A Graph-Based Indexing Technique for Efficient Searching in Large Scale Textual Documents

by

Mohamed Abdulla Kalandar Mohideen

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of
Master of Applied Science in Electrical and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
Canada

Abstract

This thesis proposes a new graph-based indexing technique to improve the search latency for textual documents by using a Graph-Based Index (GBI) structure. GBI uses a directed graph built using a hash table to effectively capture the simultaneous occurrence of multiple keywords in a document. The objective is to use the relationship between the search keywords captured in the graph structure and a fast hash table lookup to effectively retrieve all the results of a query at once. A proof-of-concept prototype has been built for both GBI and Inverted Index. A thorough performance analysis is carried out for comparing GBI with Inverted Index using a synthetic workload. GBI is also compared with an enterprise-level search engine called Elasticsearch. The results show that the graph-based indexing technique can reduce the search latency for executing queries notably in comparison to Inverted Index and Elasticsearch.

Acknowledgements

First and foremost, praises and thanks to God, the Almighty, for His mercy, throughout my research work to complete the research successfully. I would like to thank my parents and my family for supporting me throughout my education. Without their help, I could not have accomplished my work.

I would like to express my sincere and deepest gratitude to my enlightened supervisors Dr. Shikharesh Majumdar and Dr. Marc St-Hilaire for their continuous support, motivation, and feedback on this research work. I would like to thank the graduate assistant, Jennifer Poll and the department of Systems and Computer Engineering, Carleton University for their administrative support. I am grateful to TELUS and Natural Sciences and Engineering Research Council of Canada (NSERC) for providing financial aid. I would also like to show my appreciation to Compute Canada and Research Computing and Development Cloud (RCDC) of Carleton University for providing a computing resource for this research. Many thanks to all my friends, well-wishers, and amazing people I have met in different walks of life who helped me navigate through arduous situations. I am indebted to my school teacher Mr. Muthukumaraswamy for creating intrinsic motivation in me. A special thanks to Siraj and Ian for making me aware of Compute Canada and RCDC of Carleton University. A big shout-out to my friends Nirmal, Ridhuvarshan, and Pravish for various technical discussions. Solving critical technical problems would not have been possible without assistance from the online technical blog posts and help forums.

Table of Contents

List of Tables **viii**

List of Figures **ix**

List of Algorithms **xiv**

List of Symbols **xv**

List of Abbreviations **xvi**

1 Introduction **1**

1.1 Motivation 4

1.2 Research Objectives 6

1.3 Overview of the Proposed Approach 7

1.4 Contributions 8

1.5 Publication 9

1.6 Thesis Outline 9

2 Background and Related Work **11**

2.1 Inverted Index 11

2.1.1 General Concepts 11

2.1.2 Representative Set of Works on Inverted Index 13

2.2 Hash Table 20

2.2.1 General Concepts 20

2.2.2 Basic Hash Table Construction & Search Performance 21

2.2.3 Collision Resolution Strategies 23

2.2.4 Hash Table Load Factor 24

2.2.5 Hash Table Re-Sizing 25

2.2.6 Application of Hash Table in Data Indexing 26

2.3 Selecting Keywords from Documents for Indexing 27

2.3.1 Documents Pre-Processing 28

2.3.2 Keyword Assignment 29

2.3.3 Extraction Based on a Term Feature 30

2.3.4 Automatic Keyword Extraction Methods 32

2.4 Graph Theory 36

2.4.1 General Concepts 36

2.4.2 Applications of Graphs 39

2.5 Concluding Remarks 45

3 Graph-Based Indexing Technique **46**

3.1 Graph-Based Index Structure 46

3.2 Algorithm for Indexing Using GBI 48

3.3 GBI Example 50

3.4 Pruned Graph-Based Index to Process Only Boolean AND Queries . 55

3.5 Boolean Queries and Search Algorithms Using Graph-Based Index . . 59

3.5.1 Boolean AND Queries 59

3.5.2 Boolean NOT Queries 66

3.5.3 Boolean OR Queries 72

3.6 Execution of Other Queries Using GBI 77

3.6.1 Neighbours of a Keyword Queries 77

3.6.2 Exclusive Keyword Queries 81

4 Experiments and Performance Analysis **84**

4.1 Prototype Implementation 85

4.2 Experimental Setup 88

4.2.1 Workload Generation 89

4.2.2 Performance Metrics 98

4.3 Experiments for Boolean Queries 99

4.3.1 Boolean AND Queries 100

4.3.2 Boolean NOT Queries 106

4.3.3 Boolean OR Queries 114

4.4 Experiments for Neighbours of a Keyword Queries 122
4.4.1 Analysis of the Search Latency Using Random Workload . . . 122
4.4.2 Analysis of the Search Latency Using Workloads for Neighbours of a keyword Queries 123
4.5 Experiment for Exclusive Keyword Queries 128
4.5.1 Analysis of the Search Latency Using Random Workload . . . 128
4.6 Experiments for Analyzing the Indexing Performance in GBI, Pruned GBI, and Inverted Index 130
4.6.1 Workload for Analyzing the Indexing Performance 130
4.6.2 Analysis of the Indexing Performance 131
4.7 Performance Comparison between GBI and Elasticsearch 134
4.7.1 Elasticsearch Setup 134
4.7.2 Performance Metrics 136
4.7.3 Analysis of the Search Latency in Elasticsearch and GBI . . . 137
4.7.4 Comparison of the Indexing Performance between GBI and Elasticsearch 148

5 Conclusions and Future Work **151**

5.1 Synopsis of the Concepts Presented 151
5.2 Performance Insights 153
5.3 Limitations and Recommendations 156
5.4 Future Work 157

References **159**

List of Tables

4.1 Parameters for Random Workload 90

4.2 Parameters for Relationship-based Workload 92

4.3 Parameters for Neighbours of a Keyword Query Workload 96

4.4 Performance Metrics 99

4.5 Original and Compressed GBI File Size 150

List of Figures

2.1 Example of Inverted Index. 12

2.2 Basic hash table representation. 21

2.3 Selecting keywords for indexing. 28

2.4 Example of an undirected and directed graph. 37

2.5 Adjacency matrix and list representation for directed graph example given in Figure 2.4. 39

3.1 Venn diagram representing the keywords and the documents that they appear in. 51

3.2 GBI - graph representation. 52

3.3 GBI - hash table representation. 54

3.4 Pruned GBI - graph representation. 56

3.5 Pruned GBI - hash table representation. 56

4.1 Procedure for building prototypes using the workload data. 87

4.2 Procedure for executing search queries using the prototypes. 87

4.3 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean AND query. 101

4.4 Effect of number of search keywords (SK_{count}) on the search time (T_s) in GBI and Inverted Index when number of documents indexed (D_{count}) is kept at the constant value of 6M for a Boolean AND query. 102

4.5 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *No* relationship among search keywords for a Boolean AND query. 103

4.6 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Partial* relationship among search keywords for a Boolean AND query. 104

4.7 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords for a Boolean AND query. 105

4.8 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean AND query. 106

4.9 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean NOT query. 107

4.10 Effect of number of search keywords (SK_{count}) on the search time (T_s) in GBI and Inverted Index when number of documents indexed (D_{count}) is kept at the constant value of 6M for a Boolean NOT query. 109

4.11 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *No* relationship among search keywords for a Boolean NOT query. 111

4.12 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Partial* relationship among search keywords for a Boolean NOT query. 112

4.13 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords for a Boolean NOT query. 113

4.14 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean NOT query. 114

4.15 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean OR query. . 116

4.16 Effect of number of search keywords (SK_{count}) on the search time (T_s) in GBI and Inverted Index when number of documents indexed (D_{count}) is kept at the constant value of 6M for a Boolean OR query. 116

4.17 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *No* relationship among search keywords for a Boolean OR query. 118

4.18 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Partial* relationship among search keywords for a Boolean OR query. 120

4.19 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords for a Boolean OR query. 121

4.20 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean OR query. 121

4.21 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a neighbours of a keyword query. . . . 123

4.22 Evaluation of search time (T_s) in GBI and Inverted Index when P_n is varied from 20K to 100K, keeping KI_{count} and N_{count} fixed to 2K for a neighbours of a keyword query. 124

4.23 Evaluation of search time (T_s) in GBI and Inverted Index when N_{count} and KI_{count} is varied from 2K to 10K, keeping P_n fixed to 60K for a neighbours of a keyword query. 125

4.24 Evaluation of search time (T_s) in GBI and Inverted Index when KI_{count} is varied from 2K to 10K such that all the keywords occur after the search keyword in GBI, keeping P_n fixed to 60K and N_{count} to 2K for a neighbours of a keyword query. 126

4.25 Evaluation of search time (T_s) in GBI and Inverted Index when KI_{count} is varied from 2K to 10K such that all the keywords occur before the search keyword in GBI, keeping P_n fixed to 60K and N_{count} to 2K for a neighbours of a keyword query. 128

4.26 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a exclusive keyword query. 130

4.27 Effect of number of documents indexed (D_{count}) on the indexing time (T_i) for GBI, Pruned GBI, and Inverted Index. 132

4.28 Effect of number of documents indexed (D_{count}) on the indexing memory (M_i) for GBI, Pruned GBI, and Inverted Index. 133

4.29 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean AND query. 138

4.30 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Partial* relationship among search keywords for a Boolean AND query. 139

4.31 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean AND query. 140

4.32 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean NOT query. 142

4.33 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Partial* relationship among search keywords for a Boolean NOT query. 143

4.34 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean NOT query. 144

4.35 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean OR query. 145

4.36 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Partial* relationship among search keywords for a Boolean OR query. 146

4.37 Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean OR query. 147

4.38 Effect of number of documents indexed (D_{count}) on the indexing time (T_i) for GBI and Elasticsearch. 149

4.39 Effect of number of documents indexed (D_{count}) on the indexing memory (M_i) for compressed GBI and Elasticsearch. 150

List of Algorithms

1 Construction of Graph-Based Index 49

2 Construction of Pruned Graph-Based Index 58

3 Executing a Boolean AND query search using Graph-Based Index . . . 61

4 Small-Vs-Small intersection algorithm 63

5 Executing a Boolean NOT query search using Graph-Based Index . . . 68

6 Executing a Boolean OR query search using Graph-Based Index 74

7 Executing a neighbours of a keyword query using Graph-Based Index . 80

8 Executing an exclusive keyword query using Graph-Based Index 83

List of Symbols

Symbol	Description
$\boldsymbol{D}$	Document that is currently being examined for match for a search query.
$\boldsymbol{D_{count}}$	Number of documents indexed.
$\boldsymbol{E}$	Set of search keywords specified after the NOT operator in Boolean NOT query.
$\boldsymbol{G}$	Hash table representing Graph-Based Index.
$\boldsymbol{Id}$	Identifier of the document.
$\boldsymbol{K_s}$	Search keyword for exclusive keyword query and neighbours of a keyword query.
$\boldsymbol{KI}$	Set of all the keywords found in the index except the search keyword $\boldsymbol{K_s}$.
$\boldsymbol{KI_{count}}$	Total number of keywords found in the index except the search keyword.
$\boldsymbol{K_i}$	One of the keywords from the set $\boldsymbol{KI}$ that is currently being examined for match.
$\boldsymbol{K_{pool}}$	Keyword pool containing an array of unique names.
$\boldsymbol{K_{poolsize}}$	Size of keyword pool.
$\boldsymbol{M_i}$	Average memory consumed for indexing $\boldsymbol{D_{count}}$ documents.
$\boldsymbol{N_{count}}$	Neighbour count of the search keyword.
$\boldsymbol{P}$	Set of postings lists.
$\boldsymbol{PL}$	Postings list.
$\boldsymbol{P_n}$	Point of neighbourhood.
$\boldsymbol{R}$	Resultant set containing ids of the matching documents.
$\boldsymbol{R1}$	Resultant set containing keywords that occur together with the search keyword $\boldsymbol{K_s}$.
$\boldsymbol{S}$	Set of search keywords for Boolean AND and OR queries.
$\boldsymbol{SI}$	Set of keywords extracted from a document for indexing.
$\boldsymbol{S_k}$	Search keyword specified before the NOT operator in Boolean NOT query.
$\boldsymbol{SK_{count}}$	Number of search keywords used in a query.
$\boldsymbol{T_i}$	Average time consumed for indexing $\boldsymbol{D_{count}}$ documents.
$\boldsymbol{T_s}$	Average time consumed for executing a query.
$\boldsymbol{X_{random}}$	Random number.
$\boldsymbol{Y_{random}}$	Random number.

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CRF	Conditional Random Fields
DAWG	Directed Acyclic Word Graph
DBOW-PV	Distributed Bag Of Words version of Paragraph Vector
DIG	Document Index Graph
DMPV	Distributed Memory version of Paragraph Vector
DOMI	Dynamic Ordered Multi-field Index
ERP	Enterprise Resource Planning
GB	Gigabyte
GBI	Graph-Based Index
GenEx	Genitor and Extractor
Ginix	Generalized Inverted Index
GSS	Graph-based Similarity Search
HMLists	Hashed Multiple Lists
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KECNW	Keyword Extraction using Collective Node Weight
KETNEW	Keyword Extraction from Tweets using Node and Edge Weight
LS-AMS	Log-Structured Adaptive Merge Strategy
LSII	Log-Structured Inverted Index

LSM-tree	Log-Structured Merge Tree
MB	Megabyte
MIL	Multi-Layer Inverted Index
NLP	Natural Language Processing
NSERC	Natural Sciences and Engineering Research Council
OpenJDK	Open Java Development Kit
PHP	Hypertext Preprocessor
RAKE	Rapid Automatic Keyword Extraction
RCDC	Research Computing and Development Cloud
RDF	Resource Description Framework
RDMA	Remote Direct Memory Access
SVM	Support Vector Machines
SvS	Small-Vs-Small
TF-IDF	Term Frequency and Inverse Document Frequency
TI	Tweet Index

Chapter 1

Introduction

There is a continuous growth in the number of users for social networking websites such as Facebook, YouTube, Quora, and Twitter. It has been estimated that on average, 500 million tweets are being generated per day by the users on Twitter itself [1]. Different types of content such as texts, images, and audio are generated by billions of people each day leading to a massive data management requirement. Thus, new unique search and retrieval strategies depending on the type of data must be developed to serve the increasing user workload on the internet. Many techniques have been proposed to optimize data retrieval speed. Some of the techniques are data indexing, data sharding, Increasing CPU/Memory capabilities, and optimizing query construction [2] [3].

Data indexing techniques are employed to reduce the query processing time by limiting the number of times disk access is performed. The index is usually made up of multiple data structures depending on the type of data indexed or type of query being executed. Data sharding is a technique where a big database is broken into

smaller ones called shards. Related shards are kept together in a separate system distinct from unrelated shards. This kind of setup allows redirecting the query to the relevant database subsystem which results in easy data management and faster data retrieval. High-performance CPU and high-speed memory modules can help in faster query execution. However, due to the hardware upgrade limitation and pricing factor, an upgrade may not justify the benefits of faster content delivery. Inefficient framing of a query (i.e. improper usage of joins and condition clauses) can cause data search performance degradation. Optimizing the construction of queries can mitigate the search latency problem to some extent.

In information retrieval, search queries, in general, can be classified as simple, Boolean, phrasal, and wild card queries [4]. A simple query contains only a single search keyword and results in a set of documents containing that search keyword. For example, searching for "*Ottawa*" will return all the documents that contain the word "*Ottawa*". These queries are becoming uncommon as they may return large sets of uninteresting documents which are difficult to narrow down as per user need. Phrase queries return documents in which the search keywords appear in a specific order. Phrasal queries are most useful for finding popular documents with famous phrases. For example, searching for "*You too, Brutus*" in Shakespeare collections. Wildcard queries return documents that contain the search keyword matching a wildcard pattern. For example, searching for "*Ottawa is ** " will fetch all the documents that contain a phrase starting with "*Ottawa is*". In this thesis, the primary focus is on the Boolean queries. Apart from Boolean queries, two other types of queries called neighbours of a keyword and exclusive keyword queries are also discussed. These

queries are discussed next.

A Boolean query is constructed using one of the Boolean operators (AND, OR, NOT) between search keywords. For example, searching for "*Anthony* AND *Brutus*" will return only the documents that contains both *Anthony* and *Brutus*. Boolean queries find their application in many areas such as Google searches, social media, and database systems. In a Google search, Boolean queries are used to narrow down the search results to relevant links [5]. Boolean queries help the user to effectively monitor social media to get the latest updates about trending social events [6]. In databases, Boolean queries simplify writing a complex condition in a query clause, but improper construction of Boolean queries can generate different results than expected [7]. With the importance of the Boolean queries being said, it is essential to make the execution of Boolean queries fast and reliable.

Neighbours of a keyword queries are a type of queries where the result is the list of keywords that appear together with the search keyword. This type of query is useful to quickly find all the keywords that are associated with a given search keyword. For example, consider indexing product catalogs, by searching for a single product type like a wristwatch, all the brand names used in the catalogs that are associated with the keyword wristwatch can be retrieved. Another example is its application in Enterprise Resource Planning (ERP) tools. Consider a company manufacturing tens and thousands of products. Specific spare parts such as a half-inch bolt could be used in many products. Thus, searching for a half-inch bolt will result in all the products that use the half-inch bolt.

The exclusive keyword queries are a type of queries where the results are the documents that contain only the search keyword, no other keyword will be co-existing with that keyword in the resultant documents. This type of query is useful to quickly check if there is any exclusive content available for a search keyword. For example, consider indexing movies and TV show titles hosted by different online video streaming services such as Netflix, Amazon, and Hulu. An exclusive keyword query using the keyword "Netflix" retrieves exclusive content that is only available on the Netflix platform. This query can also be used in ERP tools for finding whether a product uses any exclusive items.

1.1 Motivation

Data indexing is one of the many fields in which active research has been done to build faster data access systems. Many data structures for text document indexing have been proposed. One of the most common indexing data structures for text documents is Inverted Index [8].

Inverted Index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears. Thus, by searching for a word in Inverted Index, we will get the list of documents on which they appear. Search engines like Google use Inverted Index to index web pages to fasten the web search time [9]. Lucene-based NoSQL databases like Elasticsearch use Inverted Index as the primary index structure for full-text searching [10]. One other

application of Inverted Index is in the field of the microblog. A microblog is like a normal blog except for the fact that the content size is limited. For example, the famous microblog website Twitter has a restriction of 280 characters.

When a large number of textual documents are indexed, the execution of a Boolean query becomes computationally expensive in Inverted Index. All the necessary keywords have to be accessed and the ids of the documents in which they appear have to be retrieved. Then, depending on the type of query (AND, OR, or NOT), different operations such as intersection, union, or difference have to be carried out. It becomes increasingly complex when there are more search keywords involved in a query and millions of documents to search for. In applications like web search engines, where there is no need for all the resultant document ids at once, techniques like page ranking have been used [11]. The documents or web pages to be indexed have been given a ranking based on various factors like popularity or number of hits and only the top K results are retrieved where K represents the number of resultant document ids. The same ranking concept has been used with micro-blog indexing [12] as well. However, for applications requiring all the results of a Boolean query at once, scoring the documents becomes irrelevant. All the documents are considered equally important. Thus, adopting ranking optimization and returning only the top K results will not work. The resultant set containing all the matching documents should be retrieved at once.

Executing the neighbours of a keyword query also needs a lot of computation in Inverted Index. The result of this query is the set of keywords that have occurred

together with the search keyword. To execute this query, we need to access all the keywords and their list of documents to check whether there is any common document or not. This will be cumbersome if the document list for the search keywords is very large and has very little common documents with other keywords indexed.

Executing the exclusive keyword query with Inverted Index involves removing common documents between search keywords and other keywords indexed. The remaining documents found in the search keyword's document list is the resultant set. Thus, this type of query also depends on the search keyword's document list size and how often the search keyword appears with the other keywords indexed.

As mentioned earlier, Inverted Index is inefficient in dealing with different types of queries. In fact, depending on the queries, several comparisons have to be made to get the result. This thesis researches techniques for alleviating the problems described in the previous paragraphs. By devising an effective technique to index the data, a better search time could be achieved especially for search operations performed by web search engines such as Google Web Search or micro-blog search engines such as Twitter Search.

1.2 Research Objectives

The objectives of this thesis are summarized as follows:

1. The foremost objective of this thesis is to create a new indexing technique to

provide faster search capabilities for a larger number of textual documents.

2. Create a new algorithm for executing Boolean queries. The proposed approach should handle all the operators of Boolean queries: AND, NOT, and OR. The new algorithm should also get all the matching document ids of a Boolean query at once.

3. Devise an efficient algorithm for executing neighbours of a keyword queries. The proposed approach should get all the neighbours associated with the search keyword.

4. Create an algorithm for executing exclusive keyword queries. The proposed approach should quickly check for any exclusive documents that exist for the given keyword and retrieve it.

5. Compare the performance of the proposed graph-based indexing technique with Inverted Index and an enterprise-level search engine.

1.3 Overview of the Proposed Approach

The proposed approach uses a directed graph data structure. The graph data structure is used to establish a link between the keywords that appear together in a document. This results in a keyword-keyword-document relationship as opposed to the keyword-document relationship found in Inverted Index. This triangular relationship

of keyword-keyword-document is used for executing Boolean queries, neighbours of a keyword, and exclusive keyword queries effectively.

1.4 Contributions

Based on the objectives listed in the previous section, the main contributions of this thesis include:

1. A novel graph-based text document indexing technique using Graph-Based Index (GBI) structure. GBI provides an efficient way to store data and relationships between keywords that are extracted from a set of documents.

2. New algorithms for executing the various type of queries which demonstrate the advantage of establishing a relationship between keywords and their impact on query processing for getting all the matching results at once.

3. A proof of concept prototype for demonstrating the effectiveness of the proposed graph-based indexing technique.

4. A detailed set of insights into system behaviour and performance resulting from extensive measurements made on the prototype subjected to a synthetic workload.

    - Performance analysis to validate the proposed approach. The results demonstrate that the proposed approach can improve the performance

of the Boolean queries, neighbours of a keyword queries, and exclusive keyword queries.

- A performance comparison between GBI and Elasticsearch, a popular enterprise-level search engine that uses Inverted Index.

1.5 Publication

The publication resulted from this research (so far) is presented.

- A. K. Mohideen, S. Majumdar, M. St-Hilaire and A. El-Haraki, "A Graph-Based Indexing Technique to Enhance the Performance of Boolean AND Queries in Big Data Systems," 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, Australia, 2020, pp. 677-680, doi: 10.1109/CCGrid49817.2020.00-24. [13]

1.6 Thesis Outline

The rest of the thesis is structured into four chapters. Chapter 2 – Background and Related Work, outlines the related work and different approaches to solve the problem of textual data indexing using Inverted Index. Then, various optimizations for executing Boolean queries more efficiently are discussed. Finally, an overview of relevant concepts in graph theory and hash table data structure is provided with their

applications in textual data indexing. Chapter 3 – Graph-Based Indexing Technique, outlines the proposed Graph-Based Index structure and describes the algorithms used to execute queries when textual documents are indexed using GBI. Chapter 4 – Experiments and Performance Analysis describes the synthetic workload used for performance analysis and discusses the performance results obtained from comparing GBI with Inverted Index prototype and Elasticsearch. Chapter 5 – Conclusions and Future Work presents the conclusions derived from the experiments conducted and directions for future work.

Chapter 2

Background and Related Work

This chapter explores important concepts such as Inverted Index, hash table, keyword extraction techniques, and graph theory which have shaped the proposed solution. It also covers a representative set of related research works on data indexing using graphs and Inverted Index.

2.1 Inverted Index

2.1.1 General Concepts

The authors of [8] explained the basic concepts and terminology of Inverted Index in the context of information retrieval. Inverted Index is a mapping of unique meaningful words extracted from the documents set to be indexed to their corresponding postings list. The extracted words are also referred to as terms or keywords. The process of extracting meaningful words from a document is explained in Section 2.3. Inverted

Index is also referred to as an inverted file. A document can be defined as a source of textual data from which meaningful unique keywords can be extracted. A document is sometimes called the corpus (a body of text). Examples of such documents include tweets and web pages. Each document is tagged with a document id (a serial number) so that it can be uniquely identified. These document ids identify a set of documents that contains the keyword under consideration. A postings list for a keyword is defined as a collection of document ids. The postings list of a keyword can also contain other measurements such as frequency and position of the keyword in all the documents it appears in. The collection of all postings lists is called as postings.

Consider the keyword *Anthony* appearing in documents *D1, D2* and *D5*, the keyword *Caesar* appearing in documents *D1, D2, D3 and D4* and the keyword *Brutus* appearing in documents *D1* and *D4*. Inverted Index structure for this example is shown in Figure 2.1.

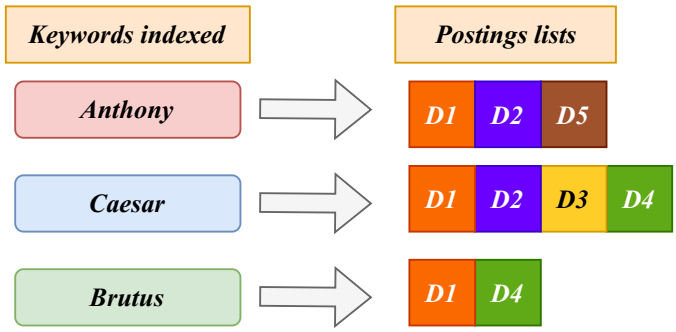


Figure 2.1: Example of Inverted Index.

The authors of [4] talked about three different implementations of Inverted Index based on what supporting information is stored in the postings list of the keyword

indexed. The first implementation contains only the keywords with document ids. The second one is keywords with document ids and their occurrence frequency in the documents they appear. Finally, the third implementation contains keywords with document ids and their position in the documents they appear in. For executing various queries discussed in Chapter 3, storing information about the document ids of the keywords indexed is alone sufficient. Thus, in all the experiments presented in this thesis, the first implementation of Inverted Index is used.

2.1.2 Representative Set of Works on Inverted Index

The representative set of works listed next covers the optimizations performed on Inverted Index and notable applications of Inverted Index.

2.1.2.1 Optimizations Performed on Inverted Index

Some research has been done in the area of increasing the search efficiency of Inverted Index. The concept of using a Bloom filter with Inverted Index was proposed in [14]. A Bloom filter is a memory-efficient data structure that can tell us with certainty that the element we are trying to find is not present, if it is really not present in the set. If the element we are trying to find is present in the set, then the Bloom filter will tell us that it might be present without any certainty. This means that a Bloom filter will never return a false negative response in well-defined operational circumstances. The advantage of avoiding false negative cases using a Bloom filter

is combined with Inverted Index to reduce the search time when the keyword that is being searched does not exist in Inverted Index in the first place. This research paper partially inspired the work presented in this thesis to build an index structure that inherently handles the case where keywords in the Boolean queries do not occur together in any document at all. The authors of [15] proposed a modified Inverted Index structure called Generalized Inverted Index (Ginix). It is similar to a regular Inverted Index except for the postings list of the keywords indexed. In Ginix, the consecutive document ids of a postings list are grouped to form concrete intervals to save memory space. Thus, the postings list of keyword *Caesar* appearing in documents *D1, D2, D3 and D4* is stored as [*D1, D4*]. The union and intersection operations are performed on the intervals directly without expanding. Though this method shows a better memory-efficient search performance for the use case of keywords appearing in consecutive document ids, the performance will suffer if the keywords are sparsely distributed in the documents to be indexed.

For solving any Boolean queries, the postings lists of the keywords found in Inverted Index need to be traversed to compare document ids. The authors of [16] used the concept of skip list to speed up the postings list traversal. In a traditional traversal of the postings list, each document id is visited to compare against other postings lists. In the skip list method, a certain number of elements in the postings list are skipped from visiting. To achieve this, skip pointers are placed on certain elements of the postings list. This way, when the list is iterated, the elements selected as skip pointers are bypassed if they are unessential for computation, thus avoiding the traversal of the entire postings list. The position of skip pointers in the postings

list plays a vital role in the search process. Too many pointers result in more skip comparison and too few pointers result in fewer opportunities to skip. The general heuristic is to use $\sqrt{L}$ equally placed skip pointers for a postings list of length L. The result shows that the usage of the skip list is irrelevant for search performance as the proposed method targets to reduce the total number of postings list used.

