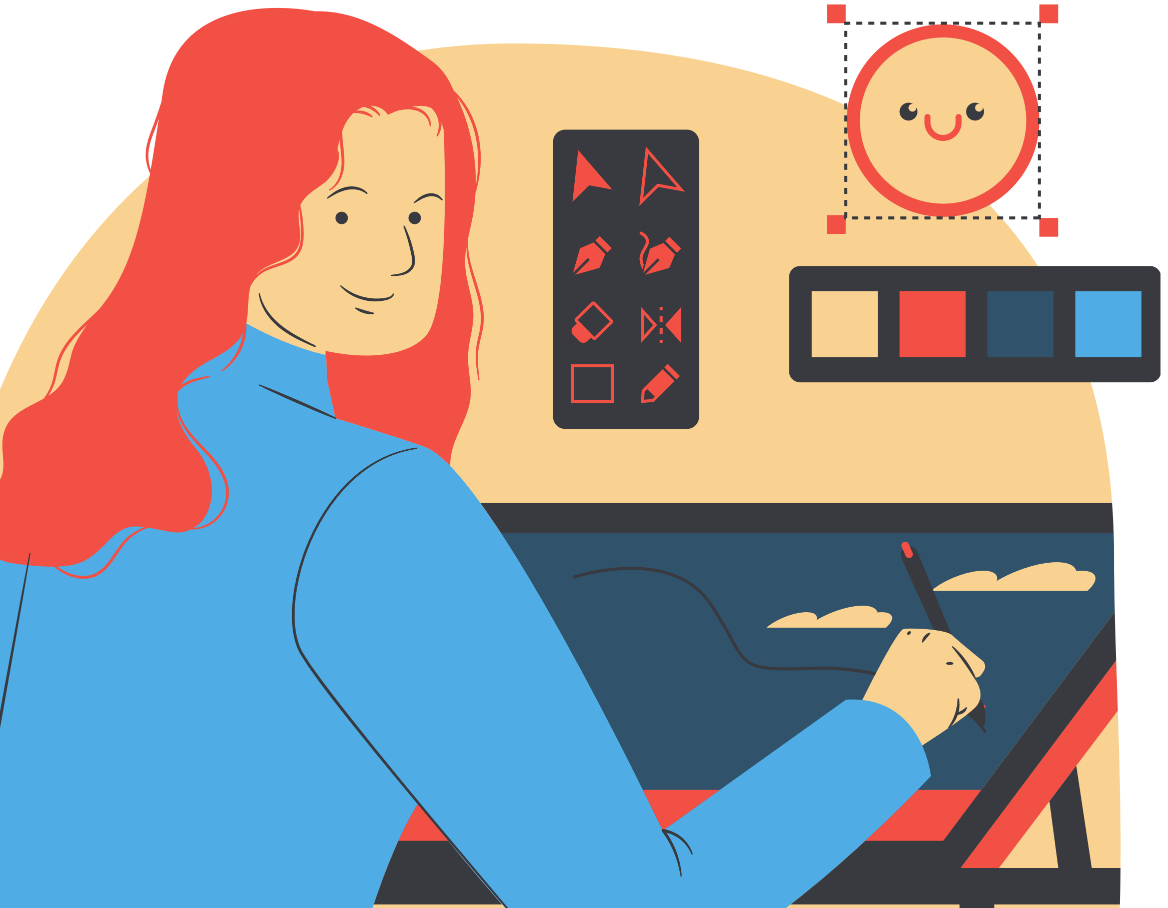
1. Implement and compare 4 search algorithms (Linear Search, Inverted Index Search, Trie Search, B-Tree Search

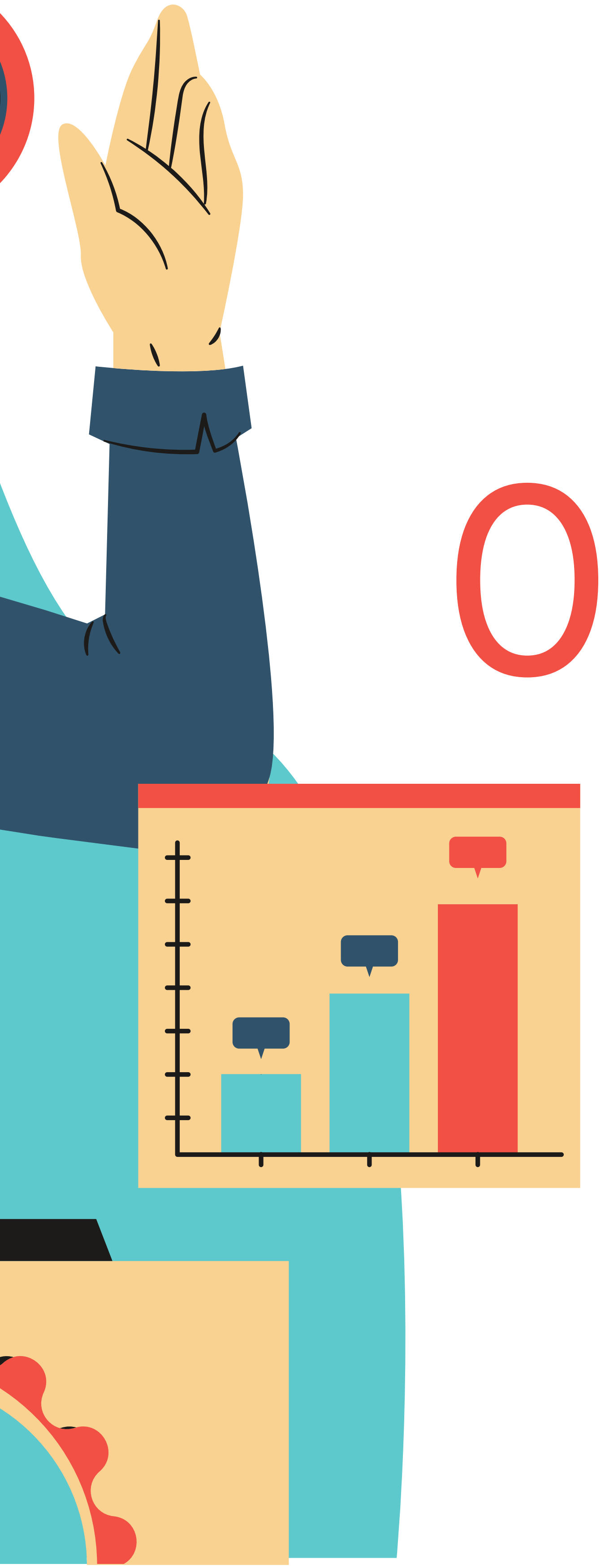
2. Measure and evaluate performance (execution time, memory usage, match count, time complexity, space complexity)

3. Implement 3 query types for performance analysis (Exact, Starts With, Contains)

PROJECT SCOPE

- Develop a web-based search system that applies multiple search algorithms to retrieve articles efficiently.
  - Linear Search – Baseline method, sequential lookup.
  - Inverted Index – Used in Google Search & Elasticsearch for fast keyword lookups.
  - Trie Search – Efficient for prefix-based searches and autocomplete.
  - B-Tree Search – Used in MySQL for structured and range queries.
- Evaluate execution time, memory usage, and scalability of each search method.





02 Literature Review

Wang et al.

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

- **Techniques Introduced:**
  - **FASTER-INV**: A refined inverted index optimized for speed and memory efficiency during keyword search.
  - **AC-INV**: A hybrid approach that combines an inverted index with the Aho-Corasick algorithm for prefix-based search by integrating trie structures.
- **Dataset Used** : **CAIL2018** (A Chinese legal corpus in JSON format)
  - Dataset size: 200,000 to 1.5 million documents.
  - Average document length: 660–693 words.
- **Scalability Evaluation:**
  - Document counts scaled from 300k to 1.5M.
  - Metrics collected: indexing time, memory usage, speedup.
  - Results visualized using graphs and tables.

Wang et al.

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

- **Key Findings:**
  - FASTER-INV:
    - Achieved up to 1.14× speedup.
    - Reduced memory usage by 10%.
  - AC-INV:
    - Reached 1.43× speedup.
    - Reduced memory usage by 35%.
- **Conclusion**: Both methods improved performance and memory usage, showing the scalability and efficiency of combining trie-based and token-based search for large-scale text retrieval.

Kanda & Tabei

2. Linear, Inverted Index, Trie-based Search with Sketch Structures

- **Technique Proposed:**
  - **b-bit Sketch Trie (bST):** A compact and scalable structure for fast similarity search over large integer-based sketch datasets.
- **Compared Methods:**
  - Linear Search
  - Inverted Index Search: Single Index Hashing (SIH)
  - Trie-based Search: SI-bST and MI-bST
- **Datasets Used:**
  - **Amazon Book Reviews**
  - **Compound–Protein Sketches (CP)**
  - **SIFT1B descriptors from BIGANN**
  - **Review sketches using b-bit MinHash or Consistent Weighted Sampling (CWS)**
- Methodology:
  - Sketches generated using b-bit MinHash or 0-bit CWS
  - 10,000 queries per dataset
  - Metrics: query time, memory usage, scalability vs threshold τ
  - Visualized with bar charts and line graphs

Kanda & Tabei

2. Linear, Inverted Index, Trie-based Search with Sketch Structures

- **Key Results:**
  - SI-bST outperformed others on Review, CP, and SIFT1B, completing billion-scale queries with just 9 GiB of memory
  - MI-bST was slower on GIST due to sketch length and threshold requirements
  - Linear Search was fast but didn't scale well
  - SIH had high memory overhead and poor scalability

- **Conclusion**: bST-based search methods offered the best balance of speed and memory, especially for similarity-based large dataset searches.

