# Project Phase 1 Deliverable 1: Data Structure Design and Implementation

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 3, 2024

**Phase 1: Data Structure Design and Implementation – Search Engine (QuickFind)**

**Deliverable 1: Search Engine Optimization Report**

1. **Define the Application Context :**

Application: QuickFind – A High-Efficiency Search Engine

Context: QuickFind is made to do extensive, real-time searches across a vast collection of web pages, producing precise and quick search results. The objective is to create a search engine that can index, search, and retrieve data from a large corpus of the internet while managing high query volumes using effective data structures.

1. **Key Data Structures :**

**The Inverted Index:**

Goal: QuickFind's primary data structure, the inverted index, associates each term with a list of web pages on which it can be found. This enables QuickFind to find and return pertinent pages for any search query in a timely manner.

Execution:

* The main structure used by QuickFind is a hash map, in which each keyword is kept as a key and its corresponding value is a posting list with document IDs for web pages that include the phrase.
* Additional metadata, such as the keyword's frequency and position in each document, may be included with each posting list entry.

Methods of Optimization:

* Compression: QuickFind uses delta encoding on the document IDs to efficiently manage memory consumption. This minimizes the memory footprint by keeping only the differences between successive document IDs.
* Partitioning and Sharding: QuickFind divides the inverted index among several servers for scalability. Large datasets require horizontal scaling, which is supported by this sharding technique and helps balance search loads.

**Trie-Based Autocomplete :**

Goal: QuickFind's autocomplete, and spelling correction functions rely on a Trie data structure. With real-time query completion, this improves the user experience and makes quick, prefix-based search suggestions possible.

Execution:

* A path from the root to a node in the Trie spells out a whole keyword, and each node in the Trie represents a character.
* By storing more data, such the frequency of search phrases, the Trie enables QuickFind to rank suggestions that are popular or pertinent based on previous user queries.

Methods of Optimization:

* Radix Tree Compression: QuickFind turns the Trie into a Radix Tree by combining nodes with only one child into a single node in order to save memory. Memory use is reduced without compromising search speed thanks to this compression.
* Hot Query Caching: QuickFind lessens the strain on the Trie structure by caching frequently searched keywords to offer immediate recommendations for well-liked searches.

**Priority Queue for Relevance Ranking :**

The goal of QuickFind is to make sure that the most relevant pages show up first in search results by managing relevance scores using a priority queue.

Execution:

* QuickFind stores document IDs and the corresponding relevance ratings in a max-heap. Prioritized and displayed at the top of the search results are the documents with the highest scores.
* A number of variables, including document authority, freshness, and keyword frequency, affect the relevance score.

Methods of Optimization:

* Lazy Scoring: QuickFind concentrates processing power on the top results by delaying scoring for documents lower in the priority queue. This eliminates needless computations, which speeds up query processing.
* Score pruning reduces the amount of data analyzed and speeds up QuickFind's findings by eliminating documents with relevance scores below a predetermined threshold.

**Hash Tables for Metadata Retrieval :**

Goal: QuickFind employs hash tables to rapidly retrieve metadata (such as URLs, titles, and summaries) for every document. Presenting snippets of search results and other document details requires this.

Execution:

* Each document is given a unique ID by QuickFind, which then utilises this ID as the key in a hash table that contains metadata objects as values.
* While secondary material is preserved on disc to conserve memory, essential metadata, including titles and URLs, is kept independently in memory to enable quick access.

Methods of Optimisation:

* Chained Hashing: QuickFind ensures constant-time access even as the hash table expands by effectively handling collisions through the use of chained hashing.
* Metadata layering optimises memory consumption and speed by keeping less frequently accessible data on disc and storing essential information in memory for instant retrieval.

1. **Additional Considerations :**

Scalability

* Distributed Indexing: To enable concurrent query processing, QuickFind's design divides the inverted index over several servers. Sharding, which helps manage big datasets and guarantees dependable performance under high loads, is used to accomplish this distribution.
* Parallel Processing: QuickFind speeds up response times and lessens single-server bottlenecks by processing distinct portions of a query concurrently across multiple shards and machines.

Memory-Speed Equilibrium:

* Multi-Layer Caching: QuickFind has a caching technique with two layers. While an SSD-based secondary cache manages less frequently accessed data, high-priority data is stored in memory caches for quick access. Memory utilization and retrieval speed are balanced in this configuration.
* Dynamic Index Updates: QuickFind uses incremental indexing to keep search results current, allowing new documents to be indexed as they are added or changed instead of completely reindexing every

**Design the Data Structures:**

**Inverted Index :**

* Design: Each keyword in the inverted index is mapped to a list of document IDs that include it, acting as a key in a dictionary. Metadata like position and frequency are included in every list entry.
* Time Complexity: The length of each posting list determines the retrieval time, and search queries are O(1) for keyword lookup.
* Space Efficiency: Posting list partitioning among servers facilitates scalability, while delta encoding compresses document IDs to save memory by maintaining only differences between consecutive entries.
* Justification: Studies on massive search engines, such as Google's MapReduce model, confirm that distributed inverted indexes optimize memory and query speed. For quick indexing, Python's dictionaries offer effective key-value mapping.