The authors of [17] proposed an optimized scoring mechanism for extended Boolean retrieval to get more relevant documents. In an extended Boolean retrieval queries, in addition to having normal Boolean operators (AND, OR, NOT), it also has a proximity operator. A proximity operator tells how close two keywords should appear together in the document to be eligible for the resultant set. The closeness is defined in terms of how many words, sentences, or paragraphs should appear between two keywords. The authors of [18] pruned the search keywords in a query instead of search results to lower the query execution time. The existence of common keywords in the queries has the least impact on the result. Thus, by pruning those search keywords before the execution of the ranked queries, the authors claim that they were able to attain reasonable precision in the results. The authors of [19] proposed a controlled ranking and pruning strategy where the quality of the result of the queries depends on the time and the system resource that has been allocated to the search queries. The authors used early search termination heuristics to get results with a reasonable quality during heavy system load and strategies for getting more refined results under less system load. The scoring or ranking mechanism will work when obtaining the most relevant document is enough. In applications, wherein all the documents are considered equally important, the scoring becomes irrelevant. The work

in this thesis addresses this issue of retrieving all the documents effectively when all of them are considered equally important.

The authors of [20] used a bitmap-based technique to optimize the Boolean AND queries execution. A bitmap is used to represent keyword co-occurrence. A bitmap is an array containing a sequence of 1's and 0's. A bitmap of size X is attached to each document id in the postings list of keyword Y to store the co-occurrences of Y with X other frequently co-occurring keywords. Thus, each bit in a bitmap represents one of X keywords that co-occur with keyword Y. The advantage of this method is that Y's postings list alone will be enough to handle any Boolean AND queries involving Y and any of the X keywords. The result will be document ids for which all the search keyword's bit is set to one in each of the document id's bitmap array found in Y's postings list. The disadvantage of this method is that it can perform Boolean AND execution only for the already selected list of keywords from the index. Also, this research paper does not discuss the usage of a bitmap in solving Boolean OR and NOT queries. The work proposed in this thesis addresses the shortcomings of this research paper by enabling all the Boolean operations over all the indexed keyword combinations.

2.1.2.2 Applications of Inverted Index

One of the popular applications of Inverted Index is its usage in Apache Lucene. Apache Lucene is an open-source search engine library written in Java supported by the Apache Software Foundation. Apache Solr and Elasticsearch are some of the

popular enterprise search engine projects built out of Lucene [21] [22]. A review of Lucene's capabilities is conducted by the authors of [23]. They explained the Lucene architecture offering full-text indexing and searching capabilities. The researchers provided a comparative analysis of Lucene with other search tools such as Hoot and Suffix tree. They concluded that Lucene is a better option for quick indexing and searching for textual documents. The performance evaluation of Lucene concerning massive indexing and searching for short text messages is conducted by the authors in [24]. The authors compared Lucene with Oracle Text and concluded that Lucene is better than Oracle for short textual documents. The authors also made some key remarks on Lucene: its memory, not the CPU, plays a vital role in searching but in terms of indexing, both powerful CPU and larger memory are needed. The authors of [17] [22] report that Lucene uses skip pointers to traverse the postings list for conjunctive operations (Boolean AND operations) with guaranteed logarithmic access times to any postings.

Inverted Index is used as a key data structure for indexing in the field of micro-blogs. Several research works used Inverted Index to address the indexing of micro-blog have been proposed. One such work [25] studies the evolution of tweets using a Multi-Layer Inverted Index structure (MIL) to monitor the evolution of social events. Each entry in every layer of the MIL contains a set of keywords pointing to a set of events that contains those keywords. Each layer in the MIL is organized in such a way that it contains as many keywords related to an event as possible. Thus, the topmost layer of the MIL contains more keywords that represent an event. This multi-layered approach provides better abstraction in terms of search. Therefore, queries involving

several keywords related to an event can use the upper layer of the MIL to optimize search time.

The authors of the Tweet Index (TI) [26] proposed an index structure for the real-time search of tweets based on Inverted Index. The idea is to index only important tweets immediately and defer the indexing process for less important ones. This strategy helps in saving indexing costs providing relevant search results. To achieve this strategy, TI ranks the incoming tweets based on the popularity of the topic and the tweeting person. The high-ranked tweets are indexed immediately while the low-ranked tweets are written in a log file and indexed in bulk later.

The authors of the Log-Structured Inverted Index (LSII) [27] propose a log-structured Inverted Index which is a sequence of Inverted indices with an exponentially increasing size. The new incoming micro-blogs are first indexed in Inverted Index of smaller size and are later merged to larger-sized Inverted indices as time passes. Thus, most recent micro-blogs can be obtained from the smaller-sized index promptly. This idea of using multiple Inverted indices of increasing size originally came from the log-structured merge tree (LSM-tree) [28]. LSM-tree is one of the classic indexing techniques for files on a disk that faces a high insertion rate over a long period. The LSM-tree structure is used to reduce the disk arm movement as the disk had a low rotational speed a couple of decades back. The basic idea of LSM-tree is to differ and batch the changes to the index. The batched changes are migrated efficiently much like the working of merge sort. LSM-tree uses two or more tree data structures in increasing size instead of Inverted Index. Log-Structured Adaptive Merge Strategy

index structure (LS-AMS) [29] is Inverted Index based indexing scheme that uses the LSM-tree concept. The authors of LS-AMS claims that they provide better indexing and query execution time than LSII by adopting an efficient merge strategy. LSII uses a direct merge strategy meaning that all the smaller-sized indices are directly merged with its next larger-sized indices. In LS-AMS, the merge strategy is determined adaptively between direct and batch merging. For merging smaller indices, batch merging is adopted where the merging task is batched and done altogether. When merging fairly larger Inverted indices, a direct merging approach is used.

Though Inverted Index is mainly applied to index textual data in general, it is not uncommon to find indexing on other data types such as audio and video. The authors of [30] did a content-based semantic search on large video data by using a text-based Inverted Index. In this case, Inverted Index is used to store important and consistent features of a video. This helped them to search and select a video among 100 million videos rapidly. Audio streaming services such as podcasts are getting increasingly famous. The authors of [31] proposed an index structure called RTSI for audio stream data. RTSI is a collection of three Inverted indices. These three Inverted indices are used to store the popularity, the freshness, and the relevance score of an audio stream indexed, respectively. It helped the authors to solve the top K query results in a faster manner.

From the literature review related to optimization and applications of Inverted Index, it can be inferred that Inverted Index has a wide range of applications from powering Google web search engine to enterprise text search to short text indexing.

However, Inverted Index does not consider relationships between keywords. Therefore, we believe that by using a different form of indexing where a relationship is established between keywords extracted from different documents, we could deliver better performance in applications requiring all results at once.

2.2 Hash Table

In this section, we review the notion of hash table since it is an important data structure used for indexing and more precisely in Inverted Index. In the work described in Chapter 3, we use a hash table to construct our proposed index structure. The hash table is a data structure that is stored in memory. Both the CPU and memory overheads associated with a hash table are discussed in the following subsections.

2.2.1 General Concepts

A hash table is an associative array type data structure that maps unique keys to values for highly efficient lookup [32]. A hash table is a perfect data structure to have a string data type as an array index. A hash table converts a string key into an integer key using a hash function. The resulting hash can then be used as an index for the array to store the desired value. In a textual indexing use case, the key will be the keywords extracted from the documents and the values are its postings lists. A good hash function should convert the string to integer rapidly with as little

collision as possible. A collision is said to have occurred when two different keywords get converted to the same integer key. Collision resolution leads to an overhead associated with hash tables and is discussed in Section 2.2.3. Hash tables are better than normal arrays and linked lists as a lookup in these data structures takes linear time. In a sorted array, a lookup using binary search is fast, but insertions become difficult. In general, for most use cases, a hash table provides a better algorithmic complexity for lookup, insert and delete operations than a primitive data structure. hash tables are used to build dictionaries and sets in programming languages like PHP [33]. A sample hash table implementation of Inverted Index example presented in Figure 2.1 is given in Figure 2.2.

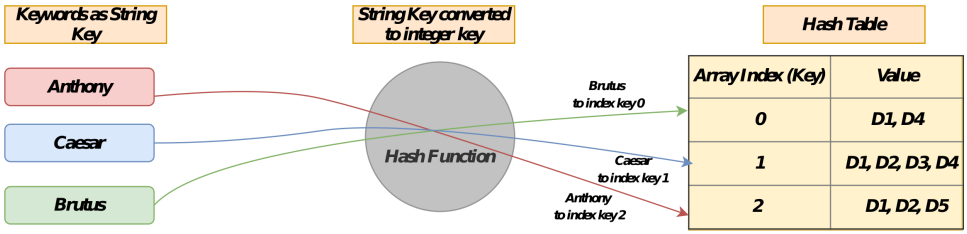


Figure 2.2: Basic hash table representation.

2.2.2 Basic Hash Table Construction & Search Performance

A simple implementation of a hash table is described by the author in [34]. An array of linked lists and a hash code function are used by the author to build the hash table. To insert a string key-value pair, the hash code of the string key is computed which returns the integer value. This integer value is mapped to an index in an array. The mapping is done by the formula (hash code % size) where size is the number of

entries to be inserted into the hash table. At the specified array index, a linked list data structure is maintained where the desired string key and its value are stored. A linked list data structure is used to avoid collisions. Since two string keys can map to the same numerical array index, a linked list is used to store all the keys and values that belong to the same array index. To retrieve the string key-value pair by the array index, the hash code of string key is computed and then this hash code is mapped to an array index. Then, the linked list in that array index is iterated to find the string key-value pairs. If the hash table is implemented with the linked list as described above and the number of collisions is very high, the worst-case lookup time is *O(N)*, where N is the number of string keys. Alternatively, if the hash table is implemented with a balanced binary search tree instead of a linked list, the lookup time will be *O(log N)* irrespective of the collision rate. However, in general, a good implementation of the hash table keeps collisions to a minimum, in which case the lookup time is *O(1)*. Thus, the implementation of a hash table effectively lies in the good design of the hash function. The authors of [35] reported two popular hash functions: SDBM and DJB2. The authors concluded that SDBM provides a good key distribution reducing the collision rate but takes a lot of computation time. DJB2 provides a perfect balance between collision reduction and execution speed. As a result, DJB2 is one of the preferred hash functions for strings. A hash function is highly dependant on the type of string keys being used so the type of hash function to be used should be decided based on the dataset. PHP language uses DJB2 hashing function [36] [37] for its string keys. In our case, since we are dealing with text documents and keys are keywords and used PHP to conduct the experiments, the

DJB2 hashing function is used to build the hash table.

2.2.3 Collision Resolution Strategies

Collision resolution is an important part of the hash table implementation. Collision resolution techniques affect the hash table performance. A collision is said to have occurred when two or more keywords map to the same index in the lookup table. Two types of resolution techniques available are chaining and open addressing technique. In chaining, a separate linked list or binary tree is maintained to store all the keywords-value pairs that map to the same location. Thus, a separate data structure needs to be maintained for every entry of the hash table. This is the technique used in the basic hash table construction explained in Section 2.2.2. In the chaining technique, there is an overhead of maintaining an additional data structure. In open addressing, all keywords are stored in the hash table itself. When a new keyword has to be inserted, the potential positions for inserting are examined, starting with the initial hashed-to position and proceeding in some probe sequence, until an unoccupied position is found in the hash table. When searching for an entry, the positions are scanned in the same sequence, until the target keyword is found. In this method, no additional data structures are used. In open addressing, the following types of probes are found.

1. Linear probing: The free slot in the hash table is found by linearly searching. Typically, for every probing, the next slot to the current slot is checked.

2. Quadratic probing: The free slot in the hash table is found by adding a quadratic polynomial unit for every probing. Typically, for every i^{th} probing, i^2 slot to the current slot is checked.

3. Double hashing: The free slot in the hash table is found by using a second hash function.

The open addressing scheme consumes more time during insertion and search than a chaining scheme depending on the probe used. PHP uses a chaining method with a linked list to handle collisions in its hash table implementation to support faster search operation [37]. Hence, the hash table used to demonstrate the proposed method uses a chaining strategy. This certainly comes with the cost of increased memory consumption to store additional linked list data structure with the hash table.

2.2.4 Hash Table Load Factor

One of the important statistics for measuring a hash table performance is the load factor. A load factor is defined as a ratio between the number of key-value entries and the number of buckets in the hash table [38]. The load factor gives the average number of entries found in a bucket. The increase in load factor denotes that there are more entries found per bucket which in turn signifies that there may be a lot of collisions happening when search operations are performed. A low load factor ensures a faster search operation (as there are fewer collisions) at the expense of wasting memory (several buckets will be left unoccupied). Thus, care should be taken to maintain

the load factor below a given limit to get a constant search time without consuming too much memory. When the hash table hits the load factor limit, the hash table is resized to ensure that it stays within the fixed load factor limit. PHP's hash table uses a load factor limit of 1 to maintain a good trade-off between time and memory costs [33]. Thus, when the total number of entries is equal to the number of buckets, a resizing operation takes place.

2.2.5 Hash Table Re-Sizing

Hash table size is usually proportional to the number of entries to be inserted. If the number of entries is not known in advance, the table must be resized when the lists become too long to be accommodated. Once the hash table reaches its full capacity or when it exceeds its load factor, it undergoes a process of resizing. There is an overhead associated with resizing as this process includes creating a new larger table, and each record in the older table getting mapped to a newer table. Since resizing is an expensive operation, the size of a hash table is usually increased by a constant factor for each resizing, i.e. by doubling its size or increasing size in the power of two [33]. This constant factor of resizing avoids resizing often and gives an amortized constant time for insertion. Thus, insertion and search performance of the hash table won't suffer much during the resizing operations [39].

2.2.6 Application of Hash Table in Data Indexing

The authors of [40] used the hash table structure to build Inverted Index and gain a significant search performance by exploiting its fast lookup characteristics. Earlybird [41], the core search engine for Twitter, uses Inverted Index to serve real-time search requests. Earlybird uses Inverted Index formulated using hash tables with open addressing collision resolution policy. Two types of Inverted indices are used in Earlybird. One for active indexing of tweets and the other for optimized query execution. LiveIndex [42] is another micro-blog indexing system that uses distributed time range partitions with each partition containing Inverted Index as the core data structure to address temporal queries with a specific time range. LiveIndex also uses Inverted Index formulated as a hash table for faster access. LiveIndex takes advantage of a distributed computing paradigm to redirect the query to a particular Inverted Index located in a particular node to execute the query. This setup helps in handling concurrent query executions in a timely fashion providing real-time search service. The authors of the Dynamic Ordered Multi-field Index (DOMI) [43] use a Radix tree as an index structure for the key-value data. The authors used a hash table to optimize search performance. The hash table stores the address of each node of the Radix tree. This allows the search process to directly visit the node and avoid traversing the tree from the beginning. The authors claim that this setup performs better than a B+ tree Index structure for key-value data. The authors in [44] proposed an indexing scheme called hashed multiple lists (HMLists) for processing Continuous queries for Geo-sensor data. This paper focuses on indexing a wide range of data intervals.

A continuous query is defined as a query in which the system will be continuously polling the incoming data to answer a particular query. The HMLists is a hash table implemented as a Binary tree structure. Since the incoming data is continuously monitored, faster insertion and access are required to meet the real-time constraints. Since HMLists are a hash table, it provides a constant lookup time providing quick access to a wider range of interval values returning a faster response to the query.

The search performance benefits reported by the research papers discussed above inspired the work presented in this thesis to adopt a hash table as the core data structure to build the proposed index structure.

2.3 Selecting Keywords from Documents for Indexing

In general, not all the words in a document are indexed. The document to be indexed undergoes various keyword filtering and extraction process to get the potential keywords for indexing. This sub-section briefly describes the document pre-processing and keyword extraction procedures. An overview of the different steps involved in the keyword selection process, as explained in [4] [45] [46], is shown in Figure 2.3. It should be noted that not all the steps listed in Figure 2.3 are mandatory. For some use cases, the pre-processing step alone might be enough. In other cases, keyword assignment or keyword extraction steps are needed. In short, depending on the use

case, the output of each step can itself be considered as the final list of keywords for indexing. The next subsections describe each step of the keyword selection process.

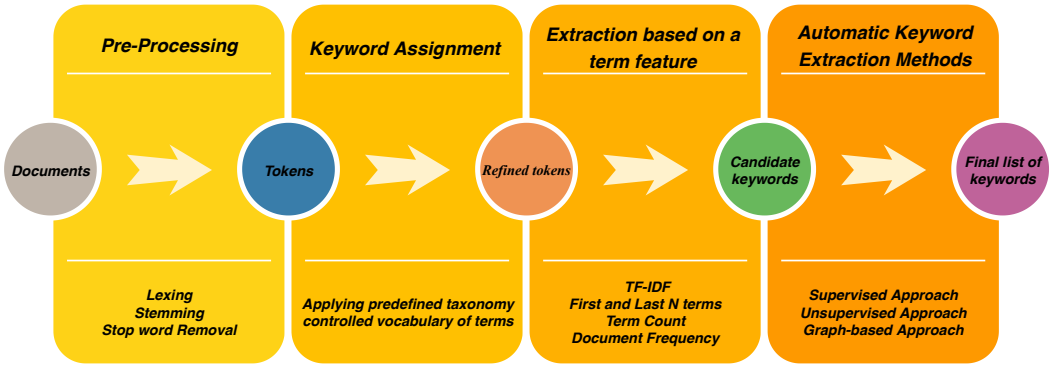


Figure 2.3: Selecting keywords for indexing.

2.3.1 Documents Pre-Processing

The pre-processing involves one or all of the three steps: lexing, stemming, and stop word removal [4]. Lexing is the process of converting a list of characters to a list of tokens by parsing the document. Each such token is a single alphanumeric word. The tokens are usually restricted to certain characters length (typically 32 characters) and are converted to lowercase. Tokens having too many numerical characters are pruned as they are not very useful in searching. After lexing, each token is then transferred into its morphological root, a process called stemming. For example, the words "*calculate*", "*calculation*", "*calculators*", "*calculated*" and "*calculating*" will all be indexed as "*calculate*". One of the popular English language stemming algorithms is Porter's stemming algorithm. Due to all the exceptions in the various language grammars, it is increasingly becoming tougher to do stemming. For this reason, most

search engines, including Google, ignore the stemming process. Stop words removal is the process of removing common words that almost occurs in any textual data like articles and preposition to name a few. Sometimes a few stop words such as "*How*" and "*Who*" are used in the query. For this purpose, not all stop words are removed rather a stop word list file is constructed. A stop word list file contains the desired stop words to be removed from the document.

2.3.2 Keyword Assignment

In the keyword assignment, a predefined list of words is made. Keywords are chosen from the document only if they exist in the list. Keyword assignment bounds the set of keywords to a confined vocabulary of words or taxonomy [45]. The aim is to find a limited set of keywords that best describes an individual document. The keyword assignment is used to categorize documents based on the subject. The advantage of keyword assignment is that similar documents are represented by the same keywords which result in a reduced Index size. The other use is that a controlled vocabulary ensures that the documents covered do not go beyond the scope of the subject. This restricted scope helps in filtering unwanted documents from getting indexed, saving indexing time, and memory consumption. There are some weaknesses associated with the keyword assignment procedure. A successful keyword assignment procedure depends on the correctness and exhaustive list of desired words in the predefined list. In most of the cases, it is difficult to create an accurate and complete list to cover a subject entirely. This results in ignoring potential keywords from the documents

if they are not in the predefined list of vocabulary. The other weakness is that the confined list of vocabulary may not be available for all subjects. It may also be very expensive to create or update one making this process highly inefficient for some use cases. The result of the keyword assignment will be refined tokens representing documents of a particular subject.

2.3.3 Extraction Based on a Term Feature

Keyword extraction is defined as the process that automatically identifies a set of terms or keywords that best describes the content of a document [46]. In keyword extraction, the selection of keywords does not depend on any predefined vocabulary list. The words are directly extracted from the document. This is the main difference between keyword assignment and keyword extraction. Since the keyword extraction process alone does not restrict the scope of the documents to a particular subject, it is a good practice to use the keyword assignment step along with the keyword extraction process.

The decision of extracting candidate keywords is governed by some features or properties of a term (word) in the document. The features of a term decide whether it is eligible to be in the final list of keywords from the refined tokens. The authors of [45] listed some common features that are considered while extracting candidate keywords. One of the features is based on *term weights*. *Term weights* determine the importance of a particular keyword in a document. The term weighting measures are:

- *Term count* - It is the number of occurrences of a particular term in a single document.

- *Term frequency* - It is the total number of occurrences of a particular term in all the documents that are to be indexed.

- *Document frequency* - It is the inverse of term frequency which measures the total number of documents that have a particular term.

- *Inverse document frequency* - It is based on the principle that rarer a word occurs in a document set, the more relevant and important to the documents it appears in.

- *Average frequency* - It is the ratio between the total number of occurrences of a term in the entire document set and the total number of documents.

- *Relative frequency* - It is the same as average frequency except for the total number of documents. In *relative frequency*, only the documents that a word appears in is taken into account.

- *Term length* - It determines the character restriction for a candidate keyword.

- *TF-IDF* is the product of *Term frequency* and *Inverse document frequency*. The higher the score, the more relevant that term is in that particular document. *TF-IDF* is one of the most common features employed in scoring the keyword importance for indexing.

The other features used to select candidate keywords are based on the *location of a term* in the document. Features in this category include:

- *first/last N terms* - As the name indicates, *first and last N terms* select only the first or last N number of terms from the documents respectively.
- *at the beginning/at the end of a paragraph* - A term's relative position to the beginning and end of the paragraph is determined and is selected respectively.
- *resemblance to title* - A term is rated to its similarity to the title of the document. The higher the rating, the more the term is preferred.
- *maximal section headline importance* - It is similar to the *resemblance to title* feature except for the tile, the headline is used as a comparison metric.
- *accumulative section headline importance* - It weights a term according to all its presences in important sections of the article.

Depending upon the use case, one or multiple features are applied to select candidate keywords from the documents.

2.3.4 Automatic Keyword Extraction Methods

In most of the cases, the candidate keywords provided by the keyword extraction based on a term feature itself will be quite sufficient. To obtain a very fine set of keywords destined for very exclusive applications, different automatic keyword

extraction methods have been proposed. Automatic keywords extraction methods can be broadly classified into *supervised*, *unsupervised* and *graph-based* [46]. Some of the automatic keyword extraction methods also use term features (discussed in Section 2.3.3) in addition to its process of extracting keywords. The graph-based keyword extraction technique is described in Section 2.4.2.2. A representative set of supervised methods are discussed next.

The main idea of the supervised approach is to transform keyword extraction into a classification problem. A supervised approach system called GenEx [47] uses a decision tree induction algorithm to classify a word as a potential keyword or not. The authors of [48] combined machine learning scheme called Bayes technique with the TF-IDF method to classify potential keywords from unwanted ones. The authors of [49] used the Natural Language Processing (NLP) technique to improve the machine learning approach like Support Vector Machines (SVM) to automatically extract keywords from scientific journals ignoring irrelevant ones. The authors of [50] proposed a keyword extraction method using conditional random fields (CRF). CRF is a sequence labeling model that treats keyword extraction as a string labeling task. It has been claimed that CRF performs better than machine learning models.

Unsupervised methods, unlike *supervised*, are self-organized meaning that they can perform without any pre-existing learning data or models. For example, the authors presented an *unsupervised* model [51] that uses n-gram lists to extract keywords from titles and abstracts of the documents. Some of the n-gram lists are uni-gram, 2-grams, 3-grams, etc. A uni-gram is one word. A 2-gram means a sequence of

two words, and so on. In general, an n-gram means a sequence of n words. The final keywords are obtained by merging the sorted n-gram lists eliminating the stop words. The authors of [52] proposed a method that uses clustering. The idea is to cluster a set of lines from the documents that are semantically related. The clusters formed are individually analyzed to get the main gist of the text. The important keywords from the main gist are chosen as keywords. The authors of [53] proposed a subject domain-independent keyword extraction system. In this system, the authors used the linguistic knowledge and features of a term (such as term frequency and term position) to extract keywords. The authors of [54] proposed an *unsupervised* extraction method based on Shannon's entropy difference. It depends on the idea that word occurrences in a document are controlled by the author's motive. The potential keywords will reflect the author's writing intention whereas irrelevant keywords are randomly distributed. These irrelevant keywords are neglected during the extraction process.

One of the recent developments in the unsupervised approach concerning textual document analysis is the advancement in document embedding techniques. A document embedding is a process by which the words found in the documents are represented numerically. One of the recent methods for document embedding called paragraph vector also referred to as doc2vec (document to vector) was proposed by Google Engineers [55]. The doc2vec method produces a numerical representation of a document of any length. The doc2vec is a mere extension of word2vec [56]. In word2vec, different words are represented in numerical vector form. In one implementation of word2vec, the different words (in vector format) are given as input and

a single context word that best describes the different input words is produced as output. In one of the doc2vec method implementations called Distributed Memory version of Paragraph Vector (DMPV), in addition to word vectors, a document vector that represents an entire document is also used as input. In another doc2vec implementation called Distributed Bag Of Words version of Paragraph Vector (DBOW-PV), only the document vector is used as input. The objective (or output) of both implementation methods is to predict a context word that best describes a set of documents from which the document and word vectors are generated.

The authors of [57] performed experiments to validate the doc2vec method with other methods such as word2vec and n-gram model (discussed earlier) and observed that the doc2vec is performing better than the two other methods. They also concluded that the DBOW-PV implementation is performing better than the DMPV implementation of doc2vec. The main applications of doc2vec are its usage in sentiment analysis and document clustering. The doc2vec method can also be used for selecting the best keywords for textual data indexing purposes. The authors of [58] used doc2vec for capturing the semantic relationship between different words in a document to extract potential keywords from short textual documents such as tweets.

Since the focus of the thesis is on proposing a new indexing technique superior to Inverted Index for performing an efficient Boolean search, a fixed list of keywords is generated for experimental purposes adhering to the steps listed in the document pre-processing procedure.

2.4 Graph Theory

Graph theory is the study of graphs. Graphs are mathematical structures used to represent a pairwise relation between two entities. This section discusses the basics of graph theory and its application in the field of text processing and indexing.

2.4.1 General Concepts

The authors of [59] described the general concepts in graph theory. A graph represents a network that consists of a set of mathematical objects called vertices or nodes. These nodes are connected, with the help of arcs or edges, based upon some common relation or traits existing between them. Based on the type of edges, a graph can be classified as follows:

1. Directed graph or directed network: This category of graphs is characterized by the association of direction with the edges. In a directed graph, the edges are represented by an arrow. The direction signifies how a node can be accessed. A directed graph contains an ordered pair of vertices.

2. Undirected graph or undirected network: In an undirected graph, the edges between vertices do not have any particular direction. An edge in the undirected graph is usually represented by a straight line. An undirected graph contains an unordered pair of vertices. In the undirected graph, a node can be accessed in any direction.

An example of two types of graphs based on the type of edge is shown in Figure 2.4. It can be seen that in the directed graph, as an example, one can navigate from vertex *3* to *2* but the reverse is not possible. In case of undirected graph, both vertex *3* and *2* are visible to each other.

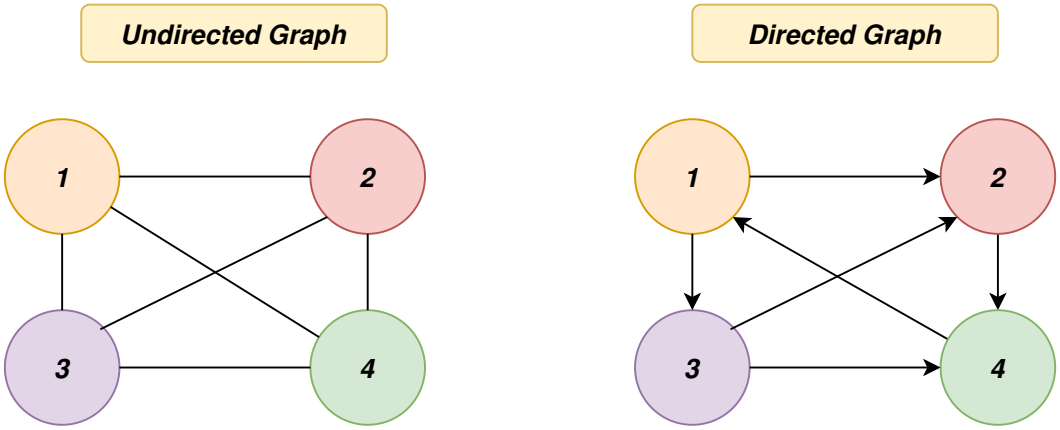


Figure 2.4: Example of an undirected and directed graph.