Rosnan et al.

3. Inverted Index and B-Tree Search on Malay Text Documents

- **Techniques Compared:**
  - Inverted Index
  - B-Tree
  - B+ Tree
- **Dataset**:
  - **Malay Honey Corpus**: A custom dataset of 500 Malay text documents
  - Processed using tokenization, stopword removal, and stemming
  - 201 unique terms extracted from only two documents
- Evaluation Metrics:
  - Indexing time
  - Retrieval time
  - Retrieval effectiveness (precision, recall, F1-score)
  - Visualized using tables and line graphs

Rosnan et al.

3. Inverted Index and B-Tree Search on Malay Text Documents

- **Findings:**
  - Inverted Files: Faster at indexing
  - B+ Tree: Superior in retrieval speed and scalability
  - No stated limit on search depth or result count
- **Conclusion:**
  - B+ Tree provided better scalability and accuracy, despite Inverted Files indexing faster, making B+ Tree more suitable for structured keyword retrieval in small to medium datasets.

Terrovitis et al.

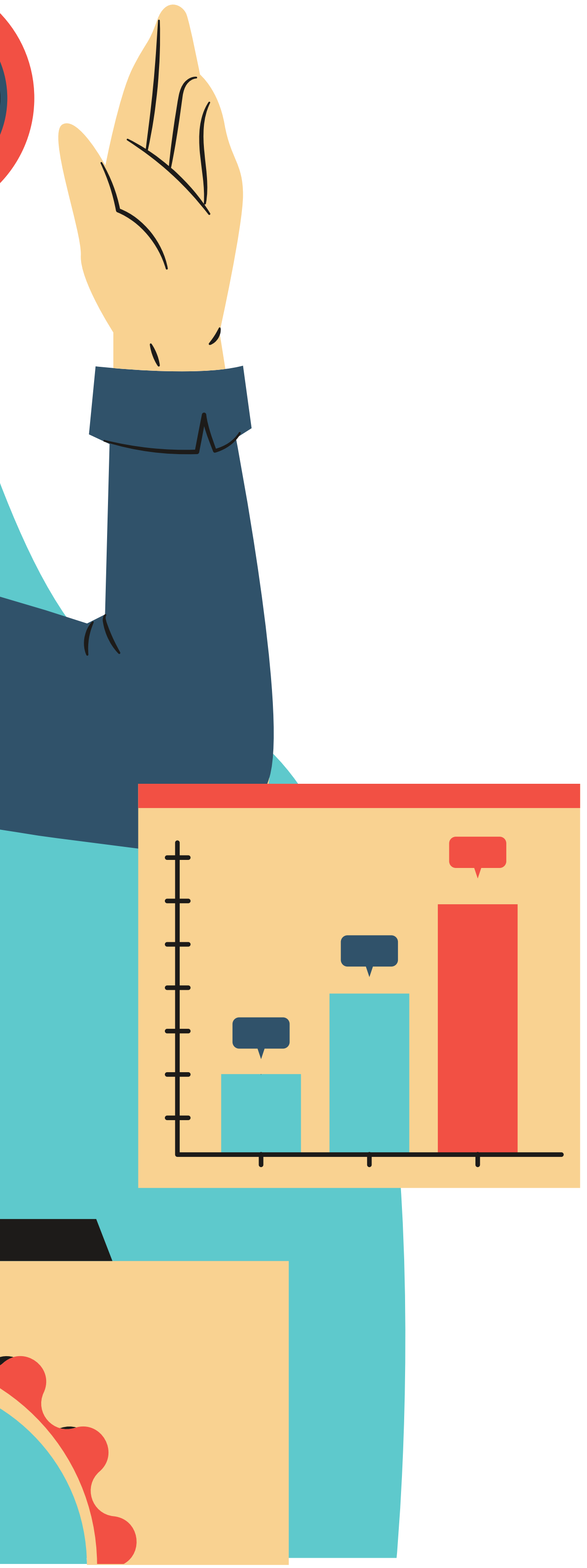
4. Hybrid Trie and Inverted Index for Set-Valued Data Search

- **Technique Introduced:**
  - HTI (Hybrid Trie-Inverted Index)
  - Combines Trie and Inverted File
  - Supports subset, superset, and equality queries
- **Dataset:**
  - **msweb**: 320,000 user sessions, 294 items (CSV)
  - **msnbc**: 1 million sessions with 17 items
  - **Synthetic data** using Zipfian distributions for scalability testing
- **Methodology:**
  - Most frequent items → stored in Trie (main memory)
  - Remaining items → stored in Inverted Index (secondary storage)
  - Reduces disk I/O by up to 90%
  - Keeps memory use under 0.5MB

Terrovitis et al.

4. Hybrid Trie and Inverted Index for Set-Valued Data Search

- **Visualization:**
  - Line graphs for scalability
  - Bar charts for query cost
  - Tables for memory footprint
- **Conclusion:** HTI offers efficient search over set-valued data, combining the advantages of Trie and Inverted Index with minimal memory overhead and high scalability.



03 Methodology

DATASET

- Amazon Reviews Dataset (Books category)
- Subset: Books_5-core.json
- Each book has ≥ 5 user-generated reviews
- JSON format with fields:
  - reviewText – Full body of the review (used in this project)
  - summary – Short description
  - overall – Rating
  - reviewerID, asin, unixReviewTime

- {"overall": 5.0, "verified": false, "reviewTime": "03 30, 2005", "reviewerID": "A1REUF3A1YCPHM", "asin": "0001713353", "style": {"Format:": " Hardcover"}, "reviewerName": "TW Ervin II", "reviewText": "The King, the Mice and the Cheese by Nancy Gurney is an excellent children's book. It is one that I well remember from my own childhood and purchased for my daughter who loves it.\n\nIt is about a king who has trouble with rude mice eating his cheese. He consults his wise men and they suggest cats to chase away the mice. The cats become a nuisance, so the wise men recommend the king bring in dogs to chase the cats away. The cycle goes on until the mice are finally brought back to chase away the elephants, brought in to chase away the lions that'd chased away the dogs.\n\nThe story ends in compromise and friendship between the mice and the king.The story also teaches cause and effect relationships.\n\nThe pictures that accompany the story are humorous and memorable. I was thrilled to discover that it is back in print. I *highly* recommend it for children ages 2 to 7.", "summary": "A story children will love and learn from", "unixReviewTime": 1112140800}
- {"overall": 5.0, "verified": true, "reviewTime": "06 20, 2016", "reviewerID": "AVP0HXC9FG790", "asin": "0001713353", "reviewerName": "Amazon Customer", "reviewText": "The kids loved it!", "summary": "Five Stars", "unixReviewTime": 1466380800}
- {"overall": 5.0, "verified": true, "reviewTime": "01 24, 2016", "reviewerID": "A324TTUBKTN73A", "asin": "0001713353", "style": {"Format:": " Paperback"}, "reviewerName": "Tekla Borner", "reviewText": "My students (3 & 4 year olds) loved this book! Definitely recommend it to other teachers.", "summary": "Five Stars", "unixReviewTime": 1453593600}
- {"overall": 5.0, "verified": false, "reviewTime": "07 9, 2015", "reviewerID": "A2RE7WG349NV5D", "asin": "0001713353", "style": {"Format:": " Paperback"}, "reviewerName": "Deborah K Woroniecki", "reviewText": "LOVE IT", "summary": "Five Stars", "unixReviewTime": 1436400000}
- {"overall": 5.0, "verified": true, "reviewTime": "01 18, 2015", "reviewerID": "A32B7QIUDQCD0E", "asin": "0001713353", "reviewerName": "E", "reviewText": "Great!", "summary": "Five Stars", "unixReviewTime": 1421539200}

ALGORITHM DESIGN

1

Linear Search

- **Time Complexity**: **O(N)**
- **Space Complexity**: **O(1)**

2

Inverted Index Search

- **Time Complexity**: **O(1)** best case, **O(log N)** worst case
- **Space Complexity**: **O(N + T)**, T = number of unique tokens

3

Trie Search

- **Time Complexity**: **O(M)**, M = length of the query
- **Space Complexity**: **O(T × L)**, T = number of tokens, L = average token length

4

B-Tree Search

- **Time Complexity**: **O(log N)**
- **Space Complexity**: **O(N)**
- Uses a balanced tree structure ideal for sorted and range-based lookups.