**Trie-Based Autocomplete :**

* Design: QuickFind's Trie is a layered dictionary structure in which every character node forms a path for every keyword by linking to sub-characters. Prioritised autocomplete is made possible by additional frequency metadata.
* Time Complexity: Trie lookups and insertions are O(k), where k is the word length. The average lookup time is decreased by caching frequently asked queries.
* Space Efficiency: By combining nodes with single offspring, Radix Tree compression minimises the size of the Trie. For quick access, popular keywords are kept in memory.
* Rationale: Studies demonstrate that Trie-based autocompletes with radix compression improve prediction speed and decrease space complexity. For quick string manipulation, Python dictionaries effectively represent Trie nodes.

**Priority Queue for Relevance Ranking :**

* Design: To guarantee that the best results show up first, the priority queue, which is constructed as a max-heap using Python's heapq, maintains document IDs with relevance ratings.
* Time Complexity: Lazy evaluation reduces heap operations by scoring only the top documents, and insertions and deletions are O(logn).
* Space Efficiency: By keeping the heap small and concentrating on the highest-ranked documents, memory usage is decreased.
* Rationale: The use of max-heaps for real-time relevance sorting is supported by research on ranking algorithms, guaranteeing accurate and efficient query responses. Heap-based priority queues are directly supported by Python's heapq.

**Hash Table for Metadata Retrieval :**

* Design: Document IDs serve as keys for constant-time lookups in a hash table that contains metadata such as URLs and titles.
* Time Complexity: Chained hashing handles collisions, and lookups and insertions are typically O(1).
* Space Efficiency: To balance speed and memory requirements, important metadata (such URLs) is kept in memory while less-used data is kept on disc.
* Rationale: Hash tables provide fast access to metadata, which is essential for displaying results in real time. For quick retrieval, Python's dictionaries offer an efficient hash table structure.

Speed, memory efficiency, and scalability are given top priority in QuickFind's data structure design. To ensure great performance for large-scale search jobs, each structure makes use of Python's built-in tools and tried-and-true research approaches, like max-heaps for ranking, distributed indexing, and Trie compression.

**Implement the Data Structures in Python:**

**Code:**

class InvertedIndex:

def \_\_init\_\_(self):

"""Initialize the inverted index as a dictionary."""

self.index = {}

def add\_document(self, doc\_id, text):

"""

Add a document to the inverted index.

:param doc\_id: Unique identifier for the document.

:param text: Text content of the document.

"""

for word in text.split():

word = word.lower() # Normalize to lower case

if word not in self.index:

self.index[word] = []

self.index[word].append(doc\_id)

def search(self, keyword):

"""

Search for documents containing the keyword.

:param keyword: The keyword to search for.

:return: List of document IDs containing the keyword.

"""

return self.index.get(keyword.lower(), [])

class TrieNode:

def \_\_init\_\_(self):

"""Initialize a Trie node with children and frequency."""

self.children = {}

self.is\_end\_of\_word = False

self.frequency = 0

class AutocompleteTrie:

def \_\_init\_\_(self):

"""Initialize the Trie for autocomplete."""

self.root = TrieNode()

def insert(self, word):

"""

Insert a word into the Trie.

:param word: The word to insert.

"""

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True

node.frequency += 1 # Increase frequency count for autocomplete relevance

def autocomplete(self, prefix):

"""

Retrieve autocomplete suggestions for a given prefix.

:param prefix: The prefix to search for.

:return: List of words that match the prefix.

"""

node = self.root

for char in prefix:

if char not in node.children:

return []

node = node.children[char]

suggestions = []

self.\_find\_words(node, prefix, suggestions)

return suggestions

def \_find\_words(self, node, prefix, suggestions):

"""Helper method to find words in the Trie."""

if node.is\_end\_of\_word:

suggestions.append(prefix)

for char, child\_node in node.children.items():

self.\_find\_words(child\_node, prefix + char, suggestions)

import heapq

class PriorityQueue:

def \_\_init\_\_(self):

"""Initialize the priority queue."""

self.elements = []

def push(self, item, priority):

"""

Add an item to the priority queue with a given priority.

:param item: The item to add.

:param priority: The priority of the item.