A graph can be further classified based upon the existence of cycles in it as follows:

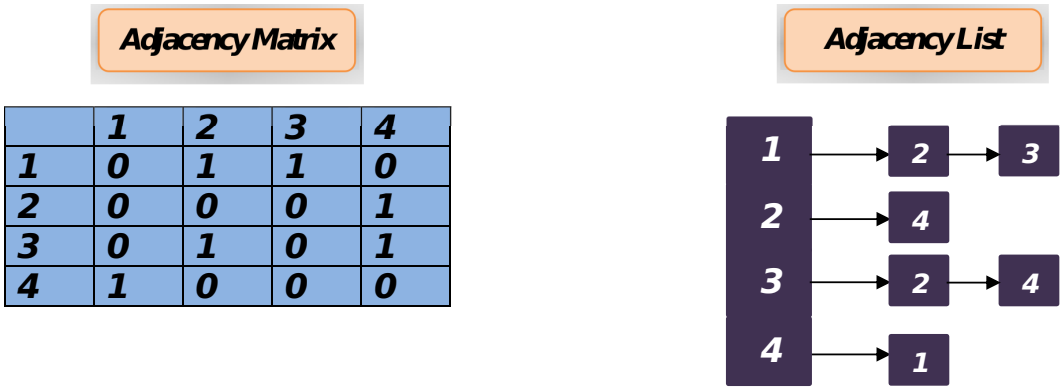
1. Cyclic Graph: A cycle is said to exist in a graph if some of the vertices form a closed chain where the starting and ending vertices are the same. A graph containing one or more cycles are called cyclic graphs.

2. Acyclic Graph: If the graph does not contain any cycle in it then it is called an acyclic graph.

A graph can be represented using one of the two following ways:

1. Adjacency Matrix: For a graph containing V vertices, an adjacency matrix is a V x V square matrix containing either 0 or 1. The entry represented by row i and column j is 1 in the adjacency matrix if and only if there exists an edge between the two vertices i and j. The value 0 is used to represent the absence of an edge between two vertices. Adjacency matrix though occupies more space because of storing information about the absence of the edge between vertices, it provides a constant access time *O(1)* to check whether there exists an edge between two vertices.

2. Adjacency List: In the case of an adjacency list, entries that contain zeroes in the adjacency matrix are eliminated and only the neighboring nodes of a particular node are stored. For each vertex i, an array of the vertices connected to it are stored. Typically, for V vertices in a graph, there are V adjacency lists, one for each vertex. The adjacency list is used when storage space is of primary concern. The algorithmic complexity for checking whether there exists an edge between two vertices i and j in adjacency list is *O(d)* where d is the number of edges, the node i or j has. The adjacency matrix and the list representation for the directed graph example given in Figure 2.4 can be seen in Figure 2.5.



	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	0	1	0	1
4	1	0	0	0

Figure 2.5: Adjacency matrix and list representation for directed graph example given in Figure 2.4.

The proposed indexing technique is built using a directed graph structure. For representing the graph structure, a modified form of the adjacency matrix is used in which the nodes which do not have any edges between them are ignored from representing in the matrix to save space. The proposed indexing technique is detailed in Chapter 3.

2.4.2 Applications of Graphs

Graph structures have many applications in various fields including but not limited to data indexing and keyword extraction. This section covers a set of representative research works on these areas.

2.4.2.1 Indexing

The authors of [60] proposed a graph-based indexing scheme for Arabic documents. In this work, each document is modeled as a graph and a term weighing feature

is applied to find the relevancy of a term to a document. The authors claim that by using a graph, document modeling is more expressive than a standard bag of words approach. A bag of words approach is the simplified way of representing a document. The bag of words approach lists the words with their frequency ignoring grammar and order in which they appear in the document. The research work also claims that graph-based indexing is better for contextual indexing and outperforms statistics-based methods like TF-IDF.

The authors of [61] investigates graph databases and indexing schemes for unstructured big data systems. A graph database represents a graph structures with a set of nodes and edges. The properties of a graph structure are used to store the relationship between different data points. Petabytes of data are converted into graph structure in batch and are stored in a specialized graph database. The commonalities existing between different graphs are extracted from the database to form a graph-based index. The authors concluded that the graph-based approach provides a better understanding of big data and provides less overhead of querying it by identifying the common traits between data points. Microsoft engineers proposed a distributed in-memory graph database called A1 to power their Bing search engine [62]. A1 is used to handle complex queries over structured data by using high-speed Remote Direct Memory Access (RDMA). The authors of A1 concluded that a graph model is easy to use and forms a robust tool for building search applications like Bing.

The authors of [63] proposed a Graph-based Similarity Search (GSS) method for a large speech data set. In this method, the graph structure for each complete speech

data from a large pile of speech data is pre-constructed. This graph index structure is used by the authors to find relevant complete speech data by only specifying the little number of phrases from the speech (called as query-by-example). The authors claim that GSS helps speed up the spoken term detection compared to the traditional method. The authors of [64] proposed a graph-based indexing scheme for Resource Description Framework (RDF) data. RDF is the standard for describing computing resources in a network. In the proposed method, initially, RDF data is converted into a data graph. Then, sub-graphs are extracted from this data graph and are indexed. The search query is converted into a graph and is compared with the sub-graph to retrieve the data. This setup reduces the query execution time on complete RDF data.

The authors of [65] proposed an indexing scheme for web documents. The indexing scheme is graph-based and is called as Document Index Graph (DIG). Unique phrases occurring in a document are identified. These phrases are used as a node in the graph structure. A graph structure for each such documents is generated. These generated graph structures are compared to identify the significant matching phrases between the documents. Similar to DIG, the authors of [66] used a graph data structure to classify web documents based on content. The paper proposed an enhanced k-Nearest Neighbor classification algorithm to work with a graph data structure. The authors claim that classification accuracy increases when a graph structure is used in comparison to a standard vector model approach. The authors of [67] used a graph-based approach for pattern matching in crime data. Different entities found in the crime documents are captured in a graph structure, and the relations between

them are identified using a graph-based clustering technique. The authors concluded that a graph-based clustering approach helps identify and analyze the crime patterns efficiently.

A graph data structure called a Directed Acyclic Word Graph (DAWG) has found its application in the field of string indexing. In this graph structure, closely related strings are stored together. This leads to the effective performance of string pattern matching and prefix search [68]. In this graph structure, the unique character in a set of related strings forms a node of DAWG. DAWG is also referred to as suffix automaton. DAWG is a data structure that permits extremely fast word searches. The entry point into the graph represents the starting character of the word to be searched. Then, successive characters in the word are used to navigate through the graph to complete the search. If any of the characters in the word is not found as a node while navigating through the graph, then the search is terminated as there is no possibility of finding the word in the graph.

2.4.2.2 Keyword Extraction

The authors of [69] proposed a graph-based keyword extraction algorithm called KeyGraph. A separate graph for each sentence in a document is made. The nodes represent terms and links represent the connection between terms that frequently occurs in the sentence. Each such graph is divided into separate clusters. Each cluster corresponds to a concept on which the author's main idea is reflected. The terms in each cluster are ranked and the top ones are selected as keywords for indexing. KeyGraph

is a content-sensitive, domain-independent algorithm. The authors of [70] proposed a graph-based text processing method called TextRank. TextRank models the abstract of a document as a graph after applying lexing and stop word removal process. The similarities between words are then calculated and stored in a matrix. The similarity matrix is then converted into a graph, with words as vertices and similarity scores as edges, for term rank calculation. Finally, a certain number of top-ranked words are made as keywords. TextRank is claimed to perform better than a supervised machine language approach.

The authors of [71] proposed a keyword extraction based on graph structure for micro-blogs. After micro-blogs are subjected to pre-processing, the remaining words are made as nodes of a graph. The words are selected as keywords from the graph based on the measures such as closeness and eccentricity of each node to its neighbouring nodes. Another technique called Keyword Extraction from the collection of Tweets using Node and Edge Weight (KETNEW) was proposed in [72]. Similar to the previous method [71], KETNEW also uses the words found in a tweet as nodes of the graph structure with edges connecting consecutive words. In this method, initially, node and edge weights are calculated based on factors such as term frequency, co-occurrence frequency, and the position of the node. Then, a node importance score is calculated based on the node and edge weights. All the nodes are ranked based on node importance score, and the top K nodes are selected as the final list of keywords. The authors of [73] proposed a graph-based keyword extraction technique for research documents. In this technique, the Rapid Automatic Keyword Extraction (RAKE) algorithm and a graph-based model called Keyword Extraction using Collective Node

Weight (KECNW) are used together for extracting keywords. Initially, RAKE is used to produce a candidate keyword set by eliminating all the stop words in the documents. Then, a graph structure is constructed using the candidate keywords as nodes with edges between co-occurring keywords. Then, the node weights are calculated based on factors such as term frequency. Finally, the candidate keyword nodes are ranked based on their weight, and the top K nodes are selected as keywords.

From the literature review presented above, it is clear that graph structures present several advantages when used to represent data that exhibit common traits (relationship). Textual document indexing can be viewed as a relation between a set of keywords and the documents they appear in. The research works inspired us to use a graph structure to capture the relationship between different keywords that co-occur in a document. The proposed method treats different keywords that appear in a document as entities that are to be related by a common trait called document id. This results in forming a graph structure where unique keywords form the nodes and the document ids are used as a label in the edge connecting the nodes. The result of the Boolean queries expresses how different keywords are connected to the set of documents indexed. By successfully forming the graph structure, we were able to model the results for the Boolean queries efficiently as illustrated in Chapter 3.

2.5 Concluding Remarks

The literature works described in this chapter highlight the importance of using a graph data structure for expressing data entities that have commonality between them. The importance of hash tables in search performance enhancement is emphasized in a number of research works discussed. With the importance of graph and hash table data structures being studied, the proposed method aims to combine these two data structures to produce a novel text-based indexing scheme to execute Boolean, neighbours of a keyword, and exclusive keyword queries efficiently.

Chapter 3

Graph-Based Indexing Technique

This chapter explains the proposed Graph-Based Index structure and the algorithm for indexing in a Graph-Based Index. It also covers the Boolean, neighbours of a keyword and exclusive keyword queries execution using the Graph-Based Index.

3.1 Graph-Based Index Structure

A Graph-Based Index consists of a directed graph where each unique keyword extracted from the document to be indexed forms a node and an edge is added between any two nodes if the two keywords exist in the same document. A directed graph structure is used to build GBI because it uses a smaller number of entries to denote a relationship. For example, let *Anthony* and *Brutus* be two keywords extracted from a document. A single entry *Anthony-Brutus* in the adjacency matrix is enough to denote this relationship (no need to store *Brutus-Anthony* in the adjacency matrix as

this would be the case with an undirected graph). To facilitate this, extracted keywords from a document are sorted in alphabetical order, and the relationship between two keywords is represented in ascending order. Two nodes are said to be neighbours if they have a direct edge between them. Throughout this thesis, the usage of connected or related keywords refers to neighbours meaning the keywords in question have occurred in a document together. The ids of the documents that contain both keywords are used as labels of the edges. Loops are allowed in the graph where the starting and ending nodes are the same. This happens when the keyword extraction process on a document results in only one keyword. This means that there is only one unique word from the document to index. A slightly modified version of the adjacency matrix representation is used to represent GBI. In the proposed method, the adjacency matrix will store only the keywords (nodes) that have occurrences together in a document. This allows GBI to consume less memory. This is opposed to the usual adjacency matrix representation of the graph depicted in Figure 2.5 where the nodes which do not have an edge between them are also stored. A two-dimensional hash table is used to store this modified version of the adjacency matrix. In a two-dimensional hash table, two keys are used to uniquely identify a set of values. The keywords that occur together in a document are grouped into a pair of keywords and are used as the key for the hash table entry containing ids of the documents as value.

3.2 Algorithm for Indexing Using GBI

Algorithm 1 exhibits the construction of the Graph-Based Index for indexing one document. The algorithm gets a list of keywords for a particular document to index and forms a link between those keywords resulting in a graph structure. The keyword extraction process on a document should result in at least one keyword to index making the document eligible for indexing. The inputs are the set of keywords (SI) and the identifier for the document (Id) from which the keywords are extracted. The output is the Graph-Based Index structure in the form of a hash table (G). A new GBI is created when the first document is indexed and gets updated when more documents are indexed. Indexing involves creating a series of entries in a two-dimensional hash table G. The value of all the entries is always the id of the document that is currently being indexed. Since it is a two-dimensional hash table, two keys are used to uniquely identify an entry. The two keys are labelled as $key1$ and $key2$. Initially, the keywords in SI are sorted in ascending order and the size of SI is determined [Line 1-2].

If SI has only one keyword to index (i.e. the size of SI is one), then that keyword is considered as $key1$. The number zero is appended to that keyword to form $key2$. An entry in G is created with these two keys [Line 3-7]. This entry signifies the document ids in the self-loop of the keyword node. This is the only keyword extracted to uniquely represent the document in GBI.

If SI has more than one keyword to index, then for each keyword in SI, an entry is created in G with the keyword as both $key1$ and $key2$ [Line 8-11]. This entry

Algorithm 1: Construction of Graph-Based Index

Inputs: Set of keywords (SI) and identifier of the document (Id)
Output: New or updated hash table representing Graph-Based Index (G)
Condition: At least one keyword exists in set SI

```
1  Sort set SI in ascending order
2  count ← size of set SI
3  if count == 1 then
       /* Occurs when the document has a single keyword to index.  It results
          in forming a loop on the keyword node */
4      key1 ← single keyword found in SI
5      key2 ← append zero (0) to key1
6      Key ← hash table key [key1][key2]
7      If an entry identified by Key exists in G, append Id to the list of values,
        otherwise create an entry using Key with Id as value
8  else
       /* Occurs when the document has more than one keyword to index */
9      foreach keyword key1 in set SI do
10         Key ← hash table key [key1][key1]
11         If an entry identified by Key exists in G, append Id to the list of
            values, otherwise create an entry using Key with Id as value
12         foreach keyword key2 occurring after key1 in set SI do
13             Key ← hash table key [key1][key2]
14             If an entry identified by Key exists in G, append Id to the list of
                values, otherwise create an entry using Key with Id as value
15         end
16     end
17 end
18 Sort all the entries of G by their key1 in ascending order
```

signifies the collection of document ids in the keyword node of GBI. After this entry, a series of entries are created in G. Each entry contains the current keyword as $key1$ and the keyword occurring next to the current keyword in SI as $key2$ [Line 12-17]. This procedure is repeated for all the keywords until it reaches the last keyword in SI. Since all the necessary information about the last keyword can be found in preceding keywords hash table entries, the indexing process is complete now. These series of entries result in creating edges between the keyword nodes with document ids as labels. This implies the relationship between different keywords extracted from the same document.

Finally, the entries in G are sorted by their $key1$ in ascending order [Line 18]. This final sorting helps in executing the neighbours of a keyword queries efficiently.

3.3 GBI Example

In this section, we present a detailed example that we will use throughout the thesis to explain the various queries that are being executed using GBI. Consider the same example as mentioned in Section 2.1.1. The keyword *Anthony* appearing in documents *D1, D2* and *D5*, the keyword *Caesar* appearing in documents *D1, D2, D3 and D4* and the keyword *Brutus* appearing in documents *D1* and *D4*. The Venn diagram representing this example is shown in Figure 3.1.

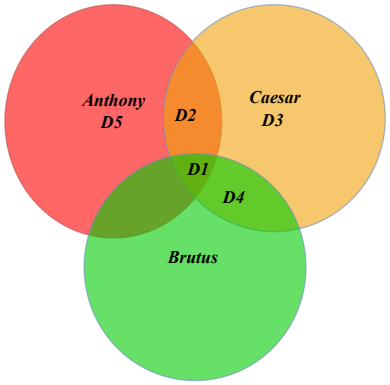


Figure 3.1: Venn diagram representing the keywords and the documents that they appear in.

Each document is processed one by one to build the GBI. For example, consider processing document *D1*. The keywords that appears in document *D1* are *Anthony*, *Caesar*, and *Brutus*. First, the keywords are sorted in ascending order. Thus, the sorted keywords set becomes {*Anthony*, *Brutus*, *Caesar*}. Each keyword in the set represents a node in the GBI. Then, an edge containing the document id as a label is added from each keyword to the remaining set of keywords that appear after it in the sorted set. This process is repeated until the last keyword in the sorted set is reached. The unique document ids found on the edges connecting the neighbour nodes alone are collected and stored in the respective nodes. This additional storage of document ids in the node is needed to effectively process the Boolean OR and Boolean NOT queries as described later in sections 3.5.3.1 and 3.5.2.1 respectively. This setup also helps in processing simple queries containing only a single keyword effectively. In some cases, for example in document *D5*, only one keyword (*Anthony*) is extracted. In order to index document *D5*, a loop with the document id *D5* as label

is constructed on the node *Anthony*. This procedure is repeated for all the documents in the given example to produce the final graph structure shown in Figure 3.2.

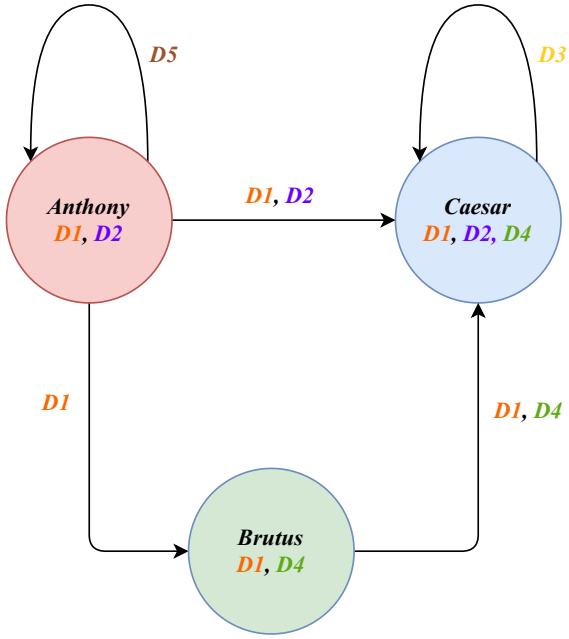


Figure 3.2: GBI - graph representation.

Figure 3.3 shows the hash table structure capturing keyword relationships shown in the graph representation. To describe the hash table structure, consider *Anthony* keyword's mappings. In the mapping, there are four key-value entries represented by the keys *Anthony*, *Anthony0*, *Brutus* and *Caesar* respectively. The key *Anthony* represents the node *Anthony* in the graph structure. The corresponding values are the document ids stored in the node. This entry represents all the documents in which the keyword *Anthony* co-occurs with other keywords. The second entry, pointed by the key *Anthony0*, represents the loop of the node *Anthony* in the graph structure. The value represents the document ids stored in the loop of the node *Anthony*. This entry

represents all the documents in which the keyword *Anthony* alone appeared. The third entry, pointed by the key *Brutus*, represents an edge between the node *Anthony* and *Brutus* in the graph structure. The values are the labels (document ids) of the edge. Thus, this entry represents all the documents in which *Anthony* and *Brutus* have appeared together. The fourth and the final entry, pointed by the key *Caesar*, represents an edge between the node *Anthony* and *Caesar* in the graph structure. Similarly, the values are the labels (document ids) of the edge. Thus, it represents all the documents in which *Anthony* and *Caesar* have appeared together. A search for the document ids that are common to "*Anthony* and *Caesar*" will be executed as follows. First, the key *Anthony* is hashed to locate *Anthony*'s mapping and then, the key *Caesar* is hashed to find the entry pointed by *Caesar* within *Anthony*'s mapping to find the values (document ids) represented by this keyword pair.

It should be noted that the document ids found in the node and self-loop of *Anthony* (when merged together) form the postings list of *Anthony* in the classical Inverted Index. This infers that GBI models Inverted Index within itself. This setup helps to apply various optimizations and algorithms used in Inverted Index on GBI as well.

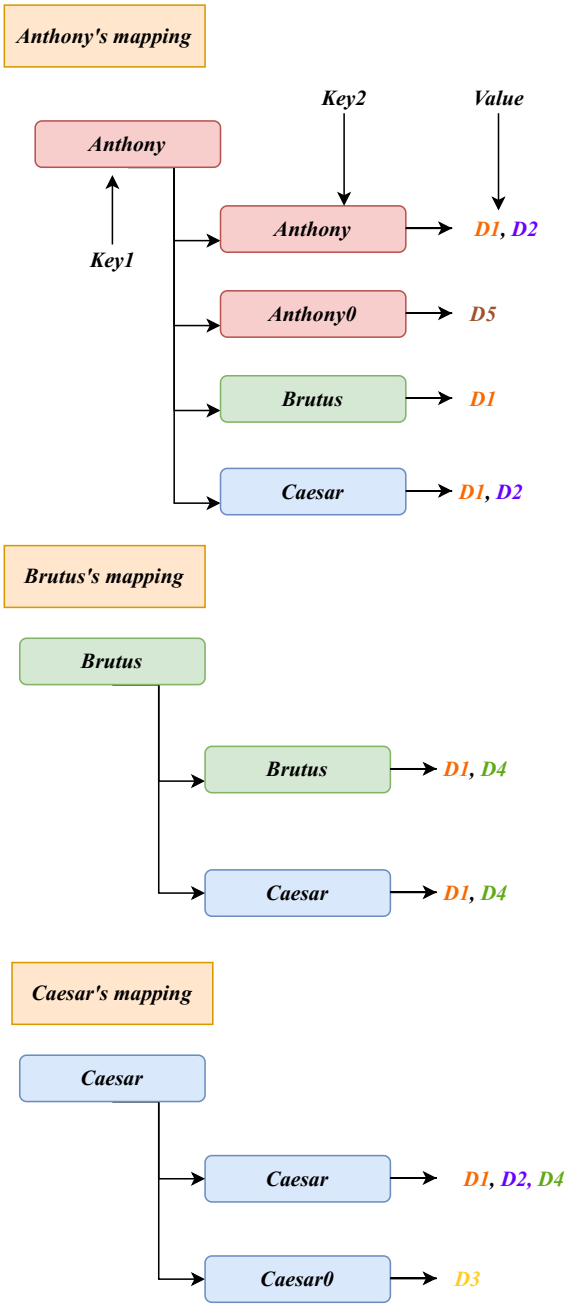


Figure 3.3: GBI - hash table representation.

3.4 Pruned Graph-Based Index to Process Only Boolean AND Queries

If we know that only Boolean AND queries will be processed, then some improvements can be made to the Graph-Based Index. In fact, certain information stored in the Graph-Based Index can be removed to improve the indexing time and memory consumption. For example, document ids stored in the keyword nodes can be removed. Since Boolean AND queries always examine the labels on the edges connecting the keyword nodes in the graph structure, the ids of the documents stored in the nodes are of no use and can be ignored. Also, the documents which result in only one keyword after the keyword extraction process can be eliminated from indexing thus eliminating self-loops in the graph structure. Since Boolean AND queries need at least two keywords to execute and that the document ids found in the self-loops are associated with only one keyword, loops will not contribute to any Boolean AND query execution. Figure 3.4 and Figure 3.5 show the modified GBI graph and the hash table representation for executing only Boolean AND queries for the same example stated above.

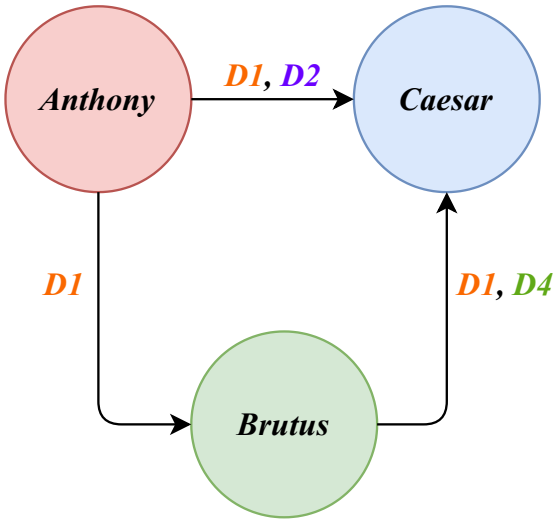


Figure 3.4: Pruned GBI - graph representation.

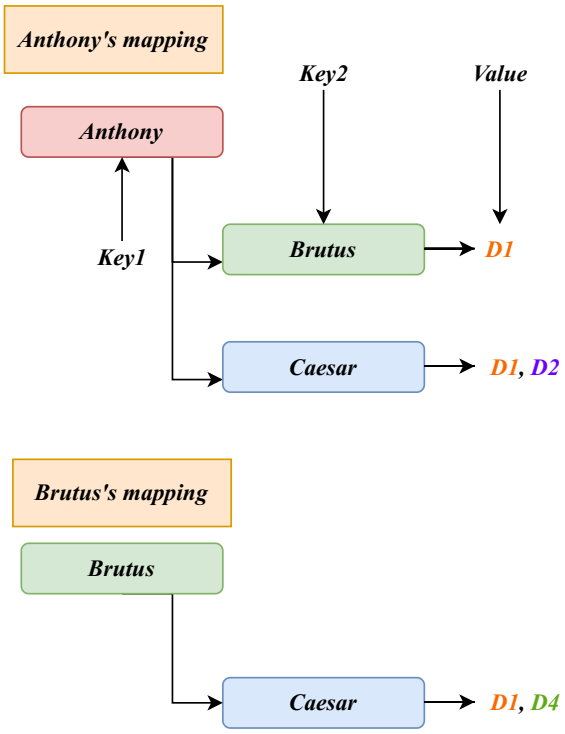


Figure 3.5: Pruned GBI - hash table representation.

Figure 3.4 shows that the document ids are no longer collected in the node. Documents 3 and 5, which were represented with a self-loop in Figure 3.2, are also eliminated from indexing. In the hash table representation, the entries representing ids collected in the node and loops are also removed. It can be noted that *Caesar*'s hash table mappings are removed completely. Since *Caesar* is alphabetically the last keyword in the given example and keywords are always sorted in ascending order for indexing and searching, *Caesar*'s relationship with *Anthony* and *Brutus* can be found in the *Anthony* and *Brutus* hash table mapping respectively. This setup for processing only Boolean AND queries eliminates the need for hash table mapping for the alphabetically last keyword among the set of keywords extracted from the documents to be indexed. This results in improved memory consumption and indexing time for GBI.

Algorithm 2 exhibits the construction of the Pruned Graph-Based Index targeted for only Boolean AND queries. It is similar to the normal version of GBI (i.e. Algorithm 1) except for storing document ids in keyword node and eliminating single keyword documents from indexing. The inputs are a set of keywords (SI) and the identifier for the document (Id) from which the keywords are extracted. The output is the Pruned Graph-Based Index structure in the form of a hash table (G). A new Pruned GBI is created when the first document is indexed and gets updated when more documents are indexed. The condition is that at least two keywords should represent a document otherwise it will be ignored from indexing. Initially, the keywords in SI are sorted in ascending order [Line 1]. Then, for each keyword in SI until the last keyword, an hash table entry is created in G. Each entry is identified

by two keys. The current keyword is used as $key1$ and the keyword occurring next to the current keyword in SI is used as $key2$ [Line 2-7]. This results in creating edges between the keyword nodes with document ids as labels. This represents the relationship between the different keywords extracted from the same document. If the algorithm is processing the last keyword in set SI, then the indexing process is complete. As mentioned earlier, this is because all the necessary information about the last keyword can be found in preceding keywords hash table entries so there is no need of creating a separate entry for the last keyword in the set. This helps in reducing the required amount of memory space for storing the graph-based index.

Algorithm 2: Construction of Pruned Graph-Based Index

Inputs: Set of keywords (SI) and identifier of document (Id)
Output: New or updated hash table representing Pruned Graph-Based Index (G)
Condition: At least two keywords exist in set SI

```
1 Sort set SI in ascending order
2 foreach keyword key1 until the last keyword in set SI do
    /* Establishing relationship between keywords extracted from a document
       */
3   foreach keyword key2 occurring after key1 in set SI do
4     Key ← hash table key [key1][key2]
5     If an entry identified by Key exists in G, append Id to the list of
        values, otherwise create an entry using Key with Id as value
6   end
7 end
```

Since the document ids found in the edges between the keyword nodes are used to perform the intersection operations in both normal and Pruned GBI, the search performance obtained by executing Boolean AND queries remains the same for both

versions of GBI. The pruning only reduces the indexing time and the required amount of memory. Hence, the normal GBI version will be used for all the search algorithms listed in this chapter.

3.5 Boolean Queries and Search Algorithms Using Graph-Based Index

This section describes the different types of Boolean queries and the various algorithms that were developed for searching in the proposed Graph-Based Index.

3.5.1 Boolean AND Queries

The Boolean AND queries are a type of queries where all the search keywords specified should be found in a document to qualify it to be a part of the resultant set. Even if the document misses one search keyword, it is discarded. In other words, let:

- $S = \{S_1, S_2, ..., S_n\}$ be the set of search keywords.
- R be the resultant set containing ids of the matching documents.
- D be the document that is currently being examined.

Then, D is qualified to be part of R if and only if all the search keywords from the set S (i.e. $S_1, S_2, ..., S_n$) are present in document D. It is important to note

that the position of the search keywords within the document is irrelevant. The only condition is that all the search keywords should present in the document to be part of the resultant set.

Thus, for the example stated in Section 3.3, the Boolean AND query "*Anthony* AND *Caesar* AND *Brutus*" yields the result *D1* since *D1* is the only document that contains all the three keywords.