DATA STRUCTURE :

1.LINEAR SEARCH

- **Search Method :** Scans each review sequentially and compares content using a matching function.
- **Data Structure Used :** List (stores only matching results).
- **Characteristics :**
  - No indexing or preprocessing
  - Loads one review into memory at a time
  - Appends matched results to the list
  - **Low memory usage** (streaming)
  - Simple and **effective for small datasets**


```
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)
```

DATA STRUCTURE :

2.INVERTED INDEX SEARCH

- **Search Method :** Constructs an index mapping each unique normalized word to a list of review texts in which it appears. Enables faster keyword-based retrieval.
- **Data Structure :** defaultdict(list) for mapping tokens to documents.
- **Characteristics :**
  - Index is built during initialization through normalization and tokenization.
  - Enables **fast candidate lookups** for **exact**, **starts_with**, and **contains** queries.
  - Final filtering step ensures result consistency using is_match().
  - **Space-efficient for moderate datasets** but scales with vocabulary size.

```
self.inverted_index = defaultdict(list)
with open(dataset_path, "r") as f:
  for i, line in enumerate(f):
    if i >= limit:
      break
    review = json.loads(line)
    text = review.get("reviewText", "").strip()
    for word in set(normalize(text).split()):
      self.inverted_index[word].append(text)
```

DATA STRUCTURE :

3.TRIE SEARCH

- **Search Method :** Builds a trie (prefix tree) from review texts, enabling fast prefix-based lookups.
- **Data Structure :** Trie with TrieNode objects storing characters and associated sentences.
- **Characteristics :**
  - **Efficient** for **starts-with queries** by following character paths.
  - Stores full review texts at each relevant prefix node (node.sentences.append(...)).
  - Falls back to checking all texts with is_match() for exact/contains queries.
  - Slightly **higher memory usage** due to storing sentences at multiple nodes.
  - Query time improves significantly with shared prefixes in data.

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

class Trie:
    def insert(self, sentence):
        node = self.root
        for char in sentence.lower():
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            node.sentences.append(sentence)
```

DATA STRUCTURE :

4. B-TREE SEARCH

- **Search Method :** Builds a B-Tree mapping each normalized token → list of matching review sentences. Performs recursive tree traversal to find matches.
- **Data Structure :** Custom BTreeNode and BTree classes, storing: 1. key: token 2. value: list of matching review texts
- **Characteristics :**
  - Suitable for sorted and range-based queries.
  - **Efficient lookups** due to balanced tree structure (O(log N) insertion and search).
  - Stores tokens only once, but values grow with match frequency.
  - Uses is_match() on keys for flexible query types (e.g., contains, starts_with).
  - Better performance in sparse data than Trie in some scenarios.

```
# Insert tokens into B-Tree
for word in set(normalize(sentence).split()):
    self.tree.insert(word, sentence)