"""

heapq.heappush(self.elements, (-priority, item)) # Use negative priority for max-heap

def pop(self):

"""

Remove and return the highest priority item.

:return: The item with the highest priority.

"""

return heapq.heappop(self.elements)[1]

def is\_empty(self):

"""Check if the priority queue is empty."""

return len(self.elements) == 0

class MetadataHashTable:

def \_\_init\_\_(self):

"""Initialize the metadata hash table."""

self.table = {}

def add\_metadata(self, doc\_id, metadata):

"""

Add metadata for a document.

:param doc\_id: Unique identifier for the document.

:param metadata: Metadata associated with the document.

"""

self.table[doc\_id] = metadata

def get\_metadata(self, doc\_id):

"""

Retrieve metadata for a document.

:param doc\_id: Unique identifier for the document.

:return: Metadata associated with the document.

"""

return self.table.get(doc\_id)

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

# Create and populate InvertedIndex

inverted\_index = InvertedIndex()

inverted\_index.add\_document(1, "Python is great for data structures")

inverted\_index.add\_document(2, "Data structures are essential for algorithms")

print("Inverted Index Search for 'data':", inverted\_index.search("data"))

# Create and populate AutocompleteTrie

autocomplete\_trie = AutocompleteTrie()

words = ["python", "pythonic", "java", "javascript", "jazz", "jargon"]

for word in words:

autocomplete\_trie.insert(word)

print("Autocomplete Suggestions for 'ja':", autocomplete\_trie.autocomplete("ja"))

# Create and use PriorityQueue

priority\_queue = PriorityQueue()

priority\_queue.push("doc1", 5)

priority\_queue.push("doc2", 10)

priority\_queue.push("doc3", 1)

print("Priority Queue Pop:", priority\_queue.pop())

print("Priority Queue Pop:", priority\_queue.pop())

# Create and use MetadataHashTable

metadata\_table = MetadataHashTable()

metadata\_table.add\_metadata(1, {"title": "Python Basics", "url": "http://example.com/python"})

print("Metadata for doc 1:", metadata\_table.get\_metadata(1))

**Implementation Overview**

Index inverted:

* keeps track of keywords that map to document IDs.
* allows for keyword searching and document insertion.

Trie Autocomplete:

* stores character nodes in hierarchical dictionaries.
* allows for word insertion and prefix-based autocomplete suggestion retrieval.

Priority Line:

* Using heapq in Python to implement a max-heap.
* enables the addition of items with priorities and the retrieval of the item with the highest priority.

Hash Table for Metadata:

* stores the metadata of documents in a dictionary.
* based on document IDs, offers ways to add and retrieve metadata.

Top Techniques

* Every class has its own state and is encapsulated.
* Inputs, outputs, and functionality are explained in detail in the method documentation.
* utilizing Python's built-in libraries (such as heapq for the priority queue) to increase efficiency.
* Each data structure's functionality is illustrated with an example usage at the end.

**Output :**

A computer screen shot of a black screen

Description automatically generated

A computer screen with text on it

Description automatically generated

**Explanation of the Output:**

Inverted Index Search: Document IDs 1 and 2 include the keyword, according to the results of the search query "data." This demonstrates that the word "data" appears in both publications.

Recommended Autocomplete Items:

'java', 'JavaScript', 'jazz', and 'jargon' are the list of potential completions for the preset 'ja'. This illustrates how the autocomplete feature works depending on the terms that are inserted.

Pop of the Priority Queue: The first output from the priority queue shows that 'doc2', the item with the highest priority (10), is removed from the queue first.

'doc1' is popped next, with a priority of 5, according to the second output. By preserving the items' priority-based order, this shows that the priority queue is operating as intended.

Retrieve Metadata:

A title and a URL are among the related metadata for document ID 1 that are displayed in the metadata retrieval output. This confirms that the info was appropriately saved and retrieved.

This result demonstrates that every data structure that was developed functions as planned and can communicate with the QuickFind search engine framework.

**REFERENCE:**

Inverted Index:

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. The MIT Press.

This book offers a comprehensive introduction to information retrieval and discusses the inverted index extensively.

Tries and Autocomplete:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.

This textbook includes information on data structures like tries and their applications in string searching and autocomplete algorithms.

Priority Queues and Heaps:

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

This book covers a variety of algorithms and data structures, including the implementation and analysis of priority queues using heaps.

Hash Tables:

Knuth, D. E. (1997). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

This volume discusses hashing techniques and hash tables, offering a deep dive into their implementation and performance.

General Data Structures and Algorithms:

Weiss, M. A. (2013). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.

This book provides a solid foundation in data structures and algorithms, including their performance analysis.