3.5.1.1 Execution of Boolean AND Queries Using Graph-Based Index

Algorithm 3 exhibits the execution of a Boolean AND query using the Graph-Based Index. In general, let S be the set containing the search keywords. Initially, the set S is sorted in ascending order and the size of S is determined [Line 1-3]. Each keyword except the last keyword in set S is paired with the next occurring keywords to find whether there exists an edge between them. The document ids are collected only if a keyword pair is non-overlapping or it is the last keyword pair in S [Line 5-11]. This is to reduce the number of keyword pairs needed to compute the intersection between them. To illustrate this, consider four keywords *A, B, C* and *D*. Since the keywords are indexed in pairs (i.e. *AB, AC, AD, BC, BD*, and *CD*), only non-overlapping keyword pairs are needed to compute the intersection. The non-overlapping keyword pairs contain all the search keywords only once (in this case AB and CD for example). Therefore, only the document ids of these two pairs are collected. Now, let us consider five search keywords say *A, B, C, D* and *E*. The number of pairs from which the document ids will be collected is *AB, CD* and also *DE*. As there is the need for one

Algorithm 3: Executing a Boolean AND query search using Graph-Based Index

Inputs: Set of search keywords (S), hash table representing Graph-Based Index (G)
Output: NULL or the matching document ids.
Condition: At least two unique keywords exists in set S

```
1  Sort set S in ascending order
2  postings_list_collection ← an empty hash table
3  search_count ← size of set S
4  foreach search keyword key1 in set S do
5      if key1 is not the last entry in S then
           /* Check if all the search keywords are connected to each other in
              GBI */
6          foreach search keyword key2 occurring next to key1 in set S do
7              Key ← hash table key [key1][key2]
8              if Key exists in hash table G then
9                  if Key is non-overlapping or last keyword pair in S then
                       /* Collect document ids from the keyword pair's postings
                          list only if it is a non-overlapping or last keyword
                          pair in S to reduce the number of postings list needed
                          for intersection */
10                     create entry in postings_list_collection with key1 as key
                         and postings list at G[Key] as value
11                 end
12             else
                   /* If any two of all the search keyword is not connected then
                      terminate keyword relationship checking.  No possibility of
                      common documents.  Terminate search process */
13                 postings_list_collection ← NULL
14                 return NULL
15             end
16         end
17     end
18 end
19 Algorithm 3 continued in next page
```

Algorithm 3: Executing a Boolean AND query search using Graph-Based Index (continued)

```
20 if postings_list_collection is not NULL then
       /* If all the search keywords are connected to each other then common
          documents are possible */
21     if search_count == 2 then
           /* If there are only two search keywords then return keyword pair's
              postings list directly as it is already available */
22         return postings_list_collection
23     else
           /* If more than two search keywords then perform intersection
              operation between keyword pairs postings lists */
24         output ← Perform Small-Vs-Small (SvS) intersection algorithm on
            postings_list_collection as described in Algorithm 4
25         return output
26     end
27 end
```

more pair to include keyword E in this case, the document ids are also collected from the last pair which is DE. If a keyword pair does not exist in the Graph-Based Index, then the search procedure stops as there is no possibility of a common document, and NULL is returned [Line 12-18]. If all the keywords are connected, then one solution to get all the matching document ids is performing the intersection between the keyword pairs. The authors of [75] studied various intersection algorithms based on how the data is stored. The study concluded that when data is stored as a sorted array of explicit values (storing the values as it is without any compression techniques), the traditional Small-Vs-Small (SvS) algorithm is one of the best way to perform an intersection. If there are only two search keywords, then the result is directly obtained as it is already indexed in pairs [Line 20-22]. The SvS algorithm is needed

Algorithm 4: Small-Vs-Small Intersection Algorithm [74]

Input: A hash table containing a set of postings lists (P)
Output: NULL or an array containing common document ids among the postings lists
Condition: At least two postings lists should be provided for performing intersection

```
1  Sort P by the size of postings lists in ascending order
2  output ← first postings list in P
3  remove first postings list from P
4  foreach remaining postings list PL in P do
5  |   foreach document_id in output do
   |   |   /* remove document ids from the smallest postings list if it does not
   |   |      exists in other postings lists */
6  |   |   if document_id does not exist in PL then
7  |   |   |   remove the document_id from output array
8  |   |   end
9  |   end
10 end
11 return output
```

only if there are more than two search keywords [Line 23-27]. If all the keywords are connected, then the number of intersections to be made using SvS depends on the number of keywords. In general, if there are n search keywords, and $n > 2$ and n is even, then only *n/2 - 1* intersections are needed. If $n > 2$ and n is odd, then only $\lfloor n/2 \rfloor$ intersections are needed. If $n = 2$, then no intersection operation is needed.

For completeness, Algorithm 4 presents the SvS intersection algorithm studied by the authors of [74]. The SvS algorithm takes a set of postings list (P) and performs an intersection to return the common document ids between them. The inputs are the set of postings lists for which intersection should be computed. The output is either NULL meaning no common documents are found or the set of common document ids between the postings lists. The condition is that at least two postings lists should be provided. Initially, P is sorted in ascending order by the size of each postings list [Line 1]. The smallest postings list is chosen as a candidate solution initially as this contains the minimum possible number of document ids that are common between the given postings lists [Line 2-3]. The document ids in the smallest postings list are checked for existence in the remaining postings lists in P. If any of the document ids in the smallest postings list are not found in the remaining postings lists, then those document ids are removed from the smallest postings list [4-10]. After this checking process, if any document ids are left untouched, then they are returned otherwise NULL is returned [Line 11].

For the Inverted Index prototype that was developed in this thesis, the same SvS algorithm is used for intersection computation. The results show that GBI always

performs better than Inverted Index. Since it implements SvS between keyword pair(s) whereas Inverted Index implements it between individual keywords in the search keywords set. Thus, the number of postings list needed for intersection is always smaller in GBI when compared to Inverted Index. Also, it is important to note that in Graph-Based Index, there is no need to execute the SvS algorithm when there is no relationship between any of the search keywords. However, in Inverted Index, at least one intersection should be performed to validate that no common document exists between the search keywords. In general, for Inverted Index, the number of intersections performed using the SvS algorithm for executing a Boolean AND search query is *n-1* for n keywords ($n > 1$) and all n keywords appear together in at least one document. Thus, irrespective of the intersection algorithm that is used, GBI will always perform better than Inverted Index as GBI always uses a smaller number of postings lists and also tries to avoid intersection.

The Boolean AND algorithm (i.e. Algorithm 3) execution sequence for the example stated in Section 3.3 for the query "*Anthony* AND *Caesar* AND *Brutus*" is discussed next. According to the algorithm, the input set S now contains {*Anthony*, *Caesar*, *Brutus*}. The first step is to sort the set S in ascending order which results in {*Anthony*, *Brutus*, *Caesar*}. Then, the second step is to check whether *Anthony* is related to *Caesar* and *Brutus* and whether *Brutus* is related to *Caesar*. From Figure 3.3, it is inferred that *Anthony*'s mapping has an entry for *Caesar* and *Brutus*. *Brutus* mapping also has an entry for *Caesar*. Thus, the keywords *Anthony*, *Brutus* and *Caesar* are related to each other. Then, the postings list for the pair (*Anthony*, *Brutus*) is retrieved with a value of *D1*. Next, the postings list of the pair (*Brutus*,

Caesar) is also retrieved with values *D1* and *D4*. Finally, the collected postings lists are sorted in ascending order by their size and it is found that *Anthony* and *Brutus* pair's postings list is the smallest one. The values in the smallest postings list are iterated one by one to check whether they exists in all other postings lists. Since *D1* exists in *Brutus* and *Caesar*'s postings list, it is not removed from the smallest postings list. Since there are no other document ids in the smallest postings list, the iteration is complete and the final output (*D1*) is returned.

3.5.2 Boolean NOT Queries

The Boolean NOT queries are a type of queries where all the search keywords specified after the NOT operator should not appear in the same document as the keyword specified before the NOT operator to be a part of the resultant set. More formally, let:

- S_k be the keyword specified before the NOT operator.
- $E = \{E_1, E_2, \ldots, E_k\}$ be the set of search keywords specified after the NOT operator.
- R is the resultant set containing ids of the matching documents.
- D be the document that is currently being examined.

Then, D is qualified to be part of R if and only if S_k appears in the document D and the keywords from the set E (i.e. $E_1, E_2, \ldots, E_k$) does not appear in the document

D. The important thing to note is that the position in which the search keywords appear within the document is irrelevant.

Thus, for the example stated in Section 3.3, the Boolean NOT query "*Caesar* NOT (*Anthony* OR *Brutus*)" yields the result *D3* since it is the only document that contains *Caesar* but not *Anthony* or *Brutus*.

3.5.2.1 Execution of Boolean NOT Queries Using Graph-Based Index

Algorithm 5 exhibits the execution of a Boolean NOT query in GBI. The inputs are: 1) the keyword (S_k) to be present in a document; 2) set of keywords (E) that should not be present in the document with S_k; and 3) the hash table representing Graph-Based Index (G). The condition is that at least one keyword should exist in E otherwise there will be no keyword to exclude. Initially, the S_k is checked for its existence in G. If it does not exist, then the algorithm terminates by returning NULL. If S_k exists in G, then the *output* array is initialized with the postings list found at $G[S_k][S_k]$ [Line 4-8]. This collects document ids from the S_k node in GBI. Then, the number zero (0) is appended to the S_k to form a key ($exclusive\_key$) to identify the entry containing the exclusive document ids from the S_k mapping. If there exists an entry in G pointed by the keys $[S_k][exclusive\_key]$, then the document ids at that entry are stored in the $exclusive\_documents$ array [Line 10-12]. After these initial assignments, a neighbour validation procedure is performed [Line 13-22].

A neighbour validation procedure checks whether the keywords in E are actually the neighbours of the S_k. To perform this validation, each keyword ($key2$) in E is

Algorithm 5: Executing a Boolean NOT query search using Graph-Based Index

Inputs: A keyword (S_k) to be present in a document, set of keywords (E) to be excluded, hash table representing Graph-Based Index (G)
Output: NULL or an array containing the matching document ids
Condition: At least one keywords exists in set E

```
1  neighbours ← empty hash table
2  exclusive_documents ← empty array
3  output ← empty array
4  if G[S_k] does not exist then
       /* If the S_k does not exist in GBI then terminate search */
5      return NULL
6  else
       /* Collect document ids from the S_k's node */
7      output ← postings list at G[S_k][S_k]
8  end
9  exclusive_key ← append zero (0) to S_k
10 if G[S_k][exclusive_key] exists then
       /* Collect any exclusive documents available for S_k from its loop */
11     exclusive_documents ← postings list at G[S_k][exclusive_key]
12 end
13 foreach keyword key2 in set S do
       /* Verify each of the keywords (key2) in E are neighbours of S_k and if
          neighbours, collect common document ids between them */
14     if key2 is alphabetically after S_k then
15         Key ← hash table key [S_k][key2]
16     else
17         Key ← hash table key [key2][S_k]
18     end
19     if G[Key] exists then
20         create entry in neighbours with key2 as key and postings list at
             G[Key] as value
21     end
22 end
```

23 Algorithm 5 continued on next page

Algorithm 5: Executing a Boolean NOT query search using Graph-Based Index (continued)

if *neighbours* is empty **then**
    /* If none of the keywords in E are neighbours of S_k then no need to remove any documents. Return document ids collected from S_k's node and loop */
    add *exclusive_documents* (if exists any documents) to *output* array
    return *output*
end
all_neighbours ← find all the neighbours of S_k using Algorithm 7
if *all_neighbours* is same as E **then**
    /* If all the neighbours of S_k are same as keywords in E then possible result is only *exclusive_documents* */
    return *exclusive_documents*
end
foreach postings list PL in *neighbours* **do**
    /* Remove common document ids between S_k and its neighbours */
    remove all the document ids found in PL from *output* array
end
add *exclusive_documents* to *output* array
return *output*

categorized based on whether it occurs before or after the S_k in G. If $key2$ appears after the S_k in G, then the existence of $key2$ is checked in the S_k's mapping. If found, an entry is created in the *neighbours* hash table with $key2$ as key and postings list at $G[S_k][Key2]$ as value. If $key2$ appears before the S_k in G, then the existence of the S_k is checked in $key2$'s mapping. If found, an entry is created in the *neighbours* hash table with $key2$ as key and postings list at $G[Key2][S_k]$ as value. This is because before keywords extracted from a document are indexed in GBI, they are alphabetically sorted. To better illustrate this, consider an example where the set E contains the keywords *Anthony* and *Caesar* and the S_k is *Brutus*. Now, to check whether *Brutus* is related to *Anthony* or not, the existence of keyword *Brutus* is checked in *Anthony*'s mapping because *Brutus* is alphabetically after *Anthony*. But in order to check whether *Brutus* is related to *Caesar* or not, the existence of keyword *Caesar* is checked in *Brutus*'s mapping because *Caesar* is alphabetically after *Brutus*. Figure 3.3 hash table representation gives a better visual about this neighbour checking process.

Once the neighbour validation process is complete, the *neighbours* hash table holds the information about the keywords in E that are related to the S_k with their corresponding postings list. If the *neighbours* hash table is empty, then the ids of the documents found in the *exclusive_documents* array are added to *output* array and are returned terminating the search operation [Line 24-27]. This means that the documents represented by their ids stored in the *output* array are free of all the keywords in E. If the *neighbours* hash table is not empty, then all the possible neighbours of the S_k are found using Algorithm 7 which is discussed later in Section 3.6.1 and are

stored in the *all_neighbours* array [Line 28]. If the *all_neighbours* array is the same as E, then document ids stored in *exclusive_documents* array are returned, and the search is terminated [Line 29-31]. This is because documents that contain only the S_k can be part of the solution as all the remaining documents contain the keywords which are to be excluded.

If the *all_neighbours* array is not the same as E, then all the document ids in all the postings lists of the *neighbours* hash table are removed from the *output* array, adding any documents in which the search keyword alone appears (if any). Finally, the document ids that are free from the keywords listed in E are returned [Line 32-36].

The Boolean NOT algorithm (i.e. Algorithm 5) execution sequence for the example stated in Section 3.3 for the query "*Caesar* NOT (*Anthony* OR *Brutus*)" is described next. According to the algorithm, the search keyword is *Caesar* and the set of keywords that are to be excluded contains *Anthony* and *Brutus*. The first step checks whether *Caesar* has any common documents with *Anthony* and *Brutus*. In other words, validating whether they are neighbours or not. From Figure 3.3, it is inferred that *Anthony*'s mapping has an entry for *Caesar* and *Brutus*'s mapping also has an entry for *Caesar*. Thus, both *Anthony* and *Brutus* are related to *Caesar*. Since all possible neighbours of *Caesar* are the same as the neighbours that are to be excluded, the Boolean NOT algorithm returns only the documents that contains *Caesar* alone which is *D3*.

For executing Boolean NOT queries using Inverted Index prototype developed, the postings lists of the keywords in the set E are compared against the postings list

of S_k and common document ids are removed. If there are any resultant document ids, then they are returned as output.

3.5.3 Boolean OR Queries

The Boolean OR queries are a type of queries where at least one of the search keywords specified should be found in a document to qualify it to be a part of the resultant set. In other words, let:

- $S = \{S_1, S_2, ..., S_n\}$ be the set of search keywords.
- R be the resultant set containing ids of the matching documents.
- D be the document that is currently being examined..

Then, D is qualified to be part of R if at least one of the search keywords from the set S (i.e. $S_1, S_2, ..., S_n$) is present in the document D. It is important to note that the position in which the search keywords appear within the document is irrelevant. If none of the search keywords is found within the document, then the document is discarded.

Thus, for the example stated in Section 3.3, the Boolean OR query "*Anthony* OR *Caesar* OR *Brutus*" results in *D1, D2, D3, D4 and D5*. It is because either *Anthony* or *Caesar* or *Brutus* is present in at least one of the document listed in the resultant set.

3.5.3.1 Execution of Boolean OR Queries Using Graph-Based Index

The Boolean OR search in GBI is modelled as a series of Boolean NOT search operations. This is because Boolean OR queries is a union operation, combining ids of the documents in which any of the search keywords occurs, removing duplicate document ids in the process. Therefore, in GBI, while processing each search keyword, the remaining search keywords are considered to be excluded. This results in faster duplicate removal while performing the union operation. This process of executing a Boolean OR query in GBI is exhibited in Algorithm 6. The inputs are the set of search keywords (S) and the hash table representing the Graph-Based Index (G). The condition is that at least two keywords should exist in S. The output is either NULL if none of the search keywords in S are found in GBI or the set of matching document ids. Initially, the Boolean OR algorithm sorts set S [Line 1]. Then, for each keyword ($key1$) in S, $key1$'s node is checked for any document ids. If it exists, then it means that $key1$ shares some documents with other keywords indexed. This signifies the potential for performing the Boolean NOT operation. The remaining keywords that occurs after $key1$ in S are collected in $keywords\_to\_exclude$. If $keywords\_to\_exclude$ is empty, then that means $key1$ is the last keyword in S. Since no keywords are there to exclude (after $key1$ in S), the document ids stored at the last keyword's node and in its loop (if exists) are stored in *output* array, and the search operation is terminated [Line 4-11]. The document ids in $key1$'s loop are found by appending the number zero (0) to the $key1$ to form a key ($exclusive\_key$) to identify the entry containing the exclusive document ids for $key1$. If there exists an entry in G pointed by the keys

Algorithm 6: Executing a Boolean OR query search using Graph-Based Index

Inputs: Set of search keywords (S), hash table representing the Graph-Based Index (G)
Output: NULL or the matching document ids.
Condition: At least two unique keywords exists in set S

```
1  Sort set S in ascending order
2  output ← an empty array
3  foreach keyword key1 in set S do
4      if G[key1][key1] exists then
5          keywords_to_exclude ← all the keywords occurring after key1 in S
6          if keywords_to_exclude is empty then
               /* If key1 is the last keyword in the set S, then no more
                  neighbours to remove, add the document ids found in its node
                  to the output array */
7              add the document ids found at G[key1][key1] to output array
8              exclusive_key ← append zero (0) to the key1
9              if G[key1][exclusive_key] exists then
                   /* add document ids found in key1's loop if exists to the
                      output array and terminate search */
10                 add documents ids found at G[key1][exclusive_key] to output
                    array
11             end
12         else
               /* Execute Boolean NOT algorithm to remove any duplicate
                  documents */
13             resultant_document_ids ← result of execution of Boolean NOT
                search algorithm as mentioned in Algorithm 5 with key1 as S_k
                and keywords_to_exclude as set of keywords to exclude (E)
14             if resultant_document_ids is not NULL then
15                 add resultant_document_ids to output array
16             end
17         end
18     end
19     Algorithm 6 from line 20 to 25 on next page goes here
26 end
27 return output
```

Algorithm 6: Executing a Boolean OR query search using Graph-Based Index (continued)

```
20 if G[key1][key1] does not exists then
       /* If key1 under consideration does not appear together with other
          keywords indexed in GBI. Boolean NOT operation is avoided.  Add any
          documents if available on key1's loop */
21     exclusive_key ← append zero (0) to the key1
22     if G[key1][exclusive_key] exists then
23         add documents ids found at G[key1][exclusive_key] to output array
24     end
25 end
```

$[key1][exclusive\_key]$, then that entry represents the document ids found in the loop of $key1$. If $keywords\_to\_exclude$ is not empty, then the Boolean NOT query algorithm (i.e. Algorithm 5) is executed. The inputs for the Boolean NOT query algorithm, in this case, will be as follows: the current keyword under consideration ($key1$) as the keyword to be found in the document and the remaining keywords that occur after the current keyword as a set of keywords to be excluded ($keywords\_to\_exclude$). If any document ids are left ($resultant\_document\_ids$) after the Boolean NOT operation, then they are stored in *output* array [Line 12-18]. This $resultant\_document\_ids$ signifies the documents that contain $key1$ but not any of the keywords occurring after $key1$ in S.

If the document ids in $key1$'s node are not present, then if document ids exist in the loop of $key1$, it is stored in the *output* array [Line 20-25]. This means that $key1$ does not share any documents with any other keywords indexed and therefore, the Boolean NOT operation can be avoided for this keyword. After iterating for all the keywords in S, the document ids stored in the *output* array are returned [Line 27].

The Boolean OR algorithm (i.e. Algorithm 6) execution sequence for the example presented in Section 3.3 for the query "*Anthony* OR *Caesar* OR *Brutus*" is discussed next. According to the algorithm, the input set of keywords is {*Anthony*, *Caesar*, *Brutus*}. The first step is to sort the input set in ascending order which gives {*Anthony*, *Brutus*, *Caesar*}. The first keyword in the sorted set *Anthony* is considered. *Anthony*'s node is checked for any document ids. Since document ids exists in *Anthony*'s node, possibility of *Anthony* sharing documents with other keywords is confirmed. Now as per the Boolean NOT query algorithm (i.e. Algorithm 5), the relationship between *Anthony* and the other remaining keywords occurring after *Anthony* (*Brutus* and *Caesar*) is determined. From Figure 3.3, it is inferred that *Anthony*'s mapping has an entry for both *Brutus* and *Caesar* meaning that both are related to *Anthony*. Since all the possible neighbours of *Anthony* are the same as the neighbours that are to be excluded, the documents containing only *Anthony* are stored in output array (in this case *D5*). Now, the second keyword in the sorted set *Brutus* is considered. *Brutus*'s node is checked for any document ids. Since document ids exists in *Brutus*'s node, the possibility of *Brutus* sharing documents with other keywords is confirmed. Now, as per the Boolean NOT query algorithm (i.e. Algorithm 5), the relationship between *Brutus* and the remaining keywords occurring after *Brutus* (*Caesar*) is determined. From Figure 3.3, it is inferred that *Brutus*'s mapping has an entry for *Caesar*. Thus, *Caesar* is related to *Brutus*. Since the number of neighbour to be eliminated (*Caesar*) is only half of the total number of neighbours (*Anthony* and *Caesar*), the document ids in *Brutus*-*Caesar*'s postings list are directly removed from the document ids stored at *Brutus*'s node giving the result

NULL. The result is NULL because all the documents in which *Brutus* appears also contain *Caesar*. Finally, the last keyword in the set (*Caesar*) is considered. Since there are no keywords that appear after *Caesar*, all the document ids found in *Caesar* node (i.e. *D1, D2, D4* and in its loop (i.e. *D3*) are stored in the output array and the search is terminated. The final output array containing document ids *D5, D1, D2, D4 and D3* is returned.

For executing Boolean OR queries using Inverted Index prototype developed, the postings lists of keywords from set S are sorted in ascending order by the postings lists size. Each smaller-sized postings lists are compared with larger-sized postings list and duplicate document ids are removed. Finally, the postings lists of the keywords from the set S with the duplicate document ids removed are returned as output.

3.6 Execution of Other Queries Using GBI

This section defines and describes the algorithms that were developed for other types of queries such as neighbours of a keyword and exclusive keyword queries that can be executed using GBI.

3.6.1 Neighbours of a Keyword Queries

The neighbours of a keyword queries are a type of queries where the resultant set contains a set of keywords that co-exist with the search keyword in any of the documents

indexed. Alternatively, let

- K_s be the search keyword.
- KI be the set of all the keywords found in the index structure except the search keyword K_s.
- K_i be one of the keywords from the set KI that is currently being examined.
- $R1$ be the resultant set containing keywords that occur together with the search keyword K_s

Then, K_i is qualified to be part of $R1$ if and only if K_i and K_s occur together in at least one document. If there is no single common document between the keyword under consideration and the search keyword, then that keyword is discarded.

Thus, for the example stated in Section 3.3, the "neighbours of a keyword query on the keyword *Brutus*" yields the result *Anthony* and *Caesar* because *Brutus* shares at least one document with *Caesar* and *Anthony*.

3.6.1.1 Execution of Neighbours of a Keyword Queries Using Graph-Based Index

Algorithm 7 exhibits the execution of neighbours of a keyword query in GBI. The inputs are the keyword (K_s) for which the neighbours should be found and the hash table representing GBI (G). The outputs are either NULL meaning that no neighbour

keywords exist or an array containing the keywords that share a common document with the K_s. Initially, the K_s is checked for its existence in G. If it does not exist, then the algorithm terminates by returning NULL [Line 2-4]. If the K_s exists in G, then for each keyword (*keyword*) in G until the K_s, the K_s is checked for its existence in the *keyword*'s mapping. If the K_s is found in the *keyword*'s mapping, then that *keyword* is added to the output array (*neighbours*). This step retrieves all the keywords that are related to the K_s that occurs before the K_s in GBI. [Line 5-9]. Finally, all the keywords that are found in the K_s's mapping are also added to the *neighbours* array and it is returned [Line 10-12]. This step retrieves all the keywords that are related to the K_s but occurs after the K_s in GBI.

A neighbours of a keyword query algorithm execution sequence for the example stated in Section 3.3 for the keyword *Brutus* is discussed next. Initially, *Brutus* is checked for its existence in GBI. Since the keyword *Brutus* exists in GBI (as shown in Figure 3.3), a search for the keyword *Brutus* in all the keyword's mapping starting from the first keyword indexed in GBI until the keyword *Brutus* is initiated. Since *Anthony* is the only keyword before *Brutus*, the keyword *Brutus* is checked in *Anthony*'s mapping and the keyword *Brutus* exists in it. Thus, the keyword *Anthony* is added to the output array. After this, all the keywords found in *Brutus*'s mapping (except the entries representing *Brutus*'s node and exclusive documents) are added to the output array. *Caesar* is the only keyword matching this condition and hence added to the output array. Thus, the final output array contains *Anthony* and *Caesar*. In other words, *Brutus* shares document *D1* with *Anthony* and *Caesar* and document *D4* with *Caesar*. These common occurrences makes *Anthony* and *Caesar*,

Algorithm 7: Executing a neighbours of a keyword query using Graph-Based Index

Inputs: A keyword (K_s) for which neighbours should be found, hash table representing Graph-Based Index (G)

Output: NULL or an array containing neighbours of a keyword

```
1  neighbours ← empty array
2  if G[K_s] does not exists then
       /* If the K_s does not exists in GBI return NULL */
3      return NULL
4  end
5  foreach keyword in G starting from the beginning until K_s do
       /* find all the keywords that are neighbour which appears alphabetically
          before the K_s */
6      if G[keyword][K_s] exists then
7          add keyword to neighbours array.
8      end
9  end
   /* Retrieve all the keywords that are neighbours which appears
      alphabetically after the K_s */
10 Keywords ← all the keys found in K_s's hash table mapping except the ones
     representing K_s's node and exclusive documents
11 add Keywords to neighbours array
12 return neighbours
```

the neighbours of *Brutus*.

For executing neighbours of a keyword queries using Inverted Index prototype developed, the postings lists of all the keywords indexed except for the search keyword are compared with the postings list of the search keyword (K_s). If any common document id is found, then that keyword is added to the neighbours list. Finally, the output array containing the neighbours list is returned as output.

3.6.2 Exclusive Keyword Queries

Exclusive keyword queries are a type of queries where the resultant set contains the documents in which only the search keyword appears, and no other keywords indexed should co-exist with the search keyword. More formally, let:

- K_s be the search keyword.
- KI be the set of all the keywords found in the index structure except the search keyword K_s.
- R be the resultant set containing ids of the matching documents.
- D be the document that is currently being examined.

Then, D is qualified to be part of R if and only if the search keyword K_s is present in document D and no keywords from the set KI should be found in the document D.

These queries are different from the Boolean NOT queries. In the Boolean NOT query, only certain specified keywords should not appear with the search keyword in the document. However, in an exclusive keyword query, all the keywords found in the index should not co-exist with the search keyword in the document. To demonstrate the difference, consider the example stated in Section 3.3. The Boolean NOT query "*Caesar* NOT *Anthony*" yields *D3* and *D4*. The "exclusive *Caesar*" query yields only *D3*. It is because *D3* is the only document which contains the search keyword *Caesar* but not any of the other keywords indexed (*Brutus* or *Anthony*). The rest of the documents which contain *Caesar* also contain other keywords indexed. Thus, they are not eligible to be part of the resultant set. The same argument can be made for the "exclusive *Anthony*" query which yields *D5*.

3.6.2.1 Execution of Exclusive Keyword Queries Using Graph-Based Index

Algorithm 8 exhibits the execution of an exclusive keyword query using GBI. The inputs are the keyword (K_s) for which the exclusive documents should be found and the hash table representing GBI (G). The outputs are either NULL meaning that no exclusive document exists for the K_s or an array containing the exclusive documents for the K_s. Initially, the K_s is checked for its existence in G. If it does not exist, then the algorithm terminates by returning NULL [Line 2-4]. If it exists in G, then the number zero (0) is appended to the K_s to form a key ($key2$) that identifies the entry containing exclusive document ids in the K_s's mapping. If there exists an entry in

G pointed by the keys $[K_s][key2]$, then the document ids at that entry are returned [Line 5-8]; otherwise NULL is returned [9-11].