# BTreeNode logic
def insert_non_full(self, key, value):
    if key in self.keys:

self.values[self.keys.index(key)].append(value)
    else:
        self.keys.append(key)
        self.values.append([value])
```

QUERY TYPES

To ensure a comprehensive evaluation of the search algorithms, the system supports three distinct query types, each aligned with common patterns in real-world information retrieval:

1

Exact Match

- Finds entries that exactly match the query string (case-insensitive, token-based).
- Best for structured keyword search (e.g., dictionary, tags).
- Most efficient due to simple lookup logic and index compatibility.

2

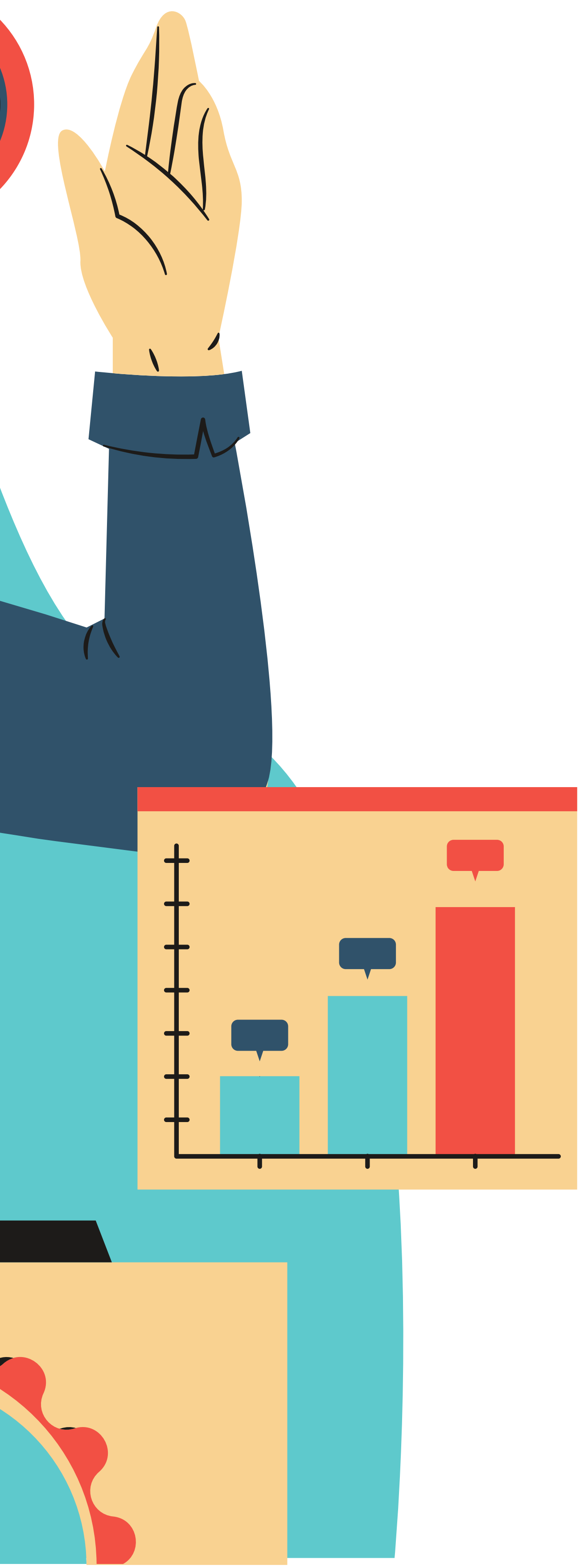
Starts With

- Matches entries where a token starts with the query string.
- Useful for prefix-based features like autocomplete or suggestions.
- Trie Search optimized for this; others incur more overhead.

3

Contains

- Retrieves entries with the query as a substring, anywhere in the text.
- Offers flexibility for partial or embedded matches.
- Highest computational cost due to full-text scanning.



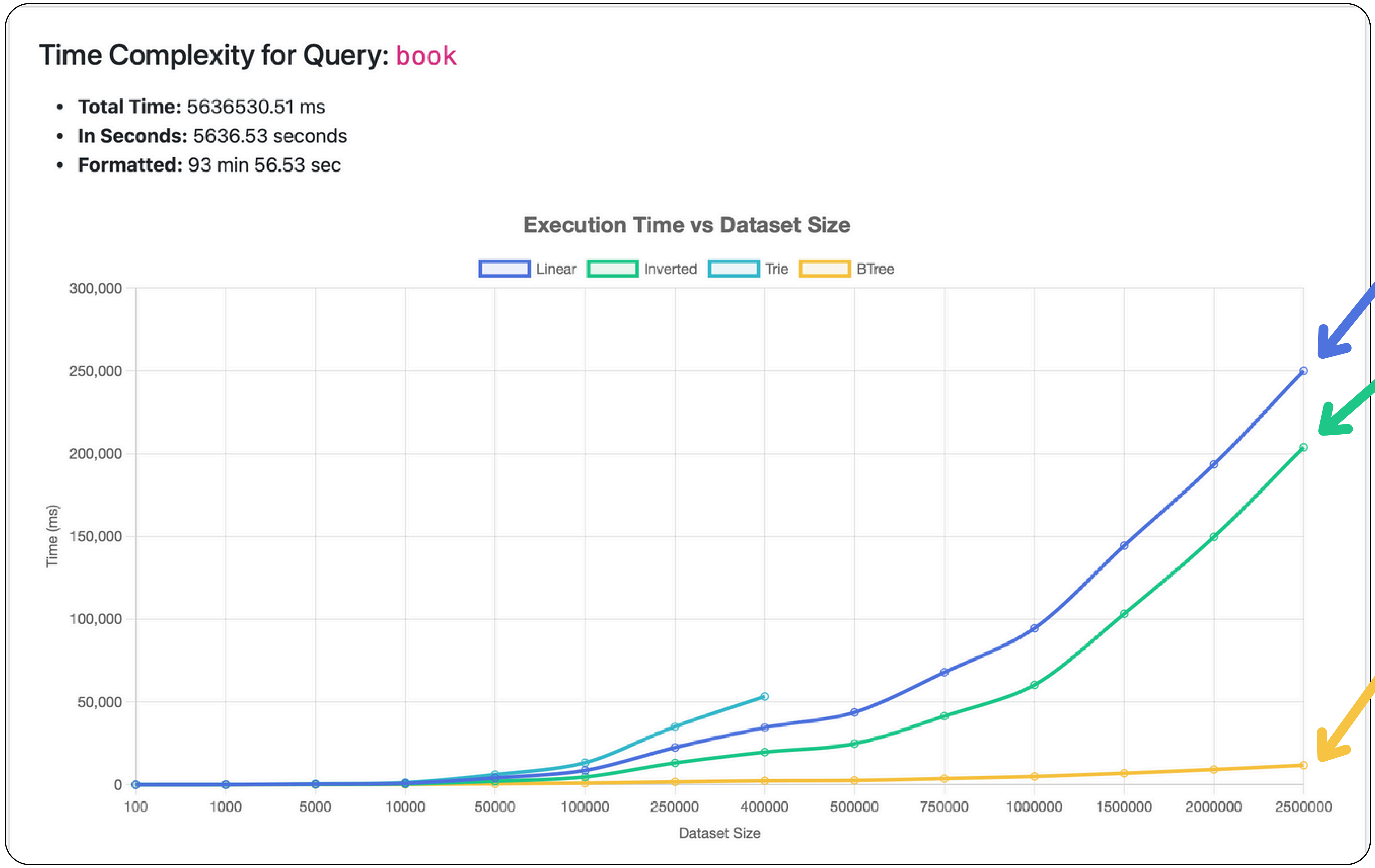
04 Results and Discussion

Theoretical Time Complexities

Search Method	Time Complexity	Notes
Linear	O(N)	Sequential scan, poor for large data
Inverted Index	O(1) avg, O(log N) worst	Fast lookup, may degrade with collisions
Trie	O(M), M = query length	Good for prefix queries, memory-heavy