Algorithm 8: Executing an exclusive keyword query using Graph-Based Index

Inputs: A keyword (K_s) for which exclusive document ids should be found, hash table representing Graph-Based Index (G)
Output: NULL or an array containing matching document ids

```
1  output ← empty array
2  if G[K_s] does not exist then
       /* If the K_s does not exists in GBI return NULL */
3      return NULL
4  end
5  key2 ← append zero (0) to the K_s
6  if G[K_s][key2] exists then
       /* Collect document ids from the K_s's loop */
7      output ← document ids at G[K_s][key2]
8      return output
9  else
10     return NULL
11 end
```

For executing exclusive keyword queries using Inverted Index prototype developed, the postings lists of all the keywords indexed except the search keyword are compared with the postings list of the search keyword (K_s) and common document ids between them are removed. If there are any resultant document ids, then they are returned as output.

Chapter 4

Experiments and Performance Analysis

This chapter describes the implementation of the prototypes and the experiments conducted on the prototypes to demonstrate the performance of GBI in comparison to Inverted Index. The experiments are conducted for analyzing the search latency of Boolean and the other types of queries described in Chapter 3. In addition to the search latency, this chapter also describes the experiments for comparing the indexing time and memory usage among GBI, Inverted Index, and Pruned GBI. Finally, this chapter concludes by comparing the performance of GBI with that of Elasticsearch, a popular enterprise-level search engine that uses Inverted Index.

4.1 Prototype Implementation

A proof of concept prototype is built for GBI, Pruned GBI, and Inverted Index using the PHP scripting language version 7.3. To implement all the three indices (GBI, Pruned GBI, and Inverted Index) using a hash table, the PHP associative array is used. An associative array is a type of array which holds data in key-value pair. Array data structures are made up of hash tables in PHP [33]. Thus, the prototypes built using an associative array are internally implemented as a hash table in PHP. Each entry of the hash table for Inverted Index is identified by a key and an array of values. The key is the unique keyword extracted from a document. The array of values represents the ids of the documents from which the key (keyword) is extracted. In GBI and Pruned GBI, extracted keywords are grouped in pairs of two and then indexed. Thus, two keys (keywords) are used to identify an entry containing an array of values (document ids) in which the keyword pair appears. As Pruned GBI is targeted for executing only Boolean AND queries, only documents in which two or more keywords can be extracted are indexed. In the case of GBI, documents resulting in one keyword after extraction are also indexed. Thus, a separate entry in the hash table of GBI is created to denote documents in which only one keyword is extracted. One more entry is also created in GBI to represent each keyword node in the graph structure to store the unique common documents ids found in the edges connecting the node. Figures 2.2, 3.3 and 3.5 represents the hash table structure of Inverted Index, GBI and Pruned GBI respectively.

Figure 4.1 describes the process of building GBI, Pruned GBI, and Inverted In-

dex from the workload data. The workload data file contains the workload for the experiment for which the index structures have to be built. The workload data file contains a series of entries in which each entry denotes a document to be indexed. Each entry has a set of keywords and id of the document to which the keywords belongs to. Depending on the experiment, one or many workload data files are possible. The workload data files are read one by one to construct the index structures by using the respective indexing algorithms of GBI, Pruned GBI, and Inverted Index. Pruned GBI is a memory-optimized version of GBI built to handle only Boolean AND queries. Thus, Pruned GBI cannot handle other search queries as efficiently as GBI. Hence, GBI is used for all the search experiments. Pruned GBI is only used for comparing the indexing time and memory usage with GBI and Inverted Index to show how much memory can be saved by using Pruned GBI if only Boolean AND queries are being executed. Therefore, Pruned GBI is only built for workloads measuring indexing performance. GBI and Inverted Index are built for all the workloads measuring both search and indexing performance. Finally, the index structures built for a workload in the main memory is written to a file in JavaScript Object Notation (JSON) for later use. JSON is a commonly used file format to store and transmit data containing key-value pairs [76].

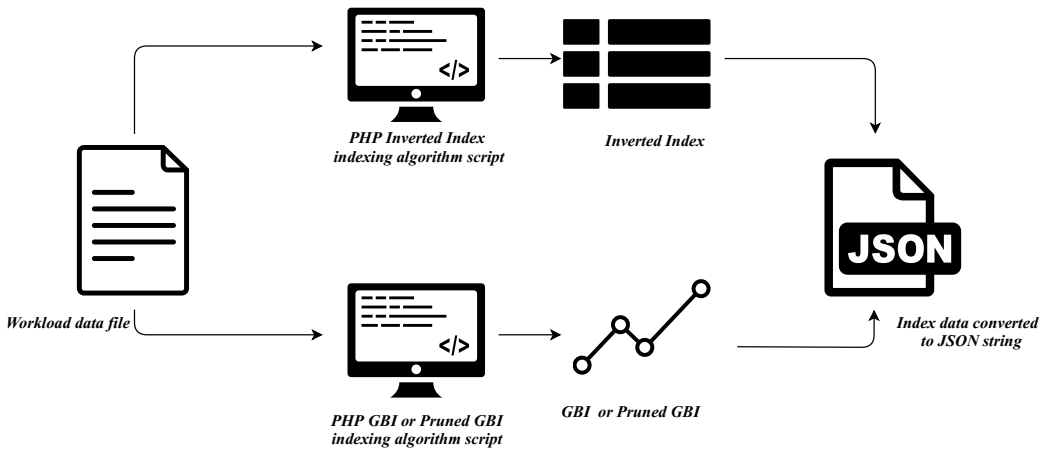


Figure 4.1: Procedure for building prototypes using the workload data.

Figure 4.2 describes the process of executing search queries using GBI and Inverted Index. GBI and Inverted Index data are read from their respective JSON file. The JSON data is then converted back into Inverted Index and GBI hash table and placed in the main memory for performing search operations. The search query algorithm to be performed is executed using Inverted Index and GBI and the results are obtained.

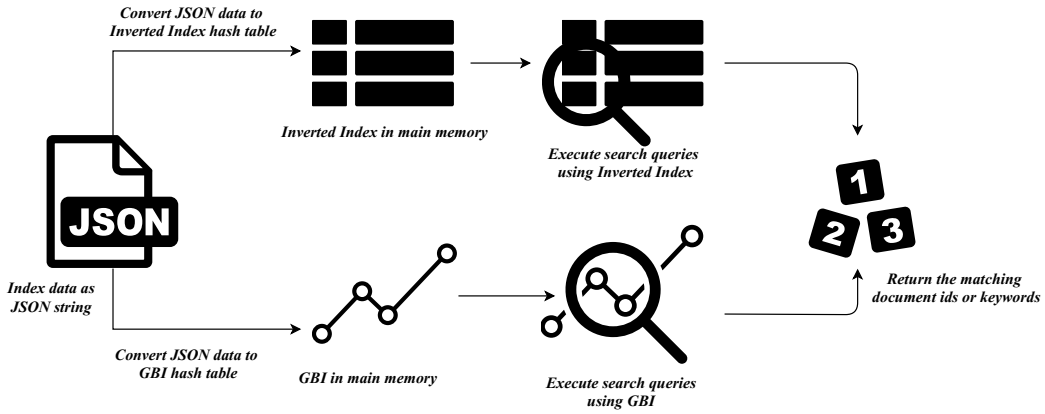


Figure 4.2: Procedure for executing search queries using the prototypes.

For measuring the time consumed for performing the search and indexing operations, PHP's *hrtime* function is used to obtain the system's high-resolution time. For finding search latency, at the start of the search query execution, the *hrtime* function is called to record the time, and then at the end of the search operation, the *hrtime* function is called again to record the current time. The difference between the end time and the start time gives the search latency. For finding the indexing latency, the difference between the end and the start times for indexing each entry from the workload data file is calculated using the *hrtime* function, and all the individual document indexing times are added together to get the total time consumed for indexing a set of documents. The *hrtime* function returns the time in nanoseconds and we converted it to milliseconds and seconds for reporting the search and indexing latencies respectively in this thesis. For obtaining the amount of main memory consumed by the index structures, the PHP's *memory_get_usage* function is used. The *memory_get_usage* function returns the memory in bytes and is converted to megabytes for reporting in this thesis.

4.2 Experimental Setup

The proof of concept prototypes for GBI, Pruned GBI, and Inverted Index described in Section 4.1 is used for experimentation. The experiments are performed on the Carleton University Research Computing and Development Cloud (RCDC) system [77] with the following resource specification: IBM POWER8E processor with 64 GB

memory running Ubuntu version 18.04.4 as the operating system. Each experiment discussed in this chapter was repeated 10 times and the resulting values were averaged and rounded up to 2 decimal places. For a 95% confidence level, less than 5% deviation from the mean value was observed from the multiple runs of each experiment presented in this chapter.

4.2.1 Workload Generation

The following workloads were generated for evaluating the performance of search queries described in Chapter 3.

4.2.1.1 Random Workload

A random workload is used for evaluating the performance of all the search queries discussed in Chapter 3 using GBI and Inverted Index. The random workload is also used for evaluating indexing time and memory for GBI, Pruned GBI, and Inverted Index. Table 4.1 presents the list of parameters and their values for generating the random workload. The default values are presented in boldface in Table 4.1. These parameters are: total number of documents to be indexed (D_{count}), number of keywords in the search queries (SK_{count}), keyword pool containing an array of unique names (K_{pool}), and the size of the keyword pool ($K_{poolsize}$). D_{count} is chosen in the range of 2M to 10M and the search keywords are chosen in the range of 2 to 10 because such values are used by other researchers as well (see [25] for example). Since

10 is the maximum number of search keywords used, it is used as a fixed keyword pool size ($K_{poolsize}$). For exclusive keyword and neighbours of a keyword queries, the number of search keywords (SK_{count}) is always 1. For Boolean queries, all the values of SK_{count} are used except 1 as Boolean queries need at least two keywords. The default SK_{count} value used in Boolean queries is 4 because most Google search queries have up to 4 keywords in length [78].

Table 4.1: Parameters for Random Workload

Parameter	**Values**	**Description**
D_{count}	2M, 4M, **6M**, 8M, 10M	Number of documents indexed
SK_{count}	1, 2, **4**, 6, 8, 10	Number of search keywords
K_{pool}	An array of unique names	Keyword pool
$K_{poolsize}$	10	Size of keyword pool

To generate the documents needed for this workload, a fixed keyword pool, (K_{pool}) is defined with unique names. A document is generated randomly by selecting the keywords from the keyword pool. There are two integer random numbers (X_{random} and Y_{random}) involved in the process of creating a set of keywords for a document to be indexed. The two random numbers follow a uniform distribution U(1, 10). X_{random} specifies the number of keywords that the document will contain. Y_{random} determines which keyword from K_{pool} goes into that document. After generating X_{random}, a loop is ran for X_{random} number of times. In each iteration, a random number Y_{random}, which specifies the index of the keyword in the array K_{pool}, is generated. Y_{random} is generated repetitively until a unique keyword is drawn from K_{pool} which is not in the set of already selected keywords. Finally, a set of unique keywords for that

document is generated after the termination of the loop. The indexing algorithm described in Algorithm 1 is used to construct the GBI and the same set of keywords is also used to construct the Inverted Index. This procedure is repeated for generating D_{count} documents. By checking the documents generated, it has been observed that all the 10 keywords co-occur with each other in at least one document within the first few documents. Thus, each of the 10 keywords has the remaining 9 keywords as neighbours in GBI.

A factor at a time experimentation is performed using this workload: one parameter is varied while the other parameters are held at their default values. Two experiments (Experiment A and B) of this workload have been conducted based on varying the two workload parameters - D_{count} and SK_{count}, one at a time. In Experiment A, the number of documents indexed D_{count} has been increased from 2M to 10M in steps of 2M while SK_{count} is held at 4. In Experiment B, D_{count} has been fixed at 6M and SK_{count} is increased from 2 to 10 in steps of 2. Both Experiment A and B are conducted for evaluating the performance of Boolean queries. Since only 1 search keyword is used in exclusive and neighbours of a keyword queries, only Experiment A has been performed for these queries.

4.2.1.2 Relationship-based Workloads

The relationship-based workloads are only used for evaluating the search performance of Boolean queries. This workload is created to study the impact of the relationship among search keywords on the search performance for Boolean queries. Table 4.2

presents the list of parameters and their values for generating the relationship-based workloads. The D_{count} and $K_{poolsize}$ values are the same as random workload. SK_{count} is set as 4 as it is the default number of search keywords used for the random workload. We define three different type of relationship between the search keywords: *No*, *Partial* and *Full* relationship. These relationships are discussed next.

No relationship: A *No* relationship between keywords refers to the situation in which none of the search keywords appear together in any of the documents indexed.

Partial relationship: A *Partial* relationship between keywords refers to the situation in which some keywords among the entire search keyword set appear together in one or more documents.

Full relationship: A *Full* relationship between keywords refers to the situation in which all the keywords in the search keyword set appear together in one or more documents.

Table 4.2: Parameters for Relationship-based Workload

Parameter	**Values**	**Description**
D_{count}	2M, 4M, 6M, 8M, 10M	Number of documents indexed
SK_{count}	4	Number of search keywords
K_{pool}	An array of unique names	Keyword pool
$K_{poolsize}$	10	Size of keyword pool

For generating workloads for all the three relationship-based experiments, all the 10 keywords in K_{pool} are utilized. The ways the 10 keywords are utilized to create

documents are described next. D_{count} has been increased from 2M to 10M in steps of 2M for all the experiments. Out of 10 keywords, 6 keywords which are not used as the search keywords appear in all the document indexed. The remaining 4 keywords are used as search keywords. These 4 search keywords are controlled on how they can co-appear with each other in a document. Let the 4 search keywords be *Anthony*, *Brutus*, *Caesar* and *Romeo*. The way these 4 search keywords can co-occur within a document based on the relationship is discussed next.

In the experiment for *No* relationship, the documents are created in such a way that no search keywords among the 4 search keywords appear together in any documents. To achieve this, each of the D_{count} document indexed contains only 1 of the 4 search keywords. Thus, each of the documents indexed contains 1 of the 4 search keywords (either *Anthony* or *Brutus* or *Caesar* or *Romeo*) apart from the fixed set of 6 keywords.

In the experiment for *Partial* relationship, the 4 search keywords are divided into 2 keyword pairs. Let one keyword pair be (*Anthony* and *Brutus*) and other keyword pair be (*Caesar* and *Romeo*). Each document indexed contains either (*Anthony* and *Brutus*) or (*Caesar* and *Romeo*) with the rest of the 6 keywords such that both pairs of keywords does not appear together in any documents out of the D_{count} documents indexed.

In the experiment for *Full* relationship, the documents are created in such a way that all the 4 search keywords appear together in one or more documents. To achieve this, the total number of documents D_{count} to index is divided into 2 parts. One-half

of the documents contain all the 4 search keywords together and the other half of the documents contain only 2 keywords together (out of 4). This is done to avoid all the documents indexed containing all the search keywords together. Thus, each of the documents contains either all the search keywords "*Anthony*, *Brutus*, *Caesar* and *Romeo*" with the rest of the 6 keywords or only "*Anthony* and *Brutus*" with the rest of the 6 keywords.

All the documents indexed containing all the 4 search keywords is a special case of the experiments concerning the *Full* relationship. For this experiment, all the search keywords *Anthony*, *Brutus*, *Caesar* and *Romeo* appear in all the documents indexed (D_{count}). Thus, all the documents contain all the 10 keywords from the K_{pool}. The motivation for doing this special experiment is to analyze how much better a GBI performs than the Inverted Index in the worst-case scenario of all the documents containing all the search keywords. This is the worst case because the size of the postings list of all the search keywords (in Inverted Index) or keyword pairs (in GBI) is equal to the total document count D_{count} indexed. For example, if 2 million documents are being indexed then all the search keywords *Anthony*, *Brutus*, *Caesar* and *Romeo* appear in all the 2 million documents.

4.2.1.3 Workloads for Neighbours of a Keyword Queries

In this section, a special set of workloads is generated to obtain a detailed performance insights for executing neighbours of a keyword queries with GBI and Inverted Index. This workload is not used for any other queries. Table 4.3 presents the list

of parameters and their values for this workload. The default values are presented in boldface in Table 4.3. These parameters are: number of keywords in the search queries (SK_{count}), keyword pool containing an array of unique names (K_{pool}), size of the keyword pool ($K_{poolsize}$), the total number of keywords found in both GBI and Inverted Index except the search keyword (KI_{count}), the number of keywords (neighbours) that co-occur with the search keyword (N_{count}), and the point of neighbourhood (P_n). Point of neighbourhood or the point of co-occurrence defines the number of document ids to be compared between the postings list of search keyword and the other keyword to confirm that they are neighbours. At the end of P_n comparisons, we get the id of the document in which the search keyword and the neighbour keyword has made their first appearance together.

Neighbours of a search keyword query has only one search keyword hence the value of SK_{count} is always 1. Neighbours of a keyword queries can be used in ERP applications where an item could be associated with thousands of products. That is the reason for choosing the $K_{poolsize}$ as 10,001 and N_{count} in the range of 2K to 10K. It is 10,001 because a maximum of 10,000 possible neighbours can be created for the search keyword plus the number of search keywords which is 1. When indexing a large number of documents, any two keywords can appear together in the very first document or the last document indexed. Since we are considering millions of documents, 20K to 100K seems to be a moderate range of values for point of neighbourhood.

Four types of experiments are conducted (Experiment *A, B, C, D*). The workloads for these experiments are discussed next. For demonstrating the workloads, let us

Table 4.3: Parameters for Neighbours of a Keyword Query Workload

Parameter	Values	Description
K_{pool}	An array of unique names	Keyword pool.
SK_{count}	1	Number of search keywords.
$K_{poolsize}$	10001	Size of keyword pool.
KI_{count}	**2K**, 4K, 6K, 8K, 10K	Total number of keywords found in the index except the search keyword.
N_{count}	**2K**, 4K, 6K, 8K, 10K	Neighbour count of the search keyword.
P_n	20K, 40K, **60K**, 80K, 100K	Point of neighbourhood.

consider the search keyword to be *Romeo*.

In Experiment *A*, N_{count} and KI_{count} are fixed to their default values of 2K and P_n is varied from 20K to 100K. To generate the workloads for this experiment, 2K unique keywords are selected from K_{pool} containing 10,001 keywords. In addition to 2K keywords, one more keyword is chosen for being a search keyword. The 2000 keywords are selected in such a way that 1000 keywords appear alphabetically earlier than the search keyword and the other 1000 keywords appear alphabetically later than the search keyword. For example, the keyword *Santos* appears alphabetically later than the search keyword *Romeo*, and the keyword *Anthony* appears alphabetically earlier than the search keyword *Romeo*. Each of 2000 keywords has a postings list size equal to P_n and will co-occur with the search keyword *Romeo* in the last document in its postings list. For example, if P_n is chosen as 20K, then each of the 2000 keyword will have 20,000 documents in its postings list and appear together with the search keyword (*Romeo*) at the 20,000$^{\text{th}}$ document (last document) in its postings list. Thus, to check whether a keyword is related to *Romeo*, 20,000 document ids should be compared to confirm the neighbourhood (co-occurrence). The goal of this experiment is to investigate the effect of P_n on the execution of neighbours of a keyword query

using Inverted Index and GBI. This experiment shows how GBI and Inverted Index search latency is affected by delaying the co-occurrence of keywords with the search keyword in the documents indexed.

In Experiment *B*, N_{count} and KI_{count} is varied from 2K to 10K in steps of 2K and P_n is fixed to its default value of 60K. To generate workloads for this experiment, N_{count} keywords are selected from K_{pool} containing 10,001 keywords. At any given time, the entire index structure contains only the search keyword and the chosen N_{count} keywords. Each of the N_{count} keywords has a postings list size equal to P_n which is 60K and will co-occur with *Romeo* at its last document in its postings list. For example, if N_{count} is chosen as 2K, then each of the 2000 keywords will appear together with the search keyword (*Romeo*) at the 60,000$^{\text{th}}$ document in its postings list. Thus, to check whether a keyword is related to *Romeo*, 60,000 document ids should be compared to confirm neighbourhood (co-occurrence). The goal of this experiment is to study the effect of N_{count} and KI_{count} on the execution of neighbours of a keyword query with Inverted Index and GBI.

In Experiment *C*, N_{count} and P_n are fixed to their default values of 2K and 60K respectively and KI_{count} is varied from 2K to 10K in steps of 2K. To generate workloads for this experiment, KI_{count} keywords are selected from the keyword pool (K_{pool}) along with the search keyword for indexing. The keywords are chosen in such a way that all the keywords appear alphabetically later than the search keyword *Romeo*. Out of the KI_{count} keywords selected, only 2000 keywords will be selected as neighbours and the remaining keywords will be non-neighbours. Documents are generated in such

a way that each of the 2000 neighbour keywords appears together with *Romeo* at the 60,000$^{\text{th}}$ document indexed. The size of the postings list for the remaining non-neighbour keywords is also equal to the default value of P_n. Thus, to check whether a keyword is neighbour to *Romeo* or not, 60,000 document ids should be compared to confirm the neighbourhood (co-occurrence). The goal of this experiment is to study the effect of KI_{count} on the search latency achieved by GBI when all the keywords indexed are alphabetically later than the search keyword.

Experiment D is similar to Experiment C except for the fact that keywords are chosen from the keyword pool such that all the keywords appear alphabetically earlier than the search keyword *Romeo*. The goal of this experiment is to study the effect of KI_{count} on the search latency achieved by GBI when all the keywords indexed are alphabetically earlier than the search keyword.

4.2.2 Performance Metrics

Table 4.4 describes the performance metrics used in the experiments conducted. Three performance metrics are collected: T_s, T_i and M_i. A definition of each metric is provided next.

- T_s: The search latency associated with the queries execution. It is the difference between the end and start times of the search algorithm used in GBI and Inverted Index.

Table 4.4: Performance Metrics

Parameter	**Description**	**Unit**
T_s	Average time consumed for executing a query	milliseconds (ms)
T_i	Average time consumed for indexing a set of documents	seconds (s)
M_i	Amount of main memory consumed	Megabytes (MB)

- T_i: The latency associated with indexing a set of documents. To calculate T_i, the difference between the end and the start times for indexing each document is calculated and all the differences are added to get the final value.

- M_i: It is the difference between the amount of memory consumed before and after indexing a set of documents using GBI and Inverted Index in the main memory.

Note that a log scale has been used for the Y-axis in some of the figures presented in this chapter.

4.3 Experiments for Boolean Queries

This section presents the search performance analysis for Boolean queries in GBI and Inverted Index using random and relationship-based workloads.

4.3.1 Boolean AND Queries

4.3.1.1 Analysis of the Search Latency Using Random Workload

Figure 4.3 shows the result of Experiment A. As D_{count} increases, the search time (T_s) for Inverted Index is increasing at a much faster rate compared to GBI. This is because as D_{count} increases, there is a sharp growth in the size of the smallest postings list among all the search keywords postings lists in Inverted Index. This leads to the comparison of more document ids resulting in increased search time. Even if postings list for each search keyword pairs also grows with D_{count} in GBI, only a single intersection operation is needed between 4 search keywords as discussed in Section 3.5.1.1. This leads to a lower search time in GBI. Thus, GBI shows an average improvement of 70% over Inverted Index in this experiment.

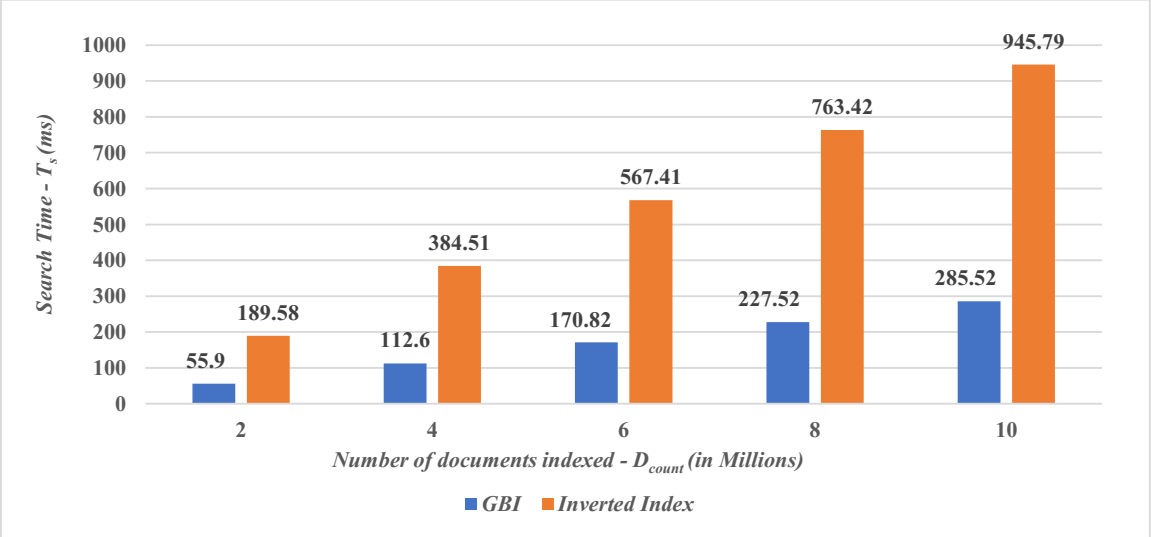


Figure 4.3: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean AND query.

In the case of Experiment B, Figure 4.4 shows that for any given SK_{count}, GBI is performing better than Inverted Index. The search time (T_s) is almost negligible (around 0.03 ms) for GBI with a search latency improvement of 99.9% when there are only two keywords involved. This is because the ids of the documents are stored in keyword pairs so a single direct access (hash table lookup) gives the result in GBI. For $SK_{count} > 2$, as the number of search keyword increases, the number of intersection operations increases in Inverted Index resulting in higher search time latency. Although the number of intersection operations increases in GBI as well, it will always be lower than Inverted Index as discussed in Section 3.5.1.1. This results in GBI performing well even when the number of search keywords increases. For $SK_{count} > 2$, the average overall improvement in GBI is 62% in comparison to

Inverted Index.

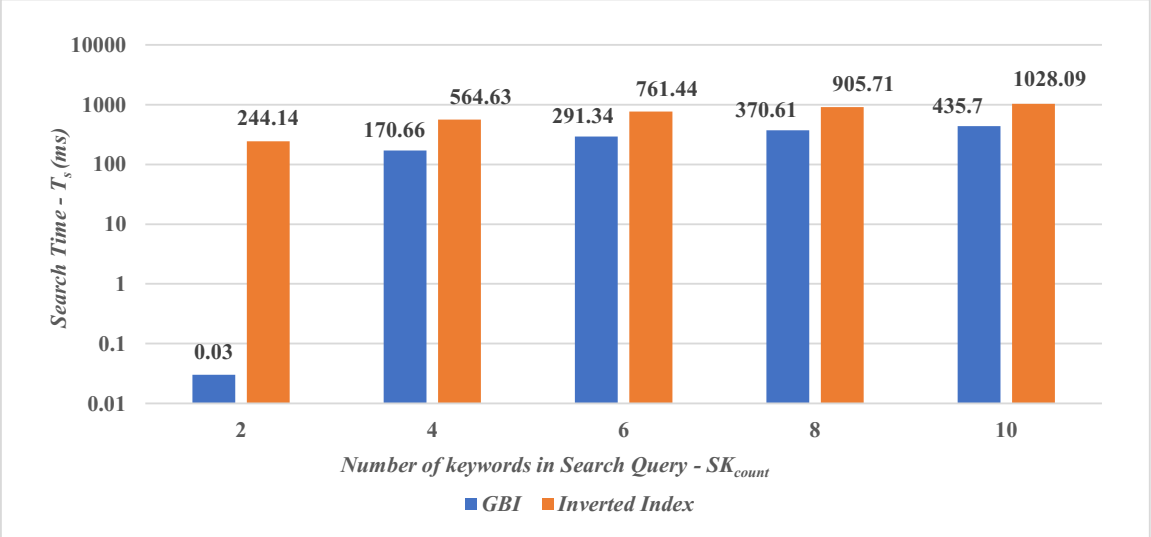


Figure 4.4: Effect of number of search keywords (SK_{count}) on the search time (T_s) in GBI and Inverted Index when number of documents indexed (D_{count}) is kept at the constant value of 6M for a Boolean AND query.