B-Tree	O(log N)	Efficient and scalable for ordered data

TIME COMPLEXITY

1. QUERY TYPE : EXACT



Observations

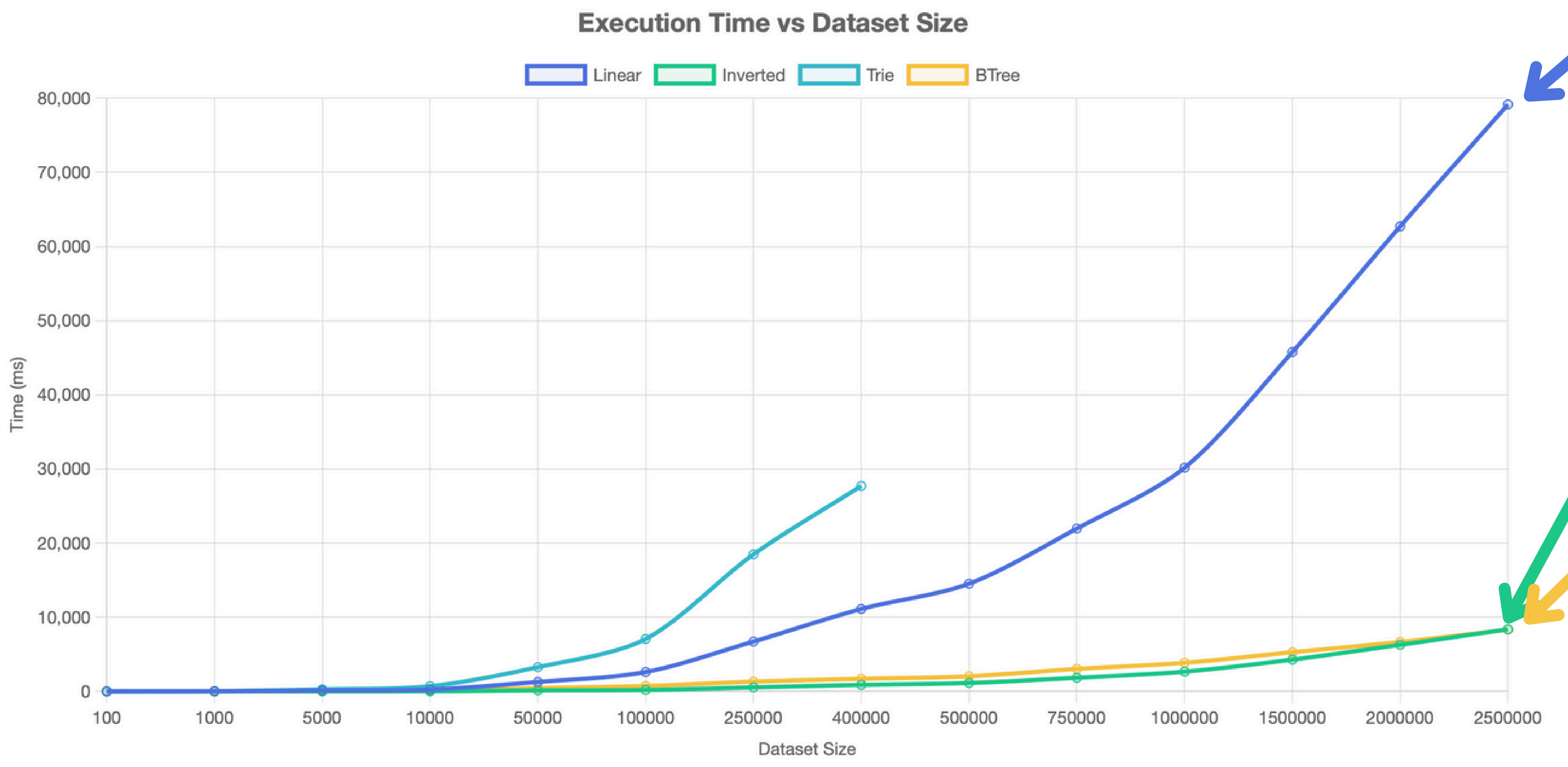
1. **Linear Search**:
    a. Linear growth O(n), worst beyond 100k
2. **Inverted Index Search**:
    a. Fast for medium-size (≤100k), slows after due to collisions
3. **B-Tree Search**:
    a. Best performance, stable at 2.5M entries
4. **Trie Search**:
    a. Excluded >400k entries due to memory; good at small scale

TIME COMPLEXITY

2. QUERY TYPE : STARTS_WITH

Time Complexity for Query: book

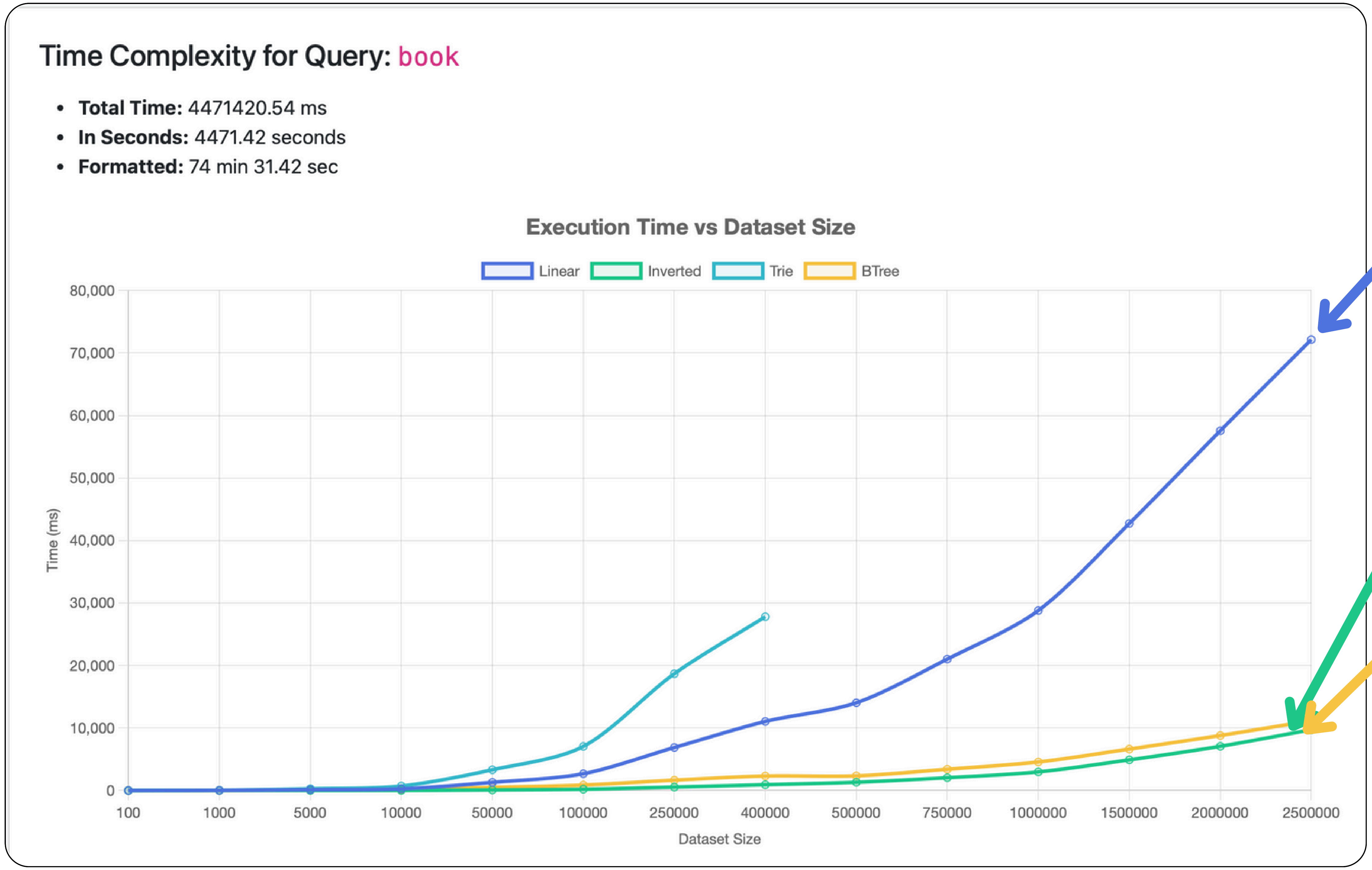
- **Total Time:** 4388913.72 ms
- **In Seconds:** 4388.91 seconds
- **Formatted:** 73 min 8.91 sec



Observations

1. **Linear Search**:
   a. O(n), efficient at small scale; outperformed Trie at mid-range
2. **Inverted Index Search**:
   a. Poor for prefix match, execution time rose quickly at scale
3. **B-Tree Search**:
   a. O(log n), best performance at all sizes, especially >100K
4. **Trie Search**:
   a. Excluded >400K records due to memory issues

3. QUERY TYPE : CONTAINS



Observations

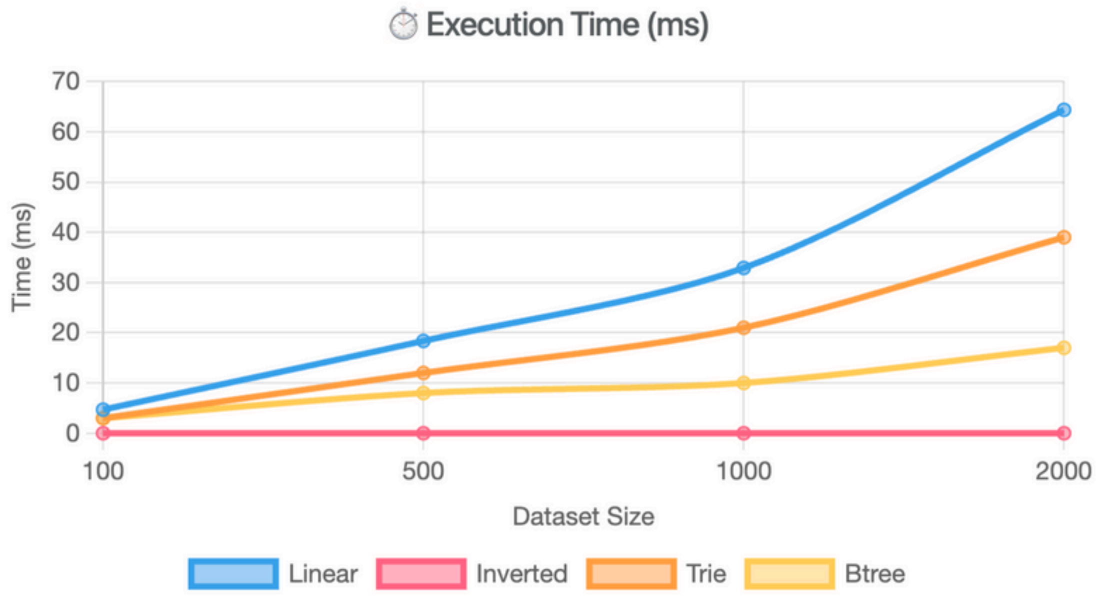
1. **Linear Search**:
   a. O(n); consistent but very slow at large scale (>75,000 ms at 2.5M)
2. **Inverted Index Search**:
   a. Performed worse than Linear; not suited for substring search
3. **B-Tree Search**:
   a. Best overall performance; scalable even if not built for substring matching
4. **Trie Search**:
   a. Excluded >400K records due to memory issues

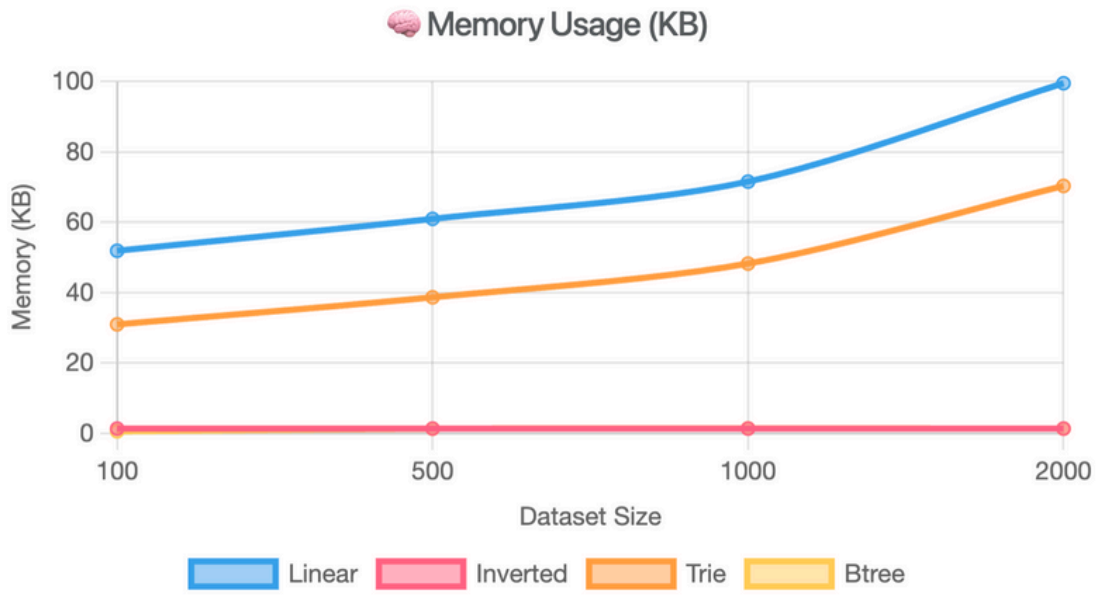
Theoretical Space Complexities

Search Method	Time Complexity	Notes
Linear	O(1)	No extra index needed, minimal memory use
Inverted Index	O(N + T)	Stores token-to-document mappings
Trie	O(T × L)	L = average prefix length; grows fast with vocabulary
B-Tree	O(N)	Stores keys + pointers in balanced tree structure

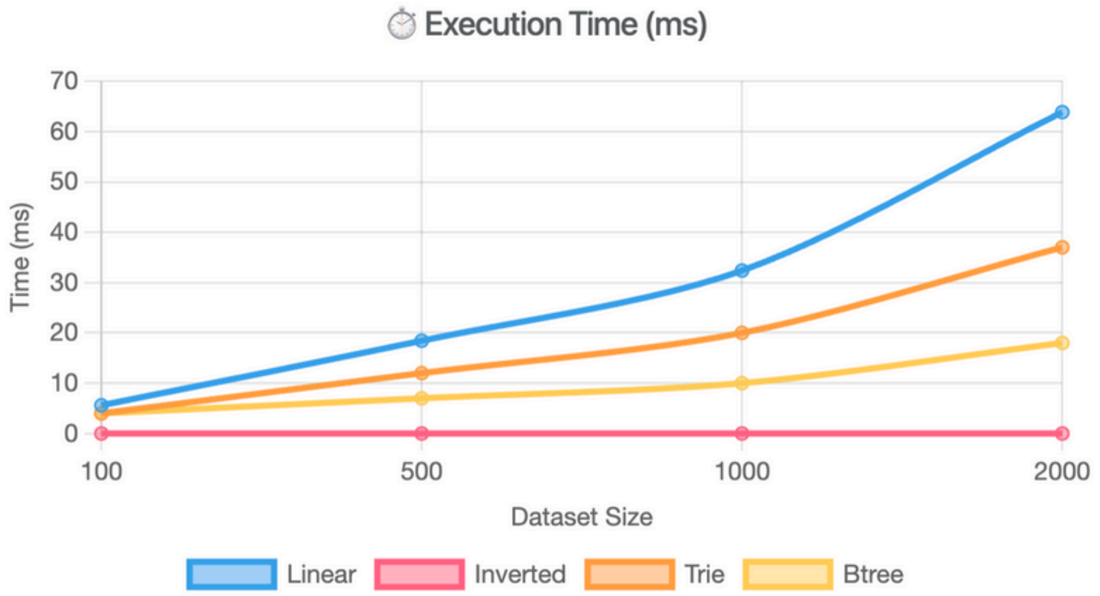
SPACE COMPLEXITY

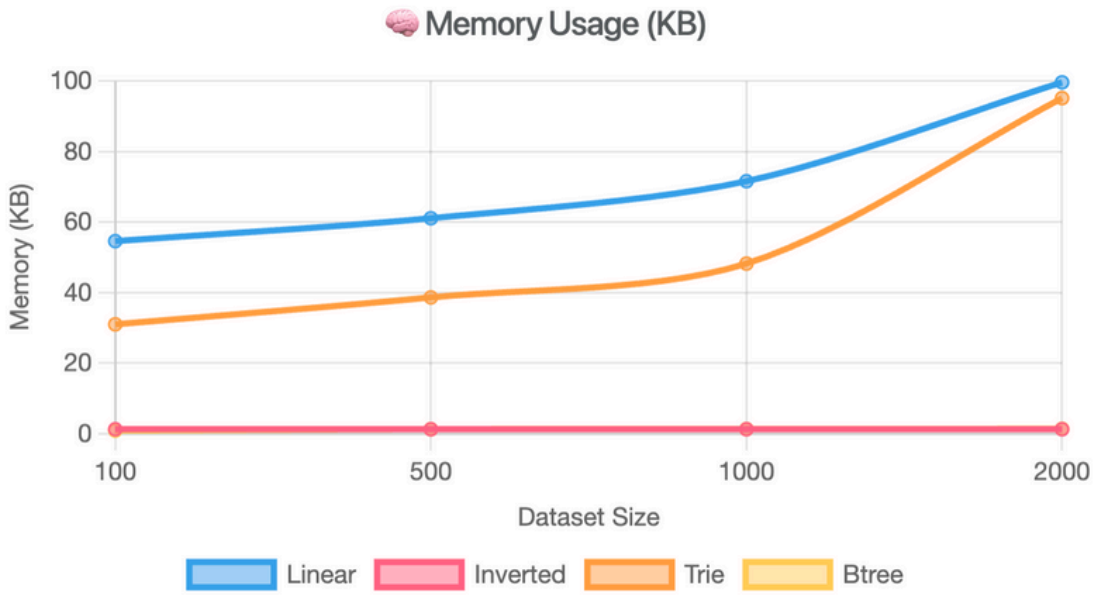
Query: `loved this book`





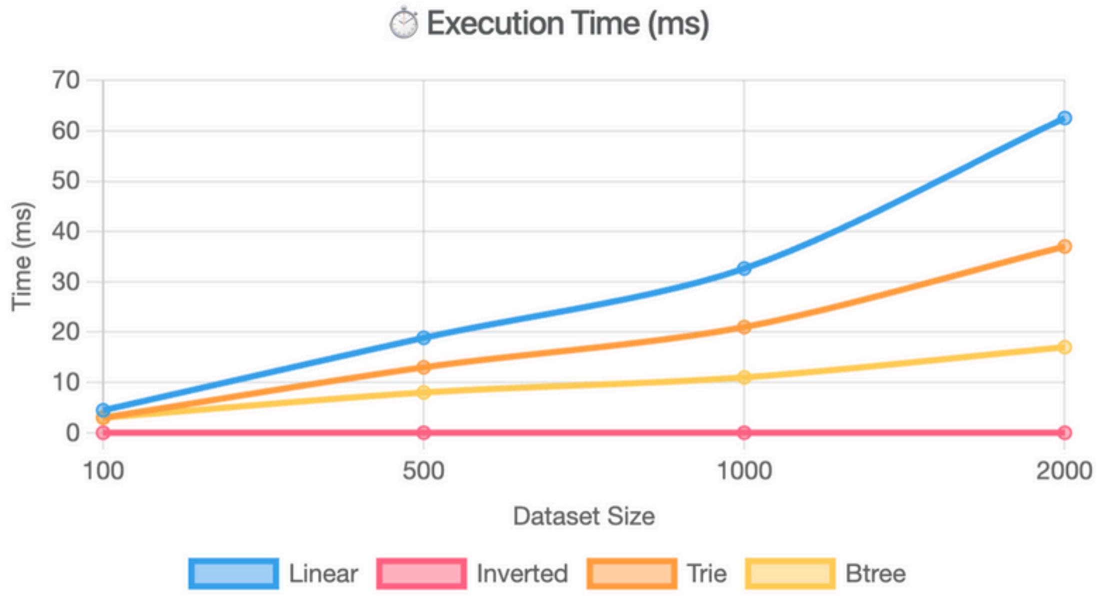
Query: `funny poems`

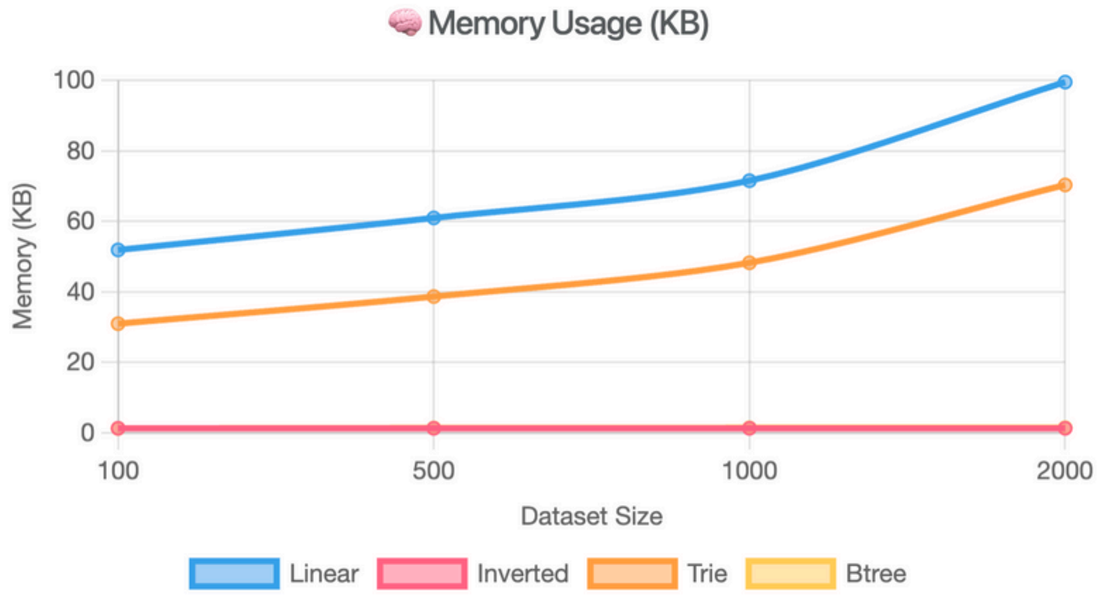




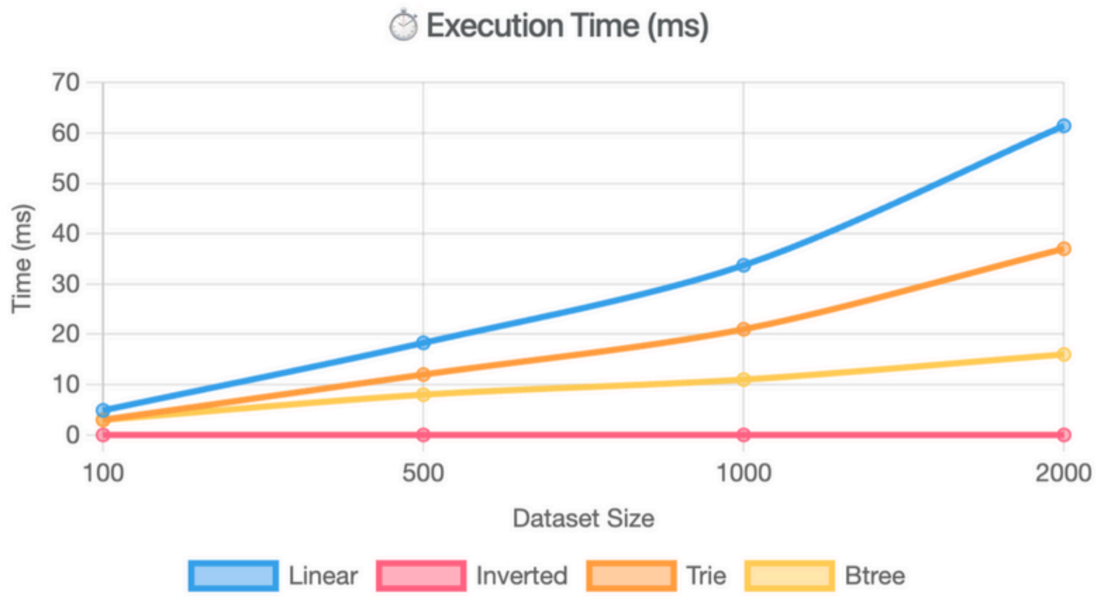
SPACE COMPLEXITY

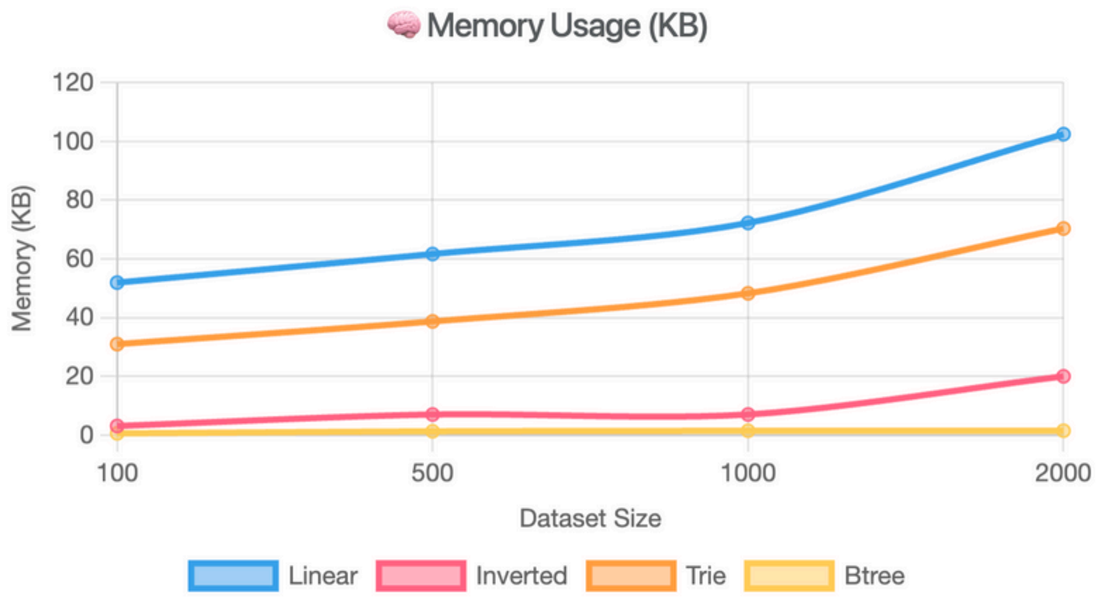
Query: `interesting premise`





Query: `boring`



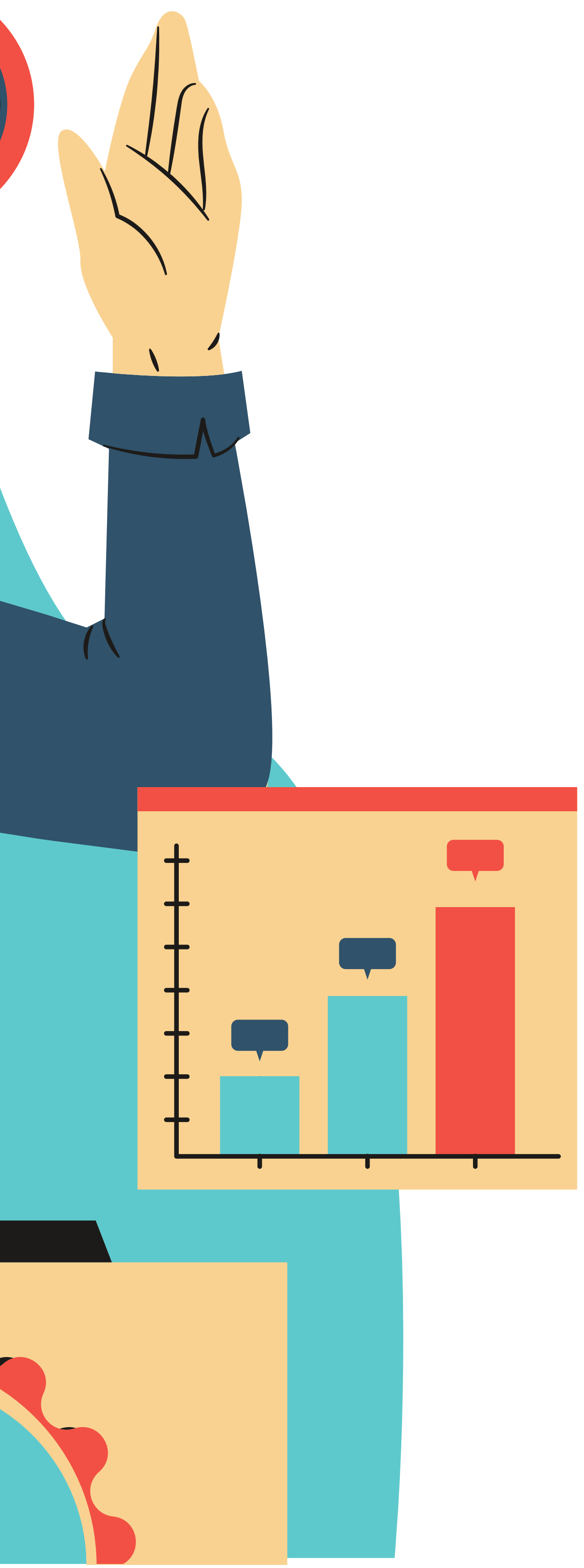


SPACE COMPLEXITY : OBERVATIONS

Observations

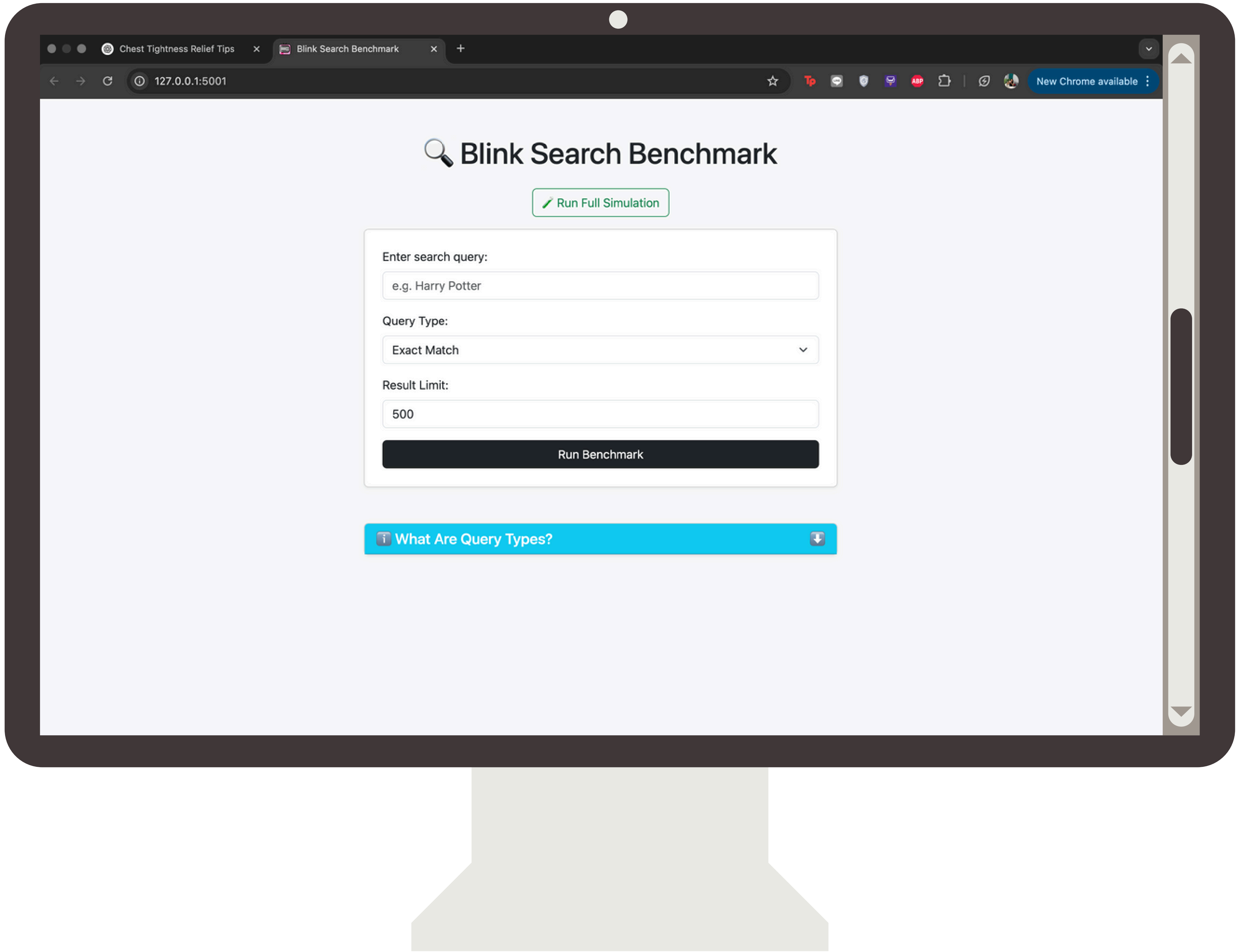
1. **Linear Search**:
    a. Lowest memory use; consistent flat footprint; ideal for low-RAM environments
2. **Inverted Index Search**:
    a. Very low space usage; scales well with unique token count
3. **B-Tree Search**:
    a. Moderate, predictable memory growth; suitable for large datasets
4. **Trie Search**:
    a. Very high memory use; even small datasets showed sharp memory spikes; impractical at scale