4.3.1.2 Analysis of the Search Latency Using Relationship-based Workloads

Figure 4.5 shows the performance of GBI and Inverted Index when the search keywords do not appear together in any of the documents indexed (i.e. the *No* relationship). For a given D_{count}, GBI shows a significant performance improvement of 99.9% over Inverted Index with a negligible search time (T_s) of around 0.02 ms. This high performance is because GBI checks whether all the search keywords occur together in any of the documents before performing an intersection. Checking for the simultaneous occurrence of two keywords in a document (connection between two keywords)

is a hash table lookup in GBI. Since there is no simultaneous occurrence of any of the search keywords, GBI stops further action and does not perform any intersection operation. If there are more connections between keywords to check, then the execution time for GBI increases. Since hash table lookup takes a negligibly small amount of time, the search time for checking more connections will not increase significantly. As the number of search keyword is kept at a constant value of 4, the number of connections to check remains the same. Thus, a similar time is observed for GBI for different values of D_{count}. With Inverted Index, the execution time for Boolean AND queries depend on the size of the smallest postings list among the search keywords. As the total number of documents indexed (D_{count}) increases, the size of the smallest postings list also increases leading to an increase in the search time for Inverted Index.

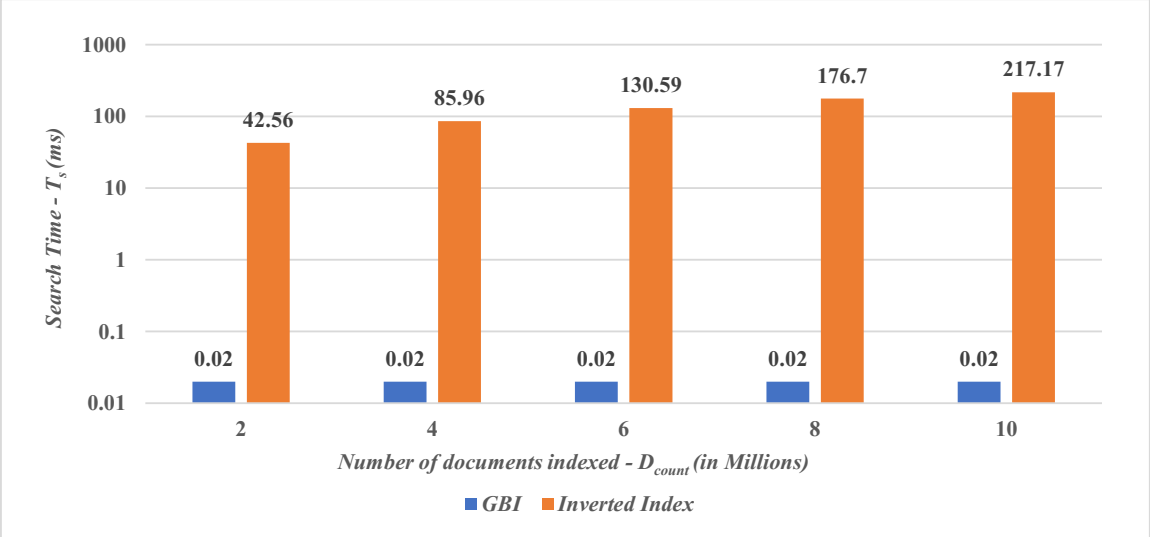


Figure 4.5: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *No* relationship among search keywords for a Boolean AND query.

Figure 4.6 shows the performance of GBI and Inverted Index when some of the search keywords appear together in one or more documents indexed (i.e. the *Partial* relationship). For a given D_{count}, Once again, GBI shows a significant performance improvement of 99.9% over Inverted Index with a search time of around 0.03 ms. Since some of the keywords do not appear together in any of the documents, no common document is possible for all the search keywords. Hence, no further action is required with GBI and the search is terminated. With Inverted Index, the execution time depends on how many keywords among the search keywords set appear together in a document. If more keywords are appearing together in many documents, then Inverted Index takes more time to perform the intersection.

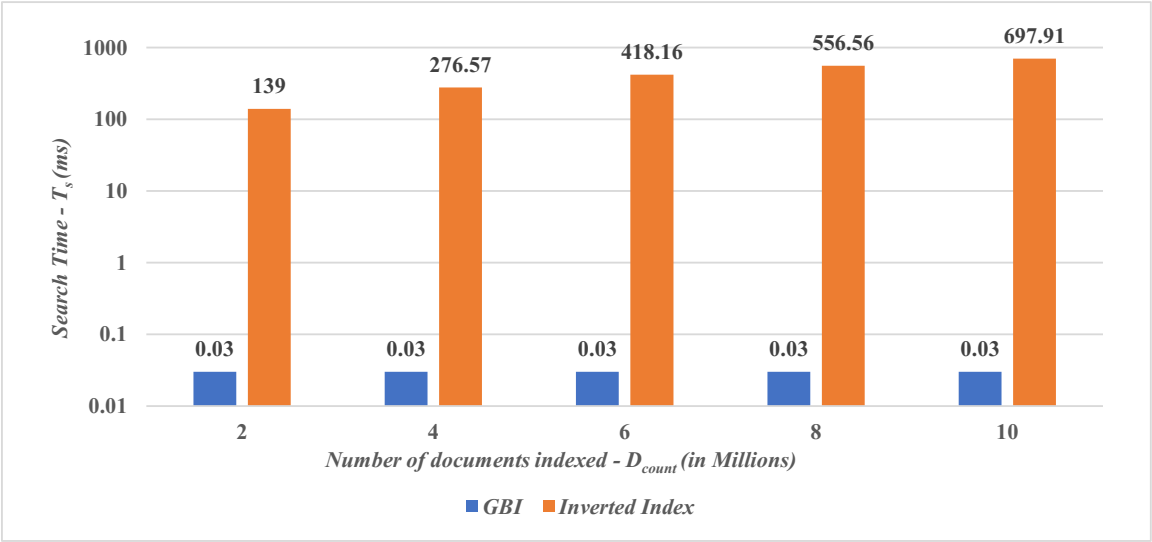


Figure 4.6: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Partial* relationship among search keywords for a Boolean AND query.

Figure 4.7 shows the performance of the GBI and Inverted Index when all the

search keywords occur together in some of the documents indexed (i.e. the *Full* relationship). Figure 4.8 corresponds to the special case of the *Full* relationship experiment where all the search keywords appear in all the documents indexed. In both Figures, for any given value of D_{count}, GBI shows a better performance over Inverted Index. This is because intersections in GBI are done in pairs and so the number of intersections performed is only one as opposed to three intersections performed with Inverted Index as discussed in Section 3.5.1.1. In both the cases, the performance improvement achieved by GBI is becoming much more evident for larger values of D_{count} with an overall average improvement of approximately 67%.

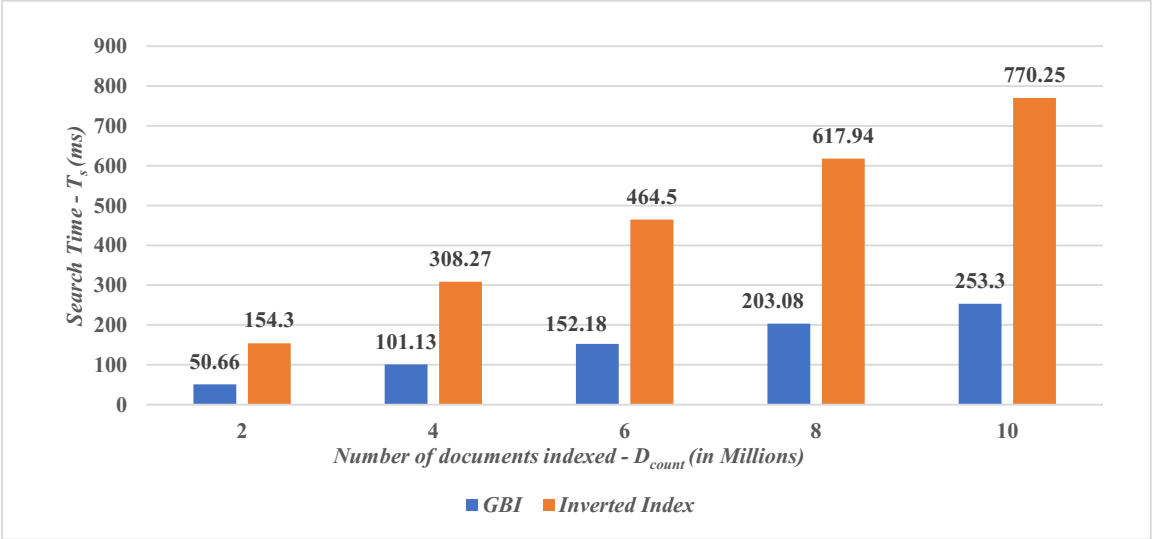


Figure 4.7: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords for a Boolean AND query.

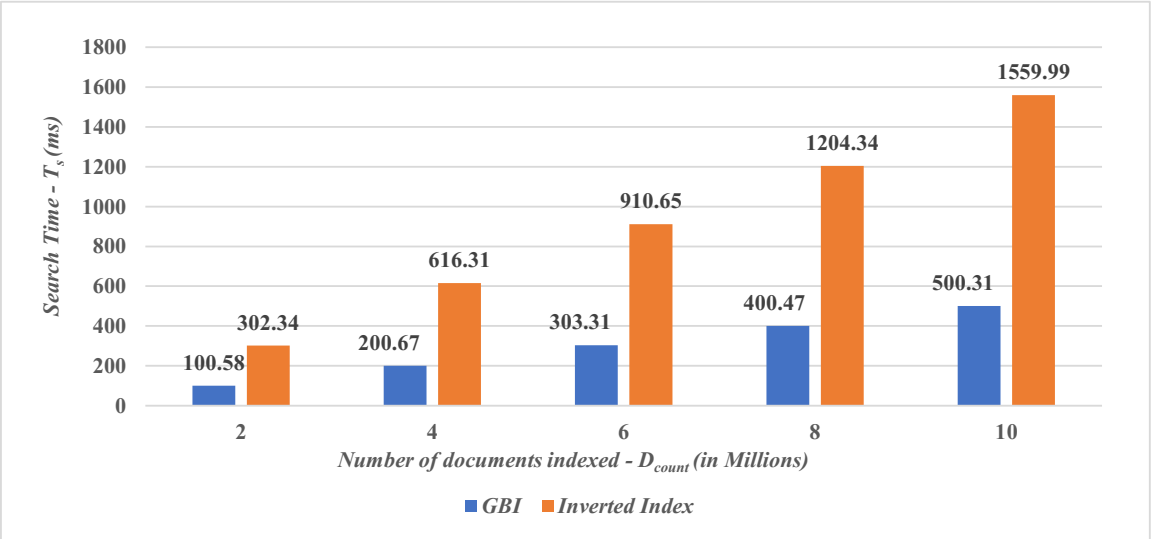


Figure 4.8: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean AND query.

4.3.2 Boolean NOT Queries

4.3.2.1 Analysis of the Search Latency Using Random Workload

For Experiment A, Figure 4.9 shows that T_s for Inverted Index is higher than T_s for GBI. This is because Inverted Index does not have knowledge about the common documents for any two keywords indexed. It has to iterate through the entire postings list to check and remove the necessary document ids. As D_{count} increases, there will be a growth in the size of the postings list of all the search keywords. This leads to iterating through more documents ids to check and remove. Hence it leads to

more search latency in Inverted Index. In case of GBI, common document ids for any two keywords are stored separately as well. This helps in iterating over common document ids and removing them instead of iterating the entire postings lists of search keywords. This helps lower the search time for Boolean NOT queries execution in GBI. GBI shows an average improvement of 28% in comparison to Inverted Index for Experiment A.

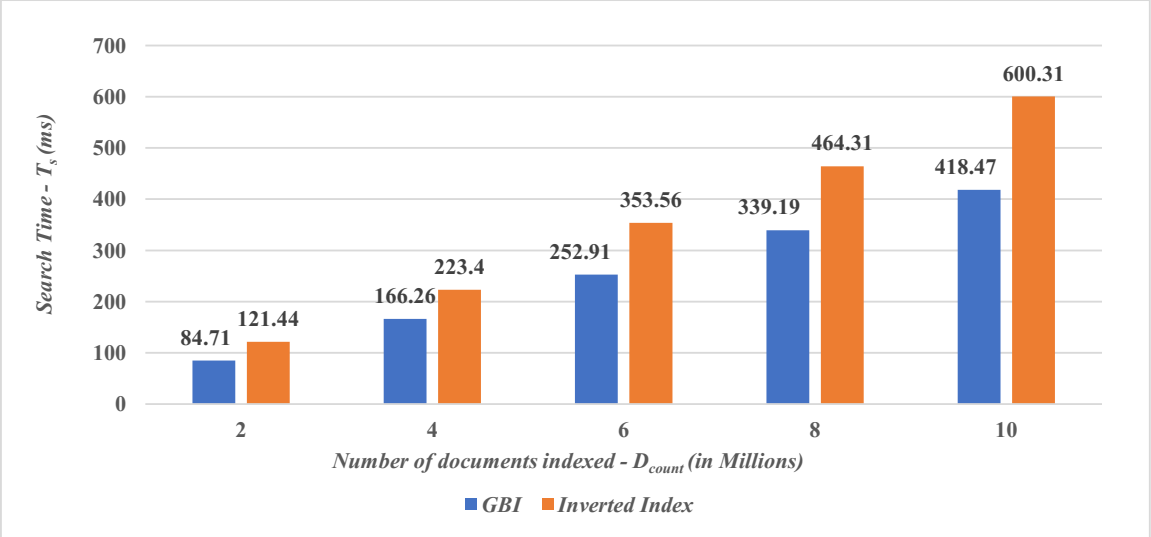


Figure 4.9: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean NOT query.

For Experiment B, Figure 4.10 also shows that GBI is performing better than Inverted Index. It is due to the same reason of knowing exactly what document ids to be removed instead of iterating through the entire postings list of the search keywords. The interesting point to note is that search time (T_s) is negligible (0.04 ms) for GBI when all the ten keywords are involved in a Boolean NOT query. To

explain this, consider a Boolean NOT query in which the search keyword *Anthony* is to be present in the document and nine search keywords E_1, E_2, ..., E_9 should not be present in the document. Also, let us assume that *Anthony* appears either alone or with one of the nine keywords in any document indexed. This means that *Anthony* is not associated with any other keyword other than these nine keywords. GBI performs an optimized execution where instead of removing common documents ids for E_1 and *Anthony*, E_2 and *Anthony* and so on. It directly returns the ids of the document in which only *Anthony* exists. This is possible because GBI checks whether all the keywords to be excluded is equal to the total number of neighbours of the search keyword before performing Boolean NOT operation as discussed in Section 3.5.2.1. Thus, GBI performs significantly better with an overall improvement of 99.9% when returning the documents which should not contain any of the neighbours of a search keyword. Recall from Section 3.1 that a neighbour of a keyword X is the keyword that is modeled by a (neighbouring) node that is directly connected to the node corresponding to X in the graph structure of GBI. For SK_{count} ! = 10, the average overall improvement in GBI is 28% in comparison to Inverted Index.

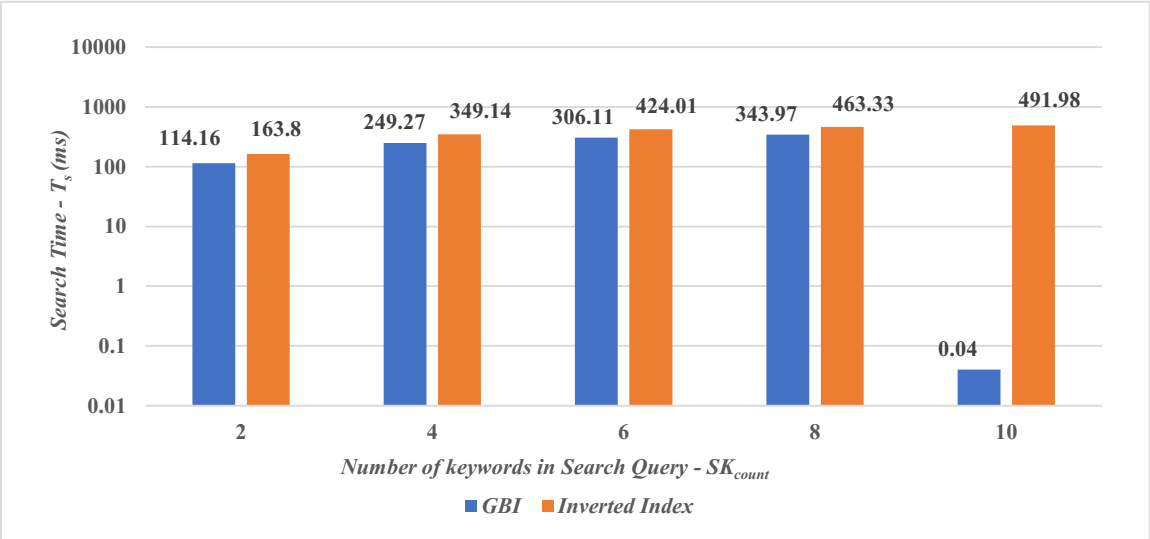


Figure 4.10: Effect of number of search keywords (SK_{count}) on the search time (T_s) in GBI and Inverted Index when number of documents indexed (D_{count}) is kept at the constant value of 6M for a Boolean NOT query.

4.3.2.2 Analysis of the Search Latency Using Relationship-based Workloads

Figure 4.11 shows the performance of GBI and Inverted Index when the search keywords do not appear together in any of the documents indexed (i.e. the *No* relationship). GBI shows a significant performance improvement of 99.9% over Inverted Index having a negligible search time (T_s) of around 0.02 milliseconds. This high performance improvement is because GBI checks whether the search keywords that are not to be found are neighbours of the search keyword that is to be found in a document before performing Boolean NOT operation. Checking whether any two keywords are neighbours is a hash table lookup in GBI. Since there is no simultaneous occurrence

of the search keywords in any of the documents indexed, GBI directly returns all the document ids in which the search keyword that is to be present appears. To illustrate this performance gain of GBI, consider the Boolean NOT query in which the search keyword *Anthony* should be present whereas the search keywords *Caesar* or *Brutus* or *Romeo* should not be present in the document. According to the workload for *No* relationship experiment, *Anthony* does not co-occur with any other search keywords (*Caesar* or *Brutus* or *Romeo*). GBI using its relationship knowledge finds that no common document ids are possible between the search keywords. Hence, GBI directly returns the ids of the documents in which *Anthony* appears and terminates the search. With Inverted Index, the knowledge of relationship among the search keywords is not present. Therefore, Inverted Index being unaware that there is no common document ids between the search keywords compares the postings list of *Anthony* with all the remaining search keywords postings lists to check and remove any common document ids. This leads to increased search time for Inverted Index.

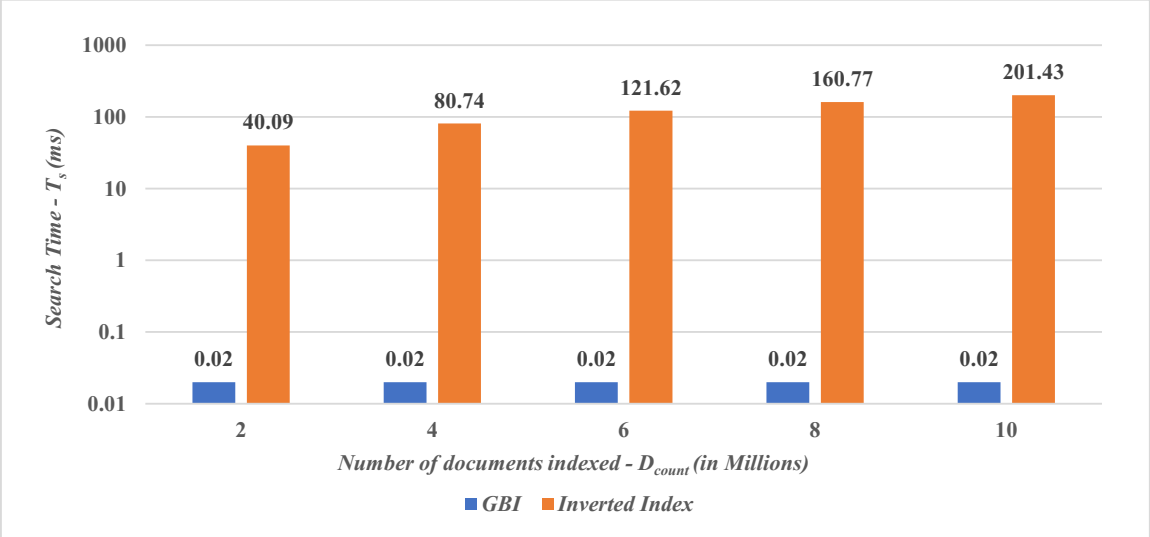


Figure 4.11: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *No* relationship among search keywords for a Boolean NOT query.

Figure 4.12 shows the performance of GBI and Inverted Index when some of the search keywords appear together in one or more documents indexed (i.e. the *Partial* relationship). GBI shows an average performance improvement of 30% over Inverted Index. To illustrate this performance gain of GBI, consider the Boolean NOT query in which the search keyword *Anthony* should be present whereas the search keywords *Caesar* or *Brutus* or *Romeo* should not be present in the document. According to the workload for the Partial relationship experiment, apart from the fixed set of keywords, *Brutus* is the only search keyword that appears with *Anthony*. Neither *Caesar* nor *Romeo* appears with *Anthony* in any document indexed. GBI, by using hash table lookup, checks the different neighbours of *Anthony* and finds that *Brutus* is the only neighbour among the search keywords that are to be removed. Therefore,

GBI removes only the common document ids between *Anthony* and *Brutus*. GBI does not care about the other search keywords *Caesar* or *Romeo* because it finds that these keywords are not neighbours of *Anthony* and so no common document exists. Thus, the knowledge of relationship among the search keywords in GBI leads to a reduced search latency for GBI. Since this relationship knowledge is not present in Inverted Index, the search operation in Inverted Index compares *Anthony*'s postings list with the postings lists of other search keywords (*Romeo* or *Caesar*) instead of directly comparing with only *Brutus*'s postings list resulting in increased search latency.

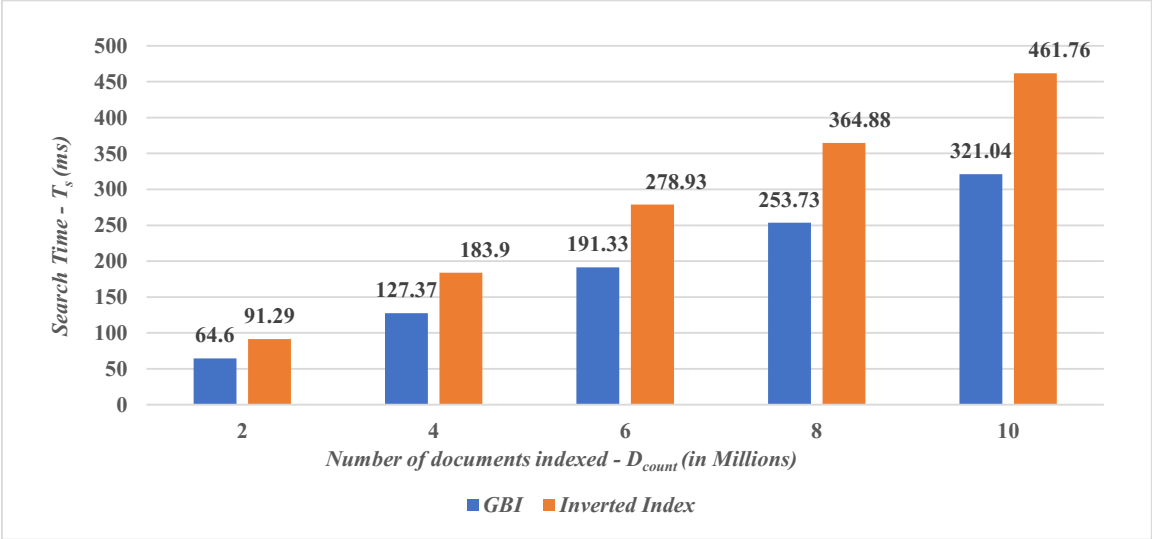


Figure 4.12: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Partial* relationship among search keywords for a Boolean NOT query.

Figure 4.13 shows the performance of GBI and Inverted Index when all the search keywords occur together in some of the documents indexed (i.e. the *Full* relationship). Figure 4.14 shows a special case of the *Full* relationship category where all the search

keywords appear in all the documents indexed. In both the figures, for any given D_{count}, GBI shows a comparable performance with Inverted Index. For the workload of *Full* and the special case of *Full* relationship, *Anthony* appears in all the documents in which *Brutus*, *Romeo*, and *Caesar* appear together. There is no document in which the search keywords (*Anthony* or *Brutus* or *Romeo* or *Caesar*) alone appears. Thus, the Boolean NOT operation for both of the workloads results in NULL. Since the total number of document ids to process for removing common document ids and the number of times the common document ids has to be removed among the search keywords postings lists remains the same for both Inverted Index and GBI, the comparable performance between the two techniques in this case is expected.

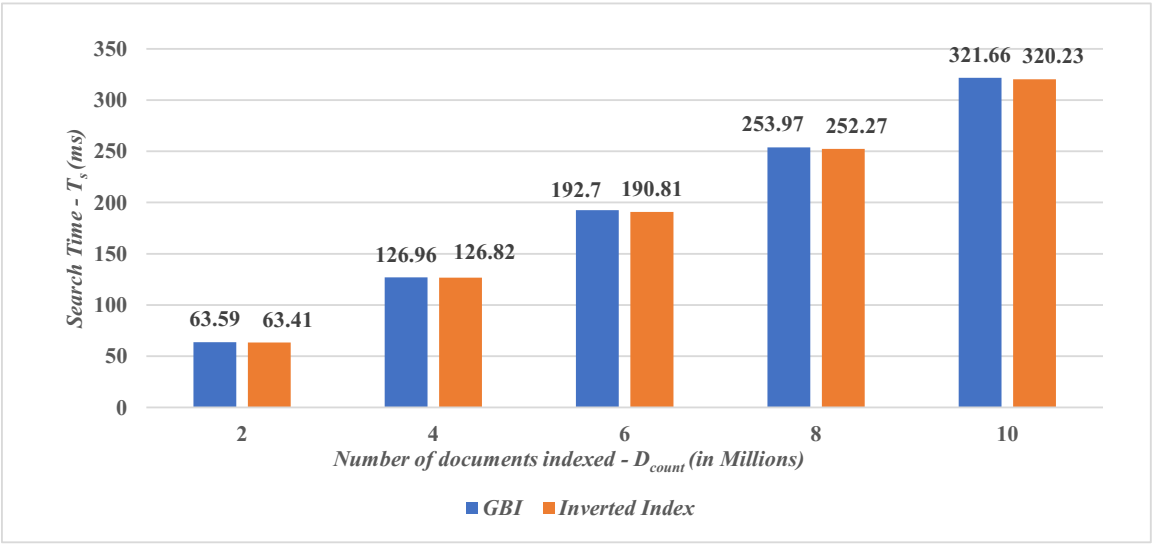


Figure 4.13: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords for a Boolean NOT query.

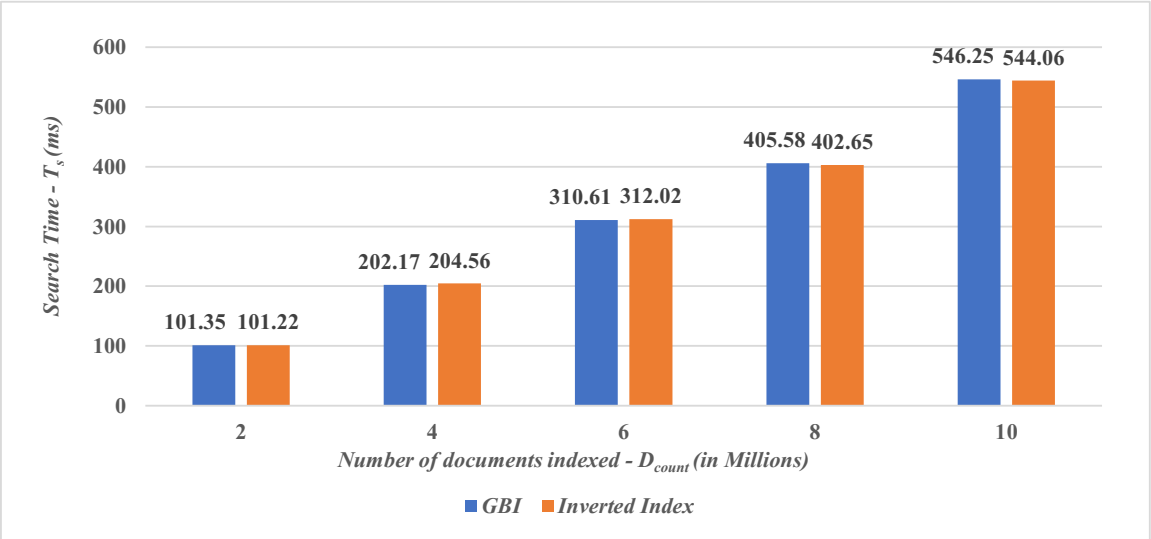


Figure 4.14: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean NOT query.