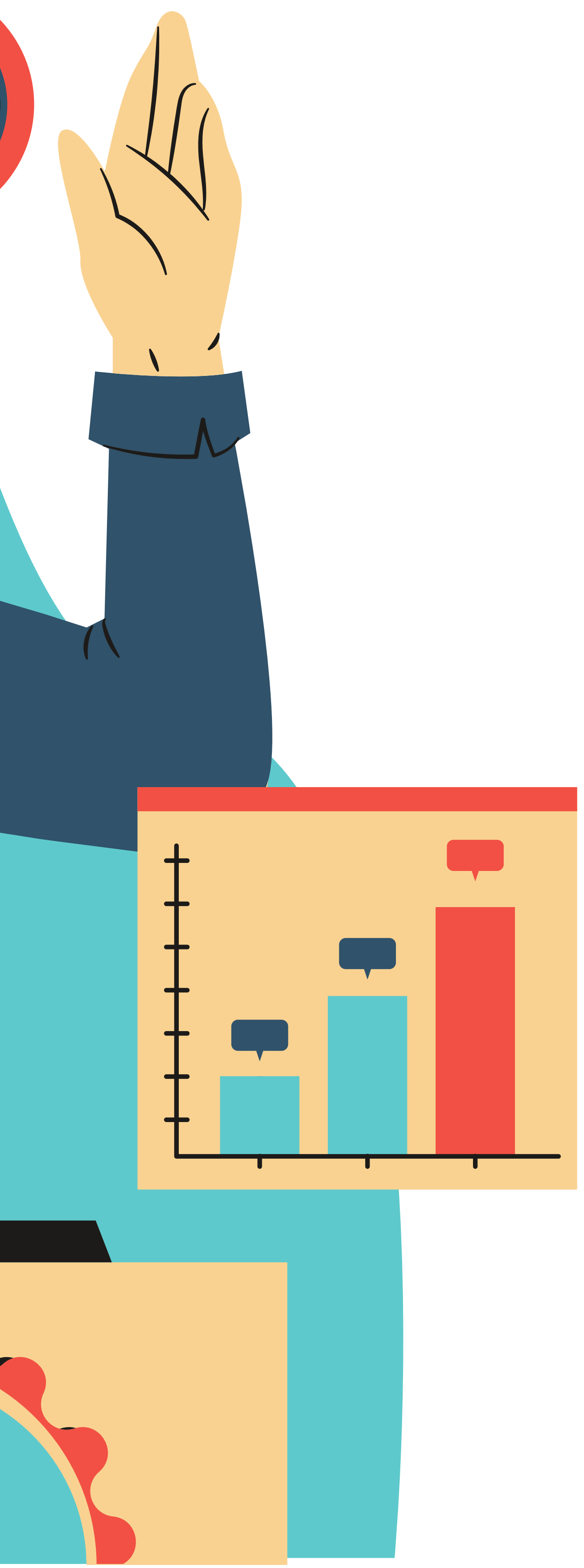
**** Note: Linear Search appeared to consume more memory than expected due to implementation-level memory accumulation. Theoretically, it uses constant auxiliary space (O(1)), while Trie remains the most memory-intensive structure.****



05 Demo

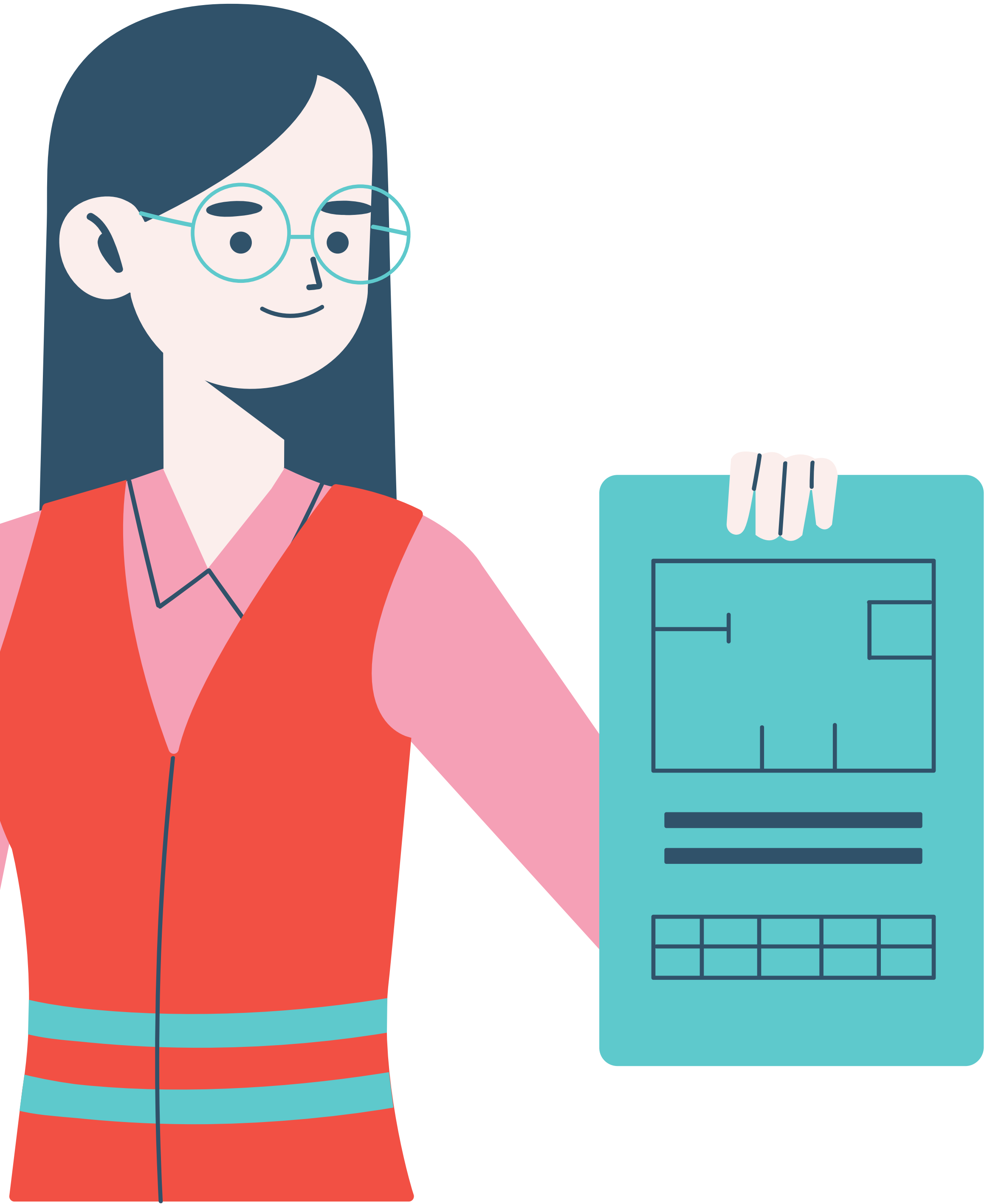
DEMO


Chest Tightness Relief Tips
Blink Search Benchmark
127.0.0.1:5001
New Chrome available
Blink Search Benchmark
Run Full Simulation
Enter search query:
e.g. Harry Potter
Query Type:
Exact Match
Result Limit:
500
Run Benchmark
What Are Query Types?



06 Conclusion

CONCLUSION



We benchmarked four search algorithms—Linear, Inverted Index, Trie, and B-Tree—on Amazon book review data across three query types: exact, starts with, and contains.

B-Tree emerged as the **most balanced algorithm**, offering consistently fast performance and scalability with reasonable memory usage.

KEY FINDINGS

1

Linear Search
While memory-efficient and simple, proved too slow at large scales

2

Inverted Index Search
excelled in exact match tasks on moderate datasets but struggled with prefix and substring queries

3

Trie Search
Consumed excessive memory and became impractical for larger datasets

4

B-Tree Search
Consistently fast performance and scalability with reasonable memory usage

FUTURE WORK

- Optimize Trie memory with Radix Trees and compression