4.3.3 Boolean OR Queries

4.3.3.1 Analysis of the Search Latency Using Random Workload

The results of Experiment A and B are shown in Figure 4.15 and Figure 4.16 respectively. In both the Figures (4.15 and 4.16), it can be seen that for a given D_{count}, search time (T_s) for Inverted Index is higher in comparison to T_s for GBI. This is because, in Inverted Index, the union operation is performed between individual postings lists of the search keywords to get the resultant document ids. A union operation involves removal of duplicates between the postings lists of the search keywords and

Inverted Index requires additional time for iterating through all the document ids in a postings list to check for duplicates and remove them. In Inverted Index, as D_{count} increases, the size of the postings list of all the search keyword increases. This results in more document ids to be checked for duplicates when performing the union operation between all the postings lists. In the case of GBI, the exact duplicates to be removed from the postings lists are know *apriori*. This is possible because GBI also stores the ids of the common document (if any) between any two keywords indexed. This setup leads to iterate over only the duplicate elements to be removed instead of iterating over the entire postings list resulting in reduced search latency in GBI as explained in Section 3.5.3.1. In Experiment *B*, as the number of search keyword increases, more number of union operations are performed leading to a increase in search time in both GBI and Inverted Index. When $SK_{count} = 10$, the search keyword count is equal to the total number of keywords found in the index. Therefore, the search operation results in returning all the document ids indexed. As discussed in Section 3.5.3.1, GBI does not perform duplicate removal process when all the document ids found in the search keyword's node have to be removed instead GBI directly returns the document ids in which only the search keywords alone appears in. This happens when all the document ids shared between search keyword and all its neighbours has to be removed. This optimization helps GBI to have lower search time than Inverted Index in this case. Thus, GBI shows a average improvement of 28% in comparison to Inverted Index in both Experiment *A* and *B*.

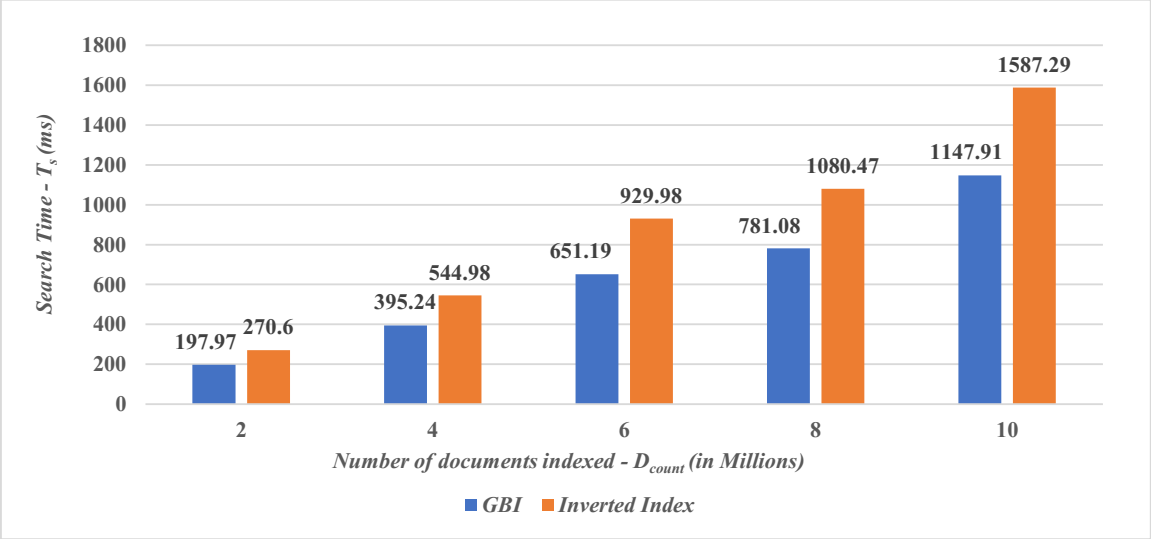


Figure 4.15: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean OR query.

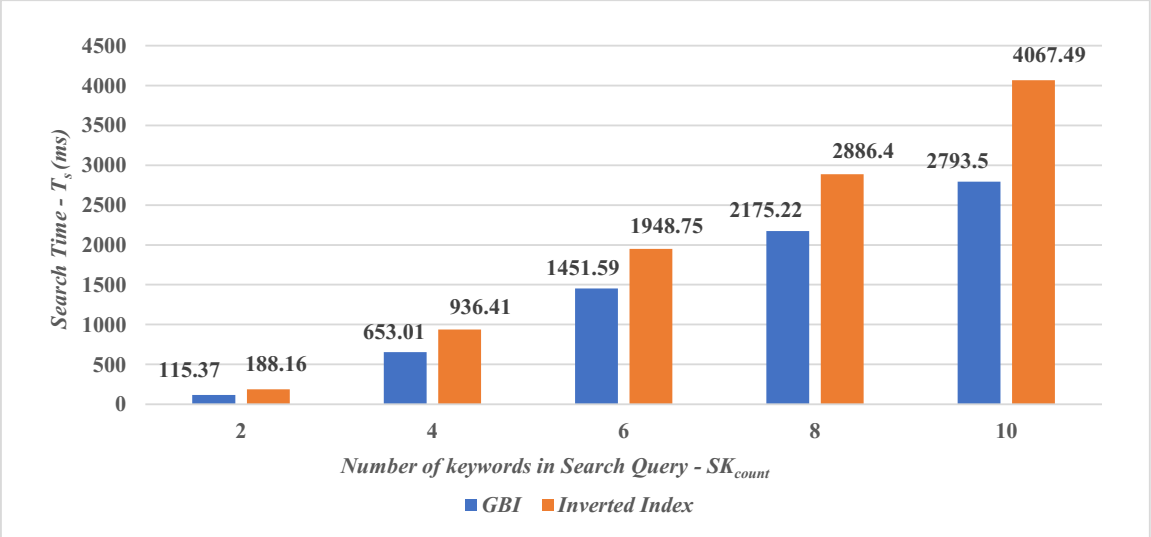


Figure 4.16: Effect of number of search keywords (SK_{count}) on the search time (T_s) in GBI and Inverted Index when number of documents indexed (D_{count}) is kept at the constant value of 6M for a Boolean OR query.

4.3.3.2 Analysis of the Search Latency Using Relationship-based Workloads

Figure 4.17 shows the performance of the GBI and Inverted Index when the search keywords do not appear together in any of the documents indexed (i.e. the *No* relationship). GBI shows a significant performance improvement (99.9%) over Inverted Index with a negligible search time (T_s) of around 0.02 ms. This high performance of GBI is because it checks whether any of the search keywords appear together in any of the documents before performing a union operation. Checking for the simultaneous occurrence of two keywords in a document is a hash table lookup in GBI. Since there is no simultaneous occurrence of any of the search keywords, there is no duplicates to remove so GBI does not perform any union operation. GBI directly returns the ids of all the documents in which each of the search keywords appears in. For Inverted Index, the execution time for Boolean OR queries depend on the size of all the postings lists except for the largest postings list. As the total number of documents (D_{count}) indexed increases, the size of the postings lists also increases resulting in more document ids to be compared for removing duplicates. This leads to an increase in the search time for Inverted Index.

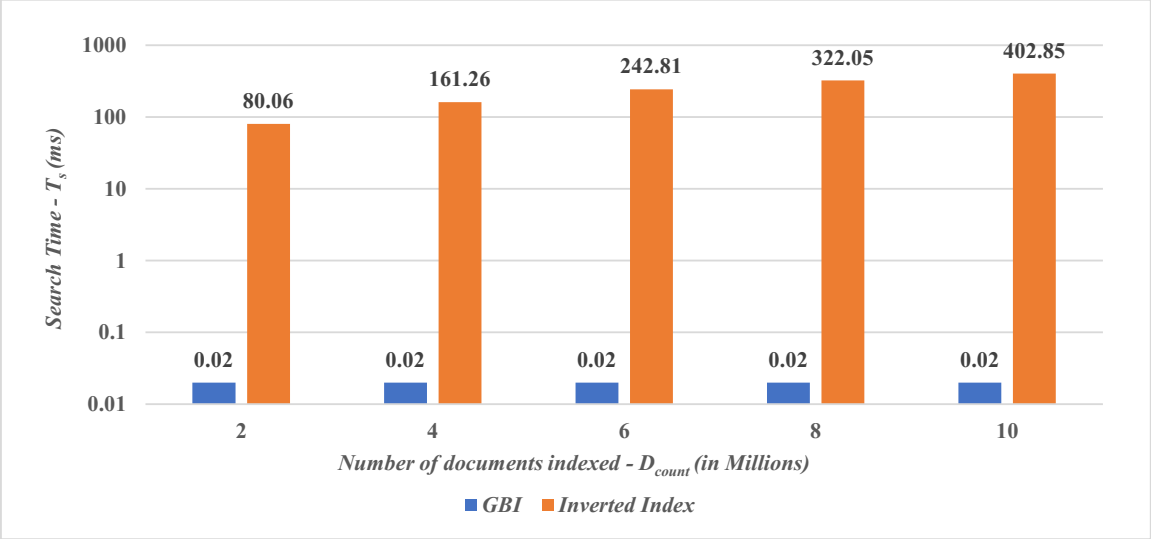


Figure 4.17: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *No* relationship among search keywords for a Boolean OR query.

Figure 4.18 shows the performance of GBI and Inverted Index when some of the search keywords appear together in one or more documents indexed (i.e. the *Partial* relationship). GBI shows a performance improvement of around 29% over Inverted Index. Since some of the keywords do not appear together in any of the documents, only the duplicate document ids for the related keywords are removed to eliminate duplicates while performing the union operation. To explain this performance gain in GBI, consider the example stated for the description of *Partial* workload in Section 4.2.1.2, *Anthony* and *Brutus* appear together in some documents and *Romeo* and *Caesar* appear together in the remaining documents. The GBI technique finds that *Anthony* is connected with only *Brutus* and not with any other search keywords (*Romeo* or *Caesar*). Similarly *Romeo* appears with only *Caesar* and does not appear

with either *Anthony* or *Brutus*. Therefore, in GBI, a Boolean OR operation between *Anthony*, *Brutus*, *Romeo* and *Caesar* involves removing only the duplicate document ids between *Anthony* and *Brutus* and between *Romeo* and *Caesar*. The reduction in number of search keyword's postings lists to compare for removing duplicates leads to lower search time in GBI. With Inverted Index, this knowledge of relationship is not present. Hence, *Anthony*'s postings list will be compared with the rest of the search keywords (*Brutus*, *Romeo*, and *Caesar*) postings lists and again, *Brutus*'s postings list will be compared with *Caesar* and *Romeo* and so on. Thus, the execution time depends on how many keywords among the search keyword set appear together in a document. If a higher number of search keywords are appearing together in many documents indexed, then Inverted Index takes more time to perform the union operation as the entire postings lists of all the search keywords has to be compared leading to an increased time spent on duplicate removal process.

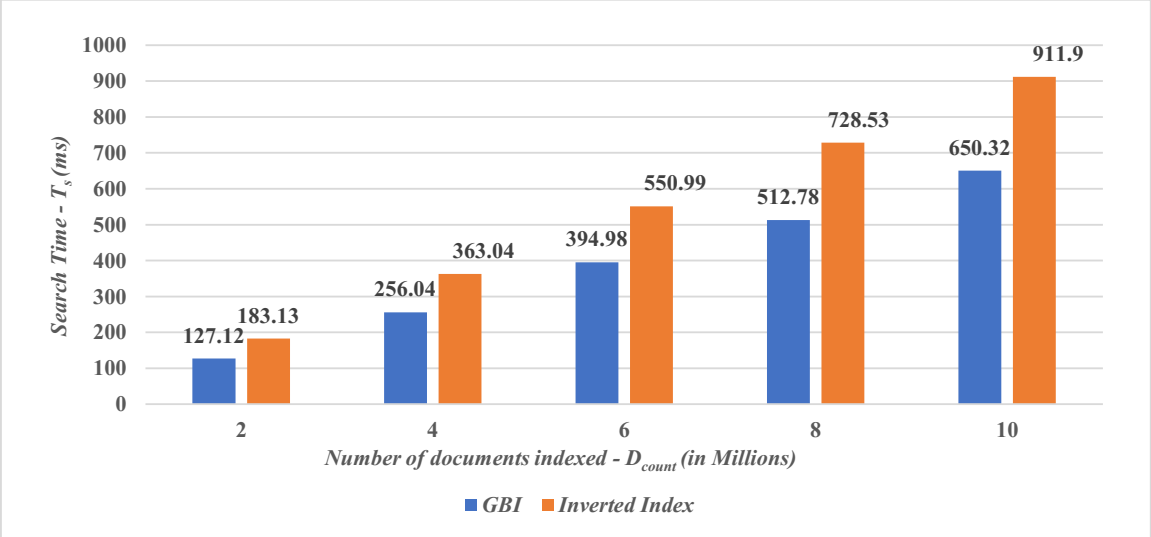


Figure 4.18: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Partial* relationship among search keywords for a Boolean OR query.

Figure 4.19 shows the performance of GBI and Inverted Index when all the search keywords occur together in some of the documents indexed (i.e. the *Full* relationship). Figure 4.20 shows a special case of the *Full* relationship category where all the search keywords appear in all the documents indexed. In both the figures, for any given D_{count}, GBI shows a comparable performance with Inverted Index. For the workload of *Full* and special case of *Full* relationship, the search keyword *Anthony*, *Brutus*, *Romeo*, and *Caesar* always appear together in some or all of the documents indexed respectively. Therefore, in this case, the total number of duplicate document ids to remove remains the same for both Inverted Index and GBI. This is the reason for the comparable performance between the two techniques.

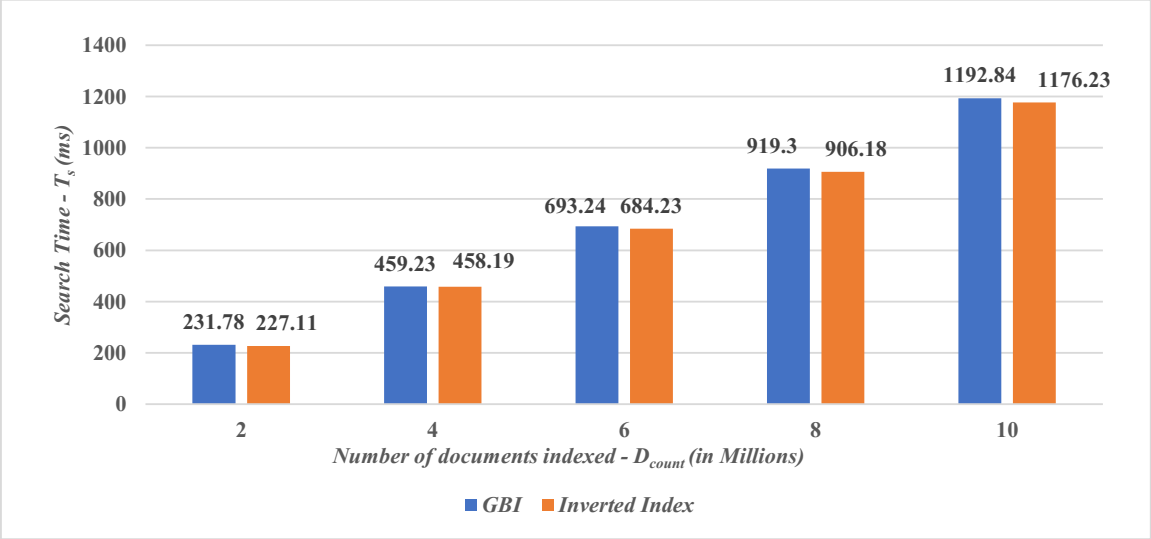


Figure 4.19: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords for a Boolean OR query.

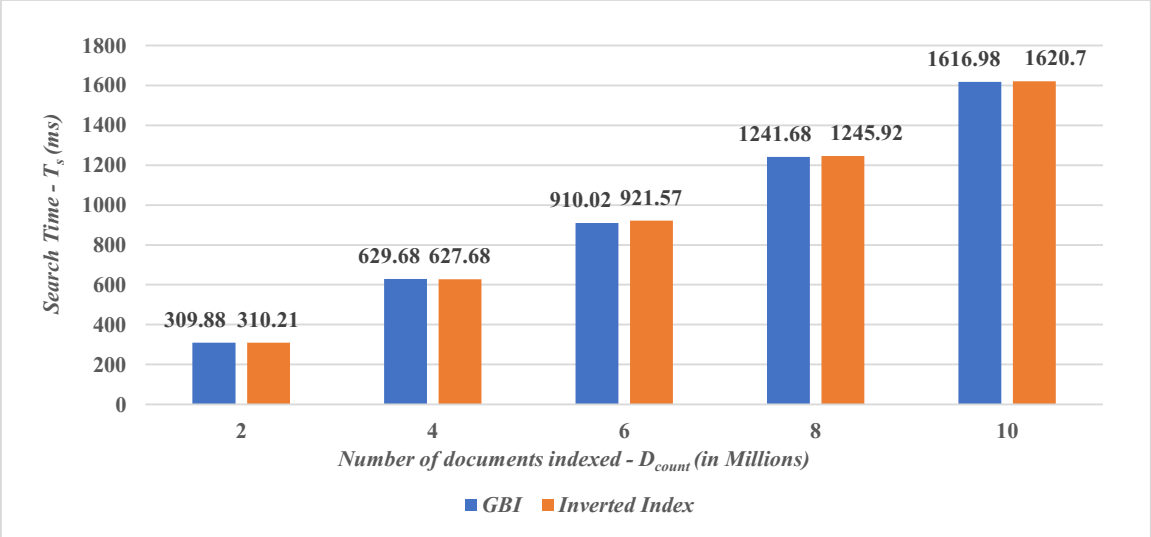


Figure 4.20: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean OR query.

4.4 Experiments for Neighbours of a Keyword Queries

4.4.1 Analysis of the Search Latency Using Random Workload

The results of Experiment A from the random workload for the neighbours of a keyword query can be seen in Figure 4.21. In GBI, the search time for the neighbours of a keyword query does not depend on the number of documents indexed rather on the number of keywords found in GBI. In other words, the search time for GBI depends on the number of incoming and outgoing edges incident on/from the search keyword node in the graph structure of GBI. According to the random workload described in Section 4.2.1.1, there can be 9 keywords that can be connected to the search keyword for any given value of D_{count}. Since the number of neighbours of the search keyword remains the same for different values of D_{count}, the search time in GBI also remains the same. For Inverted Index, the search time depends on the total number of keywords found in Inverted Index as well as how soon the keywords co-occur with the search keyword in the indexed documents. For the random workload, by checking the documents generated for different values of D_{count}, it is found that the 10 keywords in the K_{pool} used to generate the documents are co-occurring with each other within the first few documents itself. As the keywords found in the Inverted Index are co-occurring soon with the search keyword for different values of D_{count},

the search time remains the same. Since the number of edges (hash table entries) to check in the case of GBI and the number of document ids to compare in the case of Inverted Index is almost similar, the search time obtained by using both GBI and Inverted Index appears to be the same for any given value of D_{count}.

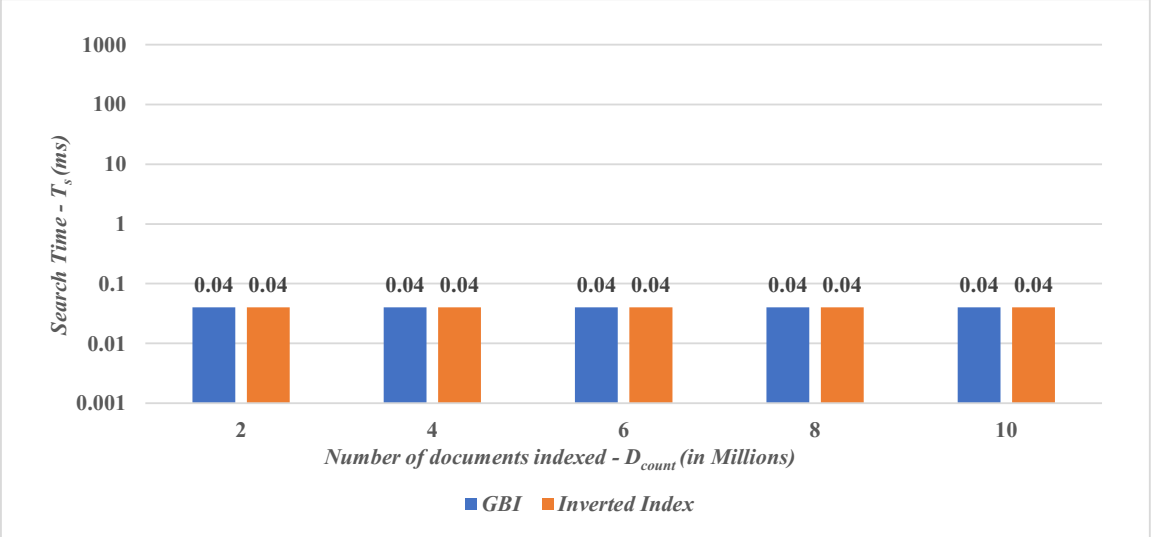


Figure 4.21: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a neighbours of a keyword query.

4.4.2 Analysis of the Search Latency Using Workloads for Neighbours of a keyword Queries

In all the four experiments conducted in this category, GBI performs significantly better than Inverted Index with an average overall improvement of 99%.

For Experiment A, Figure 4.22 shows that T_s for GBI is constant around 0.18 ms. This shows that the point of neighbourhood (P_n) does not affect the time needed for

the execution of neighbours of keyword queries in GBI. This is because GBI checks whether two keywords are neighbours by using a hash table lookup for the existence of keys representing the keywords. Thus, checking for the neighbourhood in GBI is not based on the ids of the documents in which the keywords occur together. In the case of Inverted Index, as P_n increases, the search time also increases. This is because Inverted Index does not have the advantage of a hash table lookup for finding whether two keywords are neighbours. It has to iterate through all the document ids of the postings list of one of the keywords to find which document id is matching with the document id of the other keyword's postings list. This means that sooner the search keyword and neighbouring keywords occur together in the document indexed, faster the result is obtained for Inverted Index.

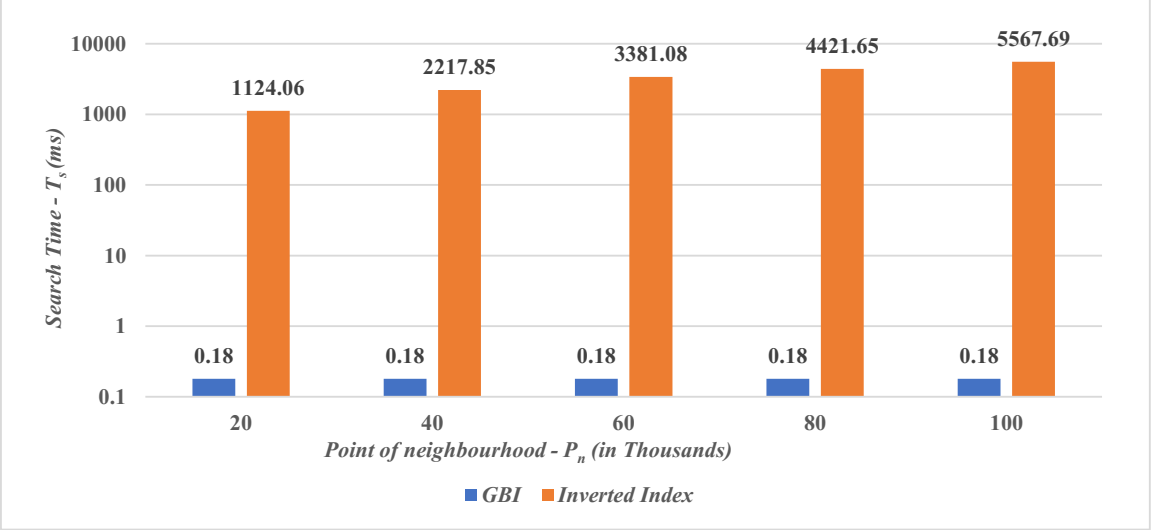


Figure 4.22: Evaluation of search time (T_s) in GBI and Inverted Index when P_n is varied from 20K to 100K, keeping KI_{count} and N_{count} fixed to 2K for a neighbours of a keyword query.

For Experiment B, Figure 4.23 shows that T_s for GBI is increasing with an increase in N_{count} but it still has a negligibly smaller value in comparison to Inverted Index for any given N_{count}. This is again because GBI takes advantage of hash table lookup to check whether two keywords are neighbours. As the number of keywords to check for neighbourhood increases, the number of hash table lookups also increases leading to an increase in search time. Since hash table lookup takes a very small time to check for an entry in the hash table, the increase in search time is not significant in comparison to Inverted Index. In the case of Inverted Index, to determine whether more keywords as neighbours or not, more postings lists comparisons have to be made, resulting in very high search time.

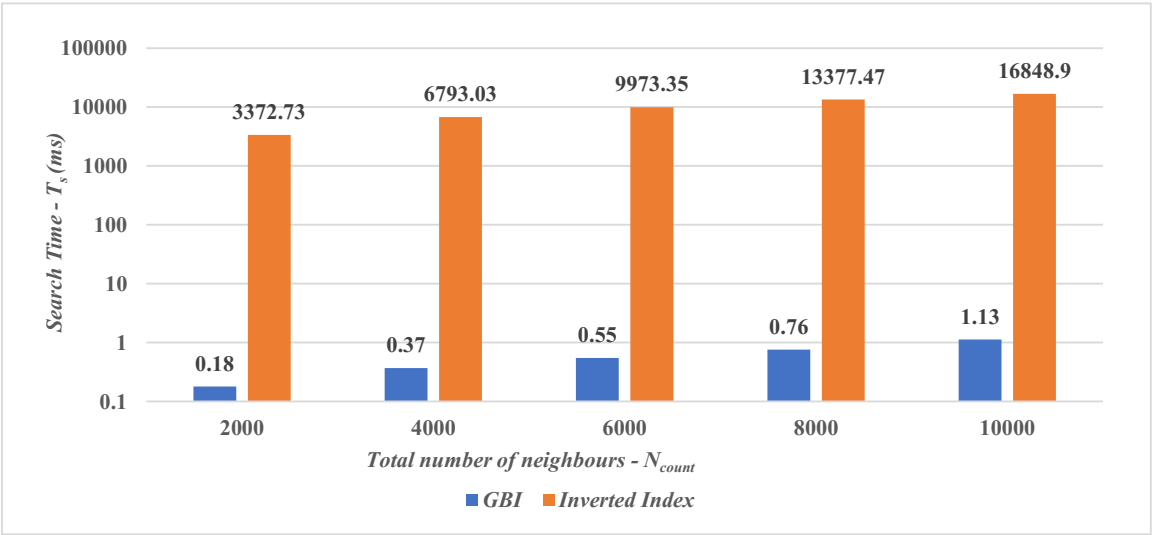


Figure 4.23: Evaluation of search time (T_s) in GBI and Inverted Index when N_{count} and KI_{count} is varied from 2K to 10K, keeping P_n fixed to 60K for a neighbours of a keyword query.

For Experiment C, Figure 4.24 shows that T_s achieved by GBI for any given

value of KI_{count}, is constant around 0.2 ms. In this case, all the neighbours of the search keyword appear lexicographically later than the search keyword. As per the GBI structure, all the neighbours which appear alphabetically later than the search keyword appears in the search keyword's hash table. Hence, a simple traversal of keys on the search keyword's hash table gives the result. Since the number of neighbours remains at the constant value of 2000, the number of neighbours found in the search keyword's hash table will be the same for different values of KI_{count}. Hence, the time it takes to traverse the 2000 keys is the same for different values of KI_{count}. Thus, if all the neighbours of a search keyword appear after the search keyword in GBI, the total number of keywords found in the index (KI_{count}) does not affect the search time for finding the neighbours of a keyword using GBI.