- Enable incremental loading and persistent indexing
- Add advanced queries like fuzzy or semantic matching
- Test on broader datasets beyond Amazon reviews

CHALLENGES AND LIMITATIONS

1. High Memory Consumption (Trie)
    - Trie nodes store characters and full review texts → sharp memory growth.
    - At 2.5M records, usage exceeded 90 GB, crashing the process.
    - Each input size required a full rebuild, amplifying memory overhead.
2. Lack of Caching / Reuse
    - No incremental reuse between input sizes.
    - Structures rebuilt from scratch per dataset → repeated memory allocation.
    - This design preserved fairness but introduced performance penalties.
3. Hardware Constraints
    - Benchmarks ran on a 96 GB RAM machine.
    - Trie could not scale beyond 400,000 entries without crashing.
    - Larger inputs were skipped to maintain system stability and result accuracy.

IMPACT & APPLICATIONS

1

- **Academic Use:**
  - Supports algorithm education and benchmarking in computer science courses.

2

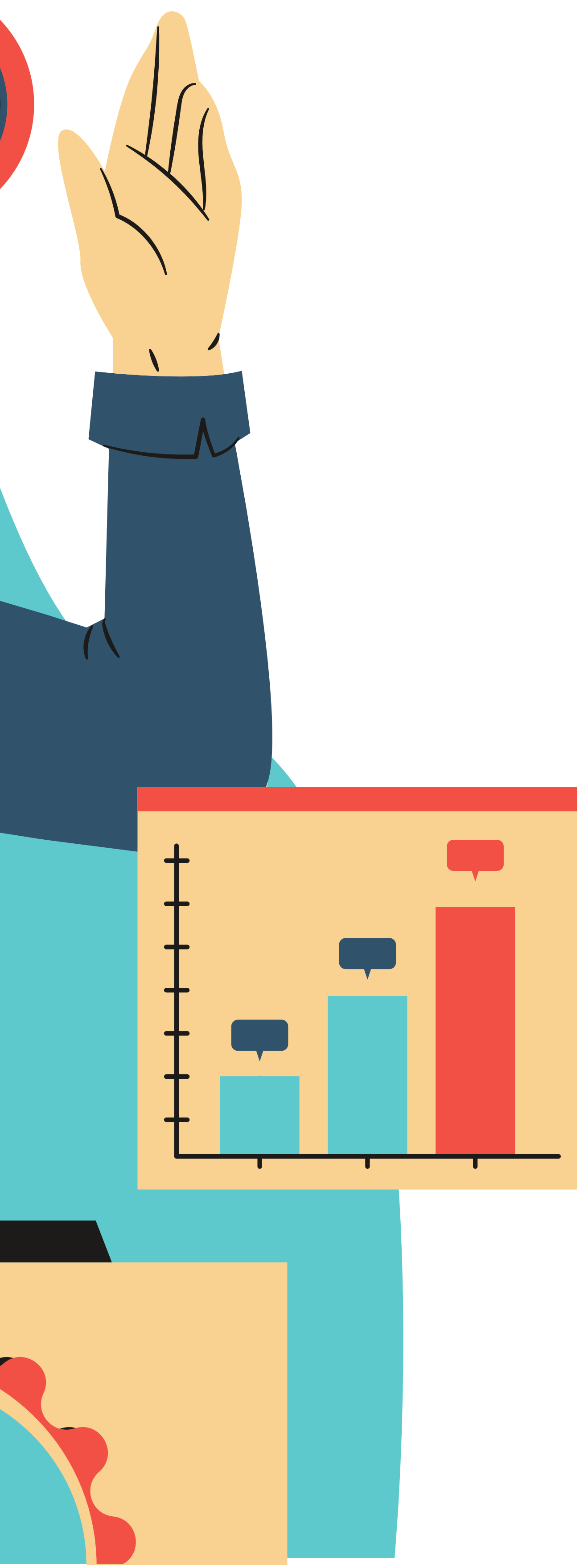
- **Real-World Relevance:**
  - Applicable to search engines, document indexing systems, and NLP pipelines.

3

- **Scalability Insight:**
  - Helps developers choose the best algorithm for datasets of different sizes and query types.

4

- **Research Potential:**
  - Can be extended to benchmark hybrid search models or optimize memory usage.



THANK YOU

Q & A