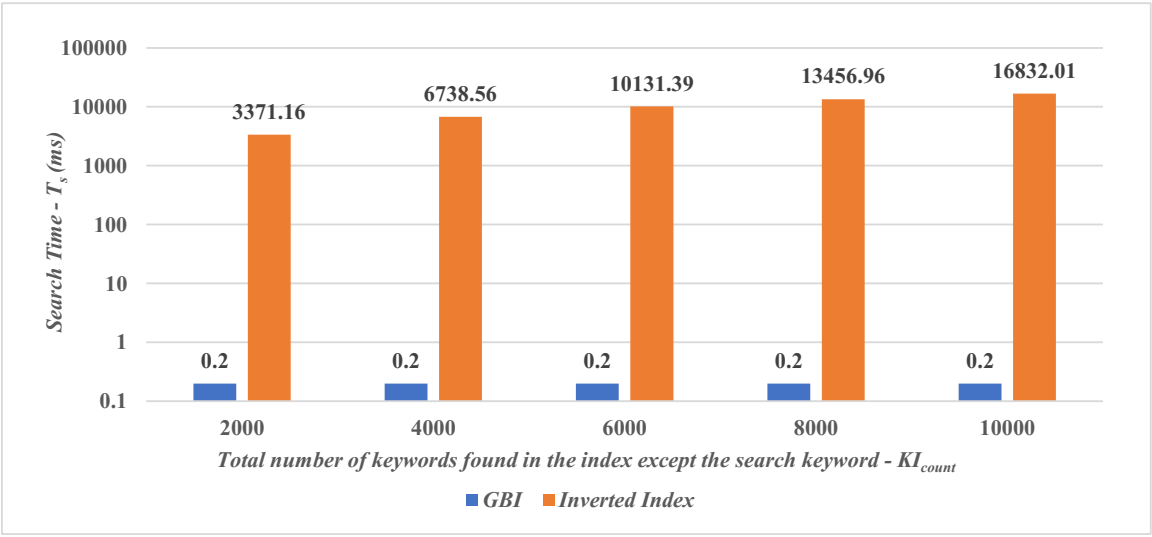


Figure 4.24: Evaluation of search time (T_s) in GBI and Inverted Index when KI_{count} is varied from 2K to 10K such that all the keywords occur after the search keyword in GBI, keeping P_n fixed to 60K and N_{count} to 2K for a neighbours of a keyword query.

For Experiment D, Figure 4.25 shows that T_s for GBI is increasing in much smaller value in comparison to Inverted Index. Since all the keywords found in the index appear alphabetically earlier than the search keyword, GBI has to visit all keywords hash table to check whether the search keyword is present or not. As the total number of keywords found in the index grows, the number of hash table to check also increases. Therefore, there is an increase in search time. In this case, KI_{count} affects the query search time for GBI but the hash table lookup time is so small it is still much better than that for Inverted Index.

In the case of Inverted Index, it does not matter whether all the keywords appear before or after the search keyword in the index structure. All the keywords postings lists have to be compared to the search keyword's postings list to find out the neighbours. Thus, an increase in KI_{count} increases the number of keywords postings lists to be compared, increasing the overall search time for Inverted Index.

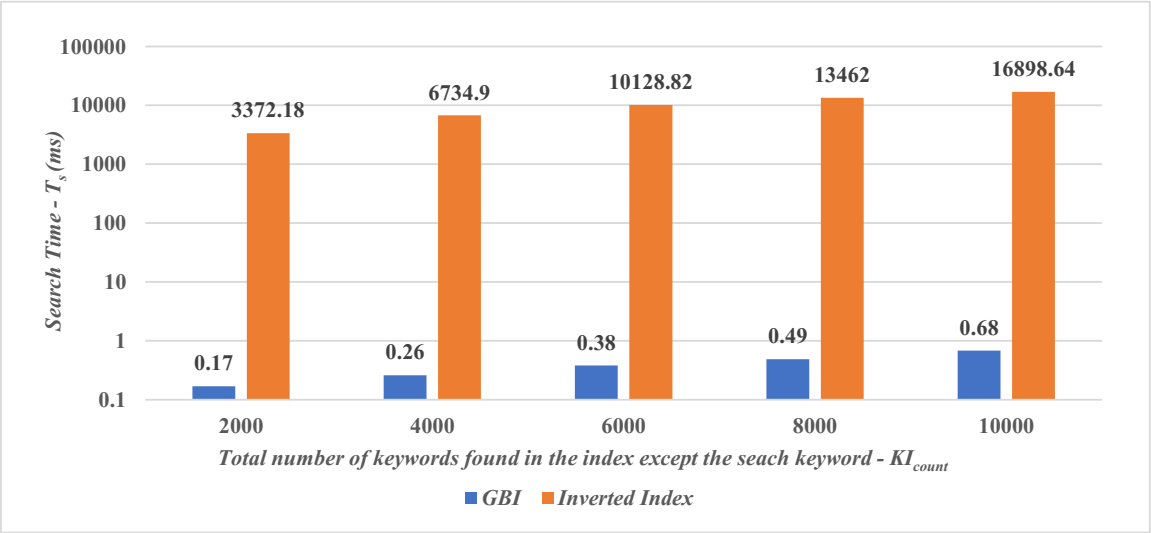


Figure 4.25: Evaluation of search time (T_s) in GBI and Inverted Index when KI_{count} is varied from 2K to 10K such that all the keywords occur before the search keyword in GBI, keeping P_n fixed to 60K and N_{count} to 2K for a neighbours of a keyword query.

4.5 Experiment for Exclusive Keyword Queries

4.5.1 Analysis of the Search Latency Using Random Workload

In the experiment for evaluating the performance of exclusive keyword queries using GBI and Inverted Index, the number of documents indexed D_{count} has been increased from 2M to 10M in steps of 2M. The exclusive keyword queries return the documents in which only the search keyword (*Anthony*) appears, and no other keywords found in the index appear together with the search keyword in those documents. The search

time for the execution of exclusive keyword query using GBI and Inverted Index is shown in Figure 4.26. As can be inferred from the figure, T_s for GBI is on average around 0.01 ms. This low search latency is due to the fact that exclusive documents are stored separately for each search keyword in the underlying graph structure of GBI. The exclusive documents for *Anthony* can be obtained by looking up *Anthony*'s hash table. Since a single hash table lookup is enough to find the exclusive documents for a search keyword, the search time is constant and does not depend on D_{count}. In the case of Inverted Index, the search keyword's postings list should be compared with all the remaining postings lists in Inverted Index until all the common documents are removed. The remaining document ids are returned as the results. As D_{count} increases, the size of the search keyword's postings list also increases. This in turn increases the number of document ids to be compared resulting in increased search latency. Thus, for this workload an overall improvement of 99.9% over Inverted Index is achieved by GBI.

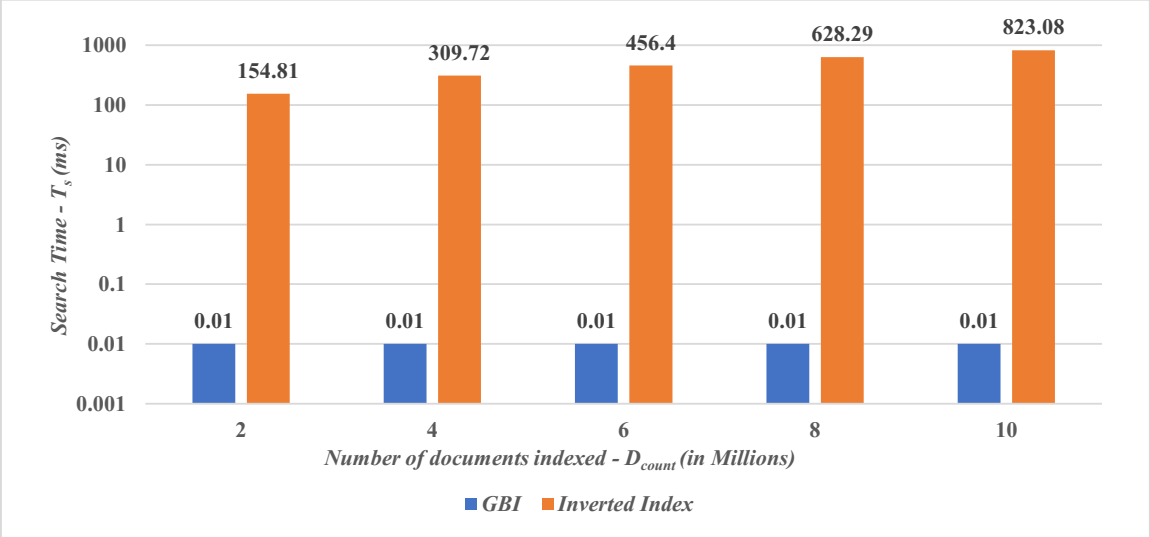


Figure 4.26: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Inverted Index for a exclusive keyword query.

4.6 Experiments for Analyzing the Indexing Performance in GBI, Pruned GBI, and Inverted Index

4.6.1 Workload for Analyzing the Indexing Performance

The random workload described in Section 4.2.1.1 is also used here for comparing the indexing performance of GBI, Pruned GBI, and Inverted Index. The size of the keyword pool ($K_{poolsize}$) is fixed to 10 and number of documents indexed (D_{count}) is varied from 2 to 10 million. The goal of the experiment is to measure time and

memory consumed by GBI, Pruned GBI, and Inverted Index when the number of documents indexed increases.

4.6.2 Analysis of the Indexing Performance

The time and memory consumed by all three indices are shown in Figure 4.27 and Figure 4.28 respectively.

It is observed that Inverted Index is performing better than the other two techniques (GBI and Pruned GBI) both in terms of indexing time and memory usage. This is evident as in Inverted Index, each keyword extracted from a document is stored individually with its postings list. But in the case of GBI and Pruned GBI, relationships are established in pairs of two keywords. Thus, there will be a postings list for every such keyword pairs. For n keywords (nodes), the number of edges needed in Pruned GBI is nC_2 (n choose 2). For GBI, it is nC_2 plus one for the edge representing the self-loop of the node. In addition to these edges, for all the keyword nodes, GBI also collects documents ids from all its edges (except the one for self-loop) and stores it in the node. GBI and Pruned GBI take time to create and form connections between these keywords and hence the increase in time and memory. For any given D_{count}, T_i and M_i achieved with GBI and Pruned GBI is higher than that of achieved with Inverted Index.

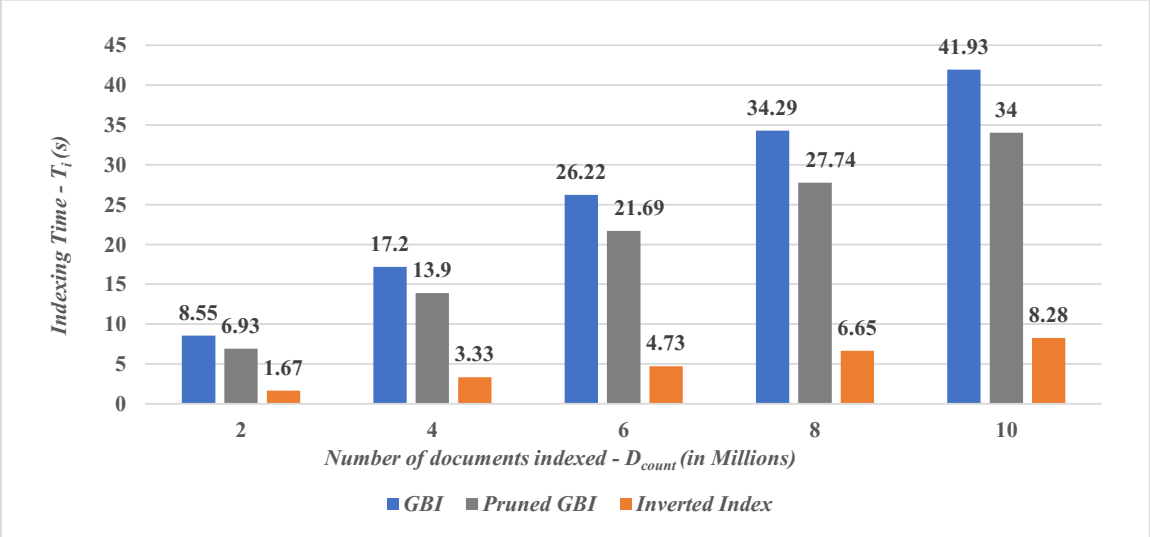


Figure 4.27: Effect of number of documents indexed (D_{count}) on the indexing time (T_i) for GBI, Pruned GBI, and Inverted Index.

From Figure 4.28, it can also be noted that the memory usage is the same for the Pruned GBI for D_{count} ranging from 6M to 10M. This can be explained in terms of hash table resizing. To accommodate more data, the hash table uses strategies like doubling or increasing the size in the power of two to avoid resizing often [33]. This exponential increase in size provides the capacity for more data. It is because of the same reason, there is a huge size difference between the Pruned GBI and Inverted Index at D_{count} = 6M. Also, it can be observed that the memory consumed by GBI is almost equal to the sum of memory consumed by Pruned GBI and Inverted Index. The difference between GBI and Pruned GBI is that GBI also stores the document ids in the keyword node and the keyword node's self-loop. The document ids found in the keyword node and its self-loop when combined forms a set of document ids that are similar to the one found in the postings list of the same keyword found in

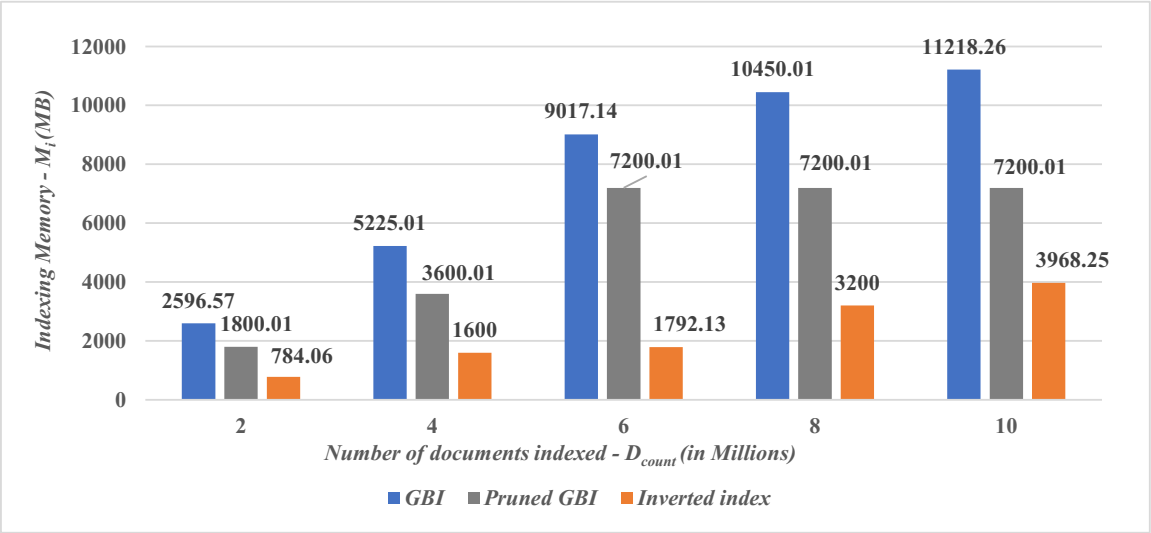


Figure 4.28: Effect of number of documents indexed (D_{count}) on the indexing memory (M_i) for GBI, Pruned GBI, and Inverted Index.

Inverted Index. Thus, GBI is essentially a combination of Inverted Index and Pruned GBI. The M_i obtained for GBI demonstrates this fact.

Thus, the significant reduction in search time of GBI and Pruned GBI comes at the cost of spending more time and memory for performing the indexing operations. GBI is nevertheless preferable to Inverted Index in the common use case where the number of times an indexing operation is performed is far less frequent than the search operations.

4.7 Performance Comparison between GBI and Elasticsearch

Elasticsearch is a popular search engine built using Apache Lucene[1]. It is used as an important back-end component in tech companies like Netflix and Tinder [79]. It provides full-text searching capabilities and uses Inverted Index as its index structure. In the remainder of this section, the performance of GBI is compared with that of Elasticsearch for Boolean queries.

4.7.1 Elasticsearch Setup

The Elasticsearch version 7.6[2] is used for experimentation. The heap memory required for the Elasticsearch instance to run is set to 30 Gigabytes as recommended[3]. Elasticsearch-PHP client API is used to send requests to and receive responses from Elasticsearch. OpenJDK version 11 is used in the system and G1 garbage collection is enabled for Elasticsearch. Memory for Elasticsearch is allocated in such a way that more memory has been allocated to the younger generation of JVM heap so that minor garbage collection is avoided [80]. In addition to this setup, Elasticsearch log file is also continuously monitored for any major garbage collection activity when executing a search query. If the garbage collection activity is observed in the log, then the Elasticsearch is shut down and restarted to completely wipe the heap memory

[1] https://www.elastic.co/what-is/elasticsearch
[2] https://www.elastic.co/downloads/elasticsearch
[3] https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html

allocated for Elasticsearch. Then, the search query is executed again on the restarted Elasticsearch. This ensures that the garbage collection does not affect the search time.

Elasticsearch is a distributed search engine. Hence, index in Elasticsearch is usually split into smaller units known as shards that are distributed across multiple computing nodes. Replicas are the duplicates of shards created for redundancy purposes [81]. Since Elasticsearch is used for performance evaluation, it is deployed in single computing node with only one shard and no replicas. Elasticsearch, by default, stores document id, keyword frequencies, and position of the keyword in the document in its Inverted Index. Since we are executing only Boolean queries, Elasticsearch is configured to store only document ids in its Inverted Index.

Elasticsearch, by default, enables scoring of the results and returns only the top ten results. The results also contain source data from which the keywords are extracted. For the experiments performed in this research, the scoring of the document is disabled and Elasticsearch is configured to return all the matching document ids at once ignoring the source data. Caching of the search results is also disabled in Elasticsearch to get consistent results while running an experiment multiple time. Indexing of documents in Elasticsearch is done in bulk with refresh interval disabled for better indexing performance[4].

[4]`https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-indexing-speed.html`

4.7.2 Performance Metrics

The search time (T_s), indexing time (T_i) and indexing memory (M_i) metrics defined in Table 4.4 are used to compare the performance of GBI and Elasticsearch. For communicating with Elasticsearch, Elasticsearch-PHP client serializes the request into JSON format and sends the request to the port on which Elasticsearch is running where the request message is obtained from de-serializing the JSON data. Again, once the response is ready in Elasticsearch, the response message is serialized back into JSON format and sent back to the client where the JSON data is de-serialized to get the response message. When the *hrtime* function provided in PHP is used to measure the search and indexing performance of Elasticsearch, the resulting time value includes the overhead time spent for communicating with Elasticsearch and other activities occurring within Elasticsearch. To avoid the additional overhead in impacting the search and indexing time measurements, a set of info APIs provided by Elasticsearch is utilized. The info APIs provides a detailed report on the time and memory consumed when handling a search or indexing request.

For measuring a query execution time, Elasticsearch has provided Profile API for obtaining time consumed by each component of the search operation[5]. Hence, for Elasticsearch, T_s is the time consumed for a Boolean query execution reported in the "BooleanQuery" field of the Elasticsearch Profile API response. The search time obtained from Elasticsearch is in nanoseconds and it is converted to milliseconds for reporting in this thesis. Similarly, Elasticsearch has provided Index stats API for

[5] `https://www.elastic.co/guide/en/elasticsearch/reference/current/search-profile.html`

measuring indexing time and memory consumed[6]. Hence, for Elasticsearch, T_i is the time value reported in "index.time_in_millis" field of Index stats API response. The indexing time obtained from Elasticsearch is in milliseconds and it is converted to seconds for reporting in this thesis. M_i for Elasticsearch is the memory value reported in "store.size_in_bytes" field of Index stats API response. The memory value obtained from Elasticsearch is in bytes and it is converted to megabytes for reporting in this thesis. For GBI, M_i represents the size of the compressed GBI index file which contains data in JSON format (discussed in Section 4.1).

4.7.3 Analysis of the Search Latency in Elasticsearch and GBI

A representative set of experiments based on random and relationship-based workloads (see Section 4.2.1) are used for analyzing the performance of Elasticsearch. Elasticsearch search latency is analyzed only for Boolean queries. To the best of my knowledge, exclusive and neighbours of a keyword queries are not supported by Elasticsearch. Thus, the workload parameters concerning the Boolean queries presented in Table 4.1 and Table 4.2 are used in this set of experiments to investigate the search performance of Boolean queries in Elasticsearch and compare it with to that obtained with GBI. The selected set of experiments are:

- Experiment A based on random workload discussed in Section 4.2.1.1.

[6] `https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-stats.html`

- *Partial* and special case of *Full* relationship experiments based on relationship-based workloads discussed in Section 4.2.1.2.

4.7.3.1 Analysis of the Search Latency for Boolean AND Queries

Figure 4.29 shows the search time (T_s) for GBI and Elasticsearch for Experiment A associated with random workload. Though Elasticsearch is equipped with the skip list (discussed in Section 2.1.2.1) to speed up the postings lists traversal for intersection operation, the reduction in the number of postings lists needed for intersection operation in GBI (as keywords are indexed in pairs) leads to a smaller search time. For this experiment, GBI shows a search time that is, on average, 50% lower than that of Elasticsearch.

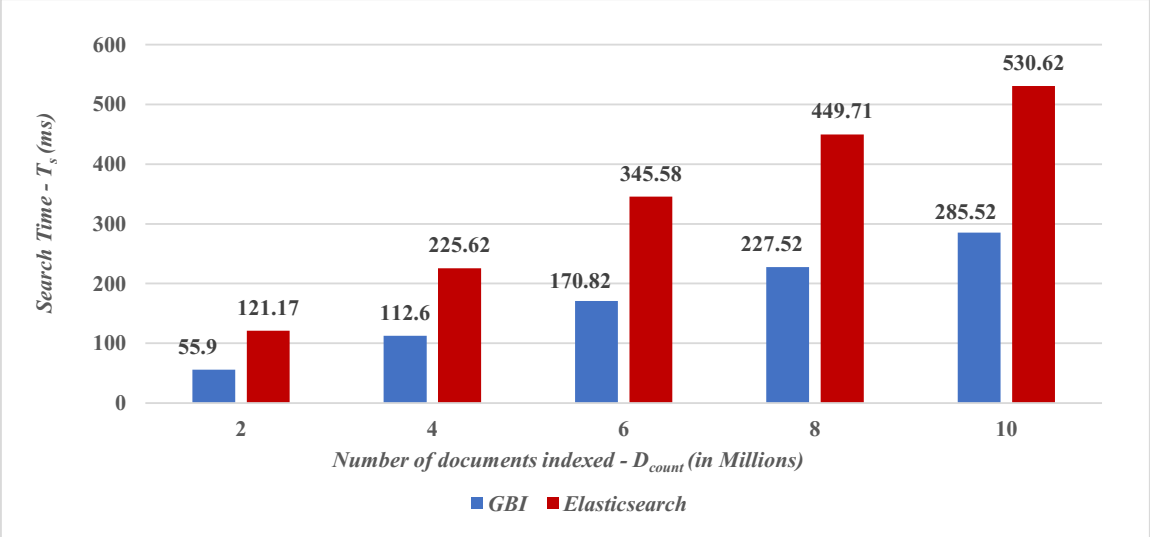


Figure 4.29: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean AND query.

Figure 4.30 shows the search time (T_s) for GBI and Elasticsearch for the *Partial* relationship experiment. For a given D_{count}, T_s for Elasticsearch is higher than GBI. Though Elasticsearch was able to reduce the search time using skip list, the hash table lookup for connection checking before an intersection operation resulted in negligibly smaller search time for GBI. The performance improvement achieved by GBI over Elasticsearch in this experiment is 99.9%.

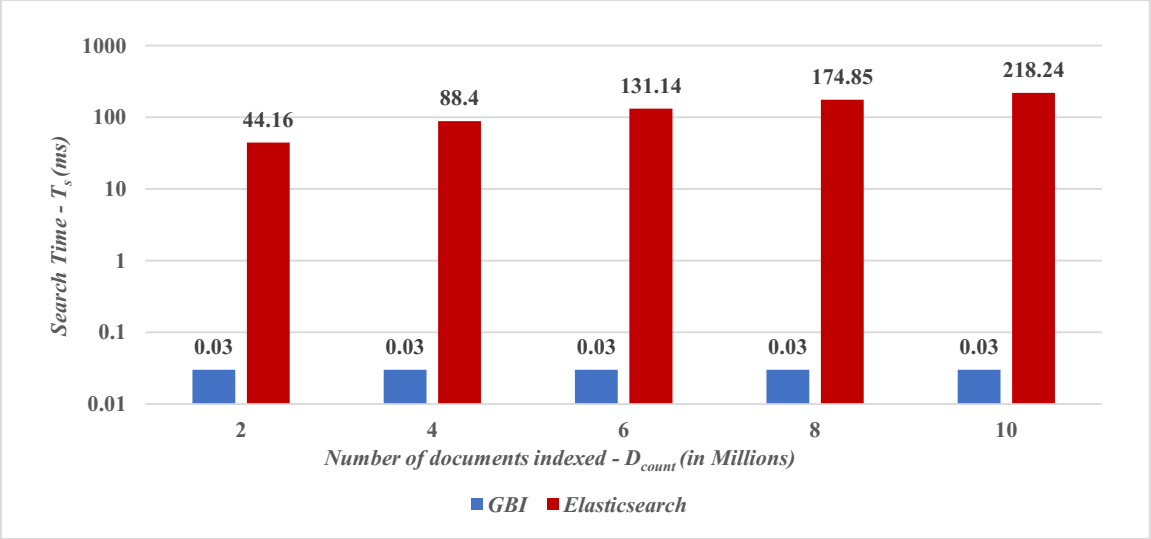


Figure 4.30: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Partial* relationship among search keywords for a Boolean AND query.

Figure 4.31 shows the search time (T_s) for the special case of the *Full* relationship experiment in which all the search keywords occur in all the documents indexed. In this experiment also, for any given value of D_{count}, Elasticsearch shows a higher search time than GBI. This is because all the search keywords appear in all the documents indexed. Hence, more number of document id comparison has to be performed to

execute the query. The presence of skip list in Elasticsearch does not seem to offer much help for this workload as none of the skip pointers can be avoided. In GBI, the lesser number of intersection operations leads to a lower search time. In this experiment, the performance improvement for GBI over Elasticsearch is around 70%.

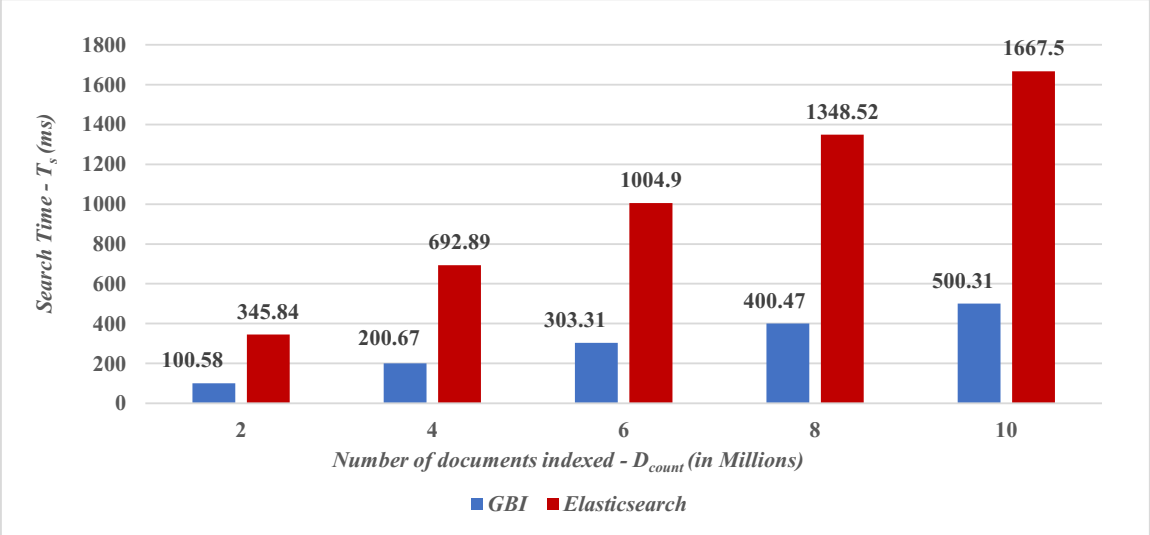


Figure 4.31: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean AND query.

4.7.3.2 Analysis of the Search Latency for Boolean NOT Queries

As explained in Section 3.5.2.1, a Boolean NOT query returns a set of documents in which a particular search keyword (S_k) is present and set of search keywords (E_1, E_2, ..., E_k) are not present.

Figure 4.32 shows the search time (T_s) for GBI and Elasticsearch for Experiment

A associated with random workload. As inferred from Profile API, Elasticsearch is performing Boolean NOT query in two steps. In the first step, a Boolean OR operation is performed between the search keywords E_1, E_2, ..., E_k to get a list of document ids. In the second step, the resultant document ids for the Boolean OR query operation are removed from the postings list of S_k. This two-step execution of Boolean NOT queries make Elasticsearch search time much higher than that of GBI when retrieving all the matching document ids at once. In the case of GBI, document ids of all the search keywords that are not to be present (E_1, E_2, ..., E_k) are known *apriori* as keywords are always indexed in pairs in GBI. Thus, the common documents (if any) between S_k and E_1, S_k and E_2 and so on are known beforehand and these common documents are directly removed from S_k's postings list. Therefore, an expensive Boolean OR operation is avoided in the case of GBI. This leads to a lower search time for GBI in comparison to that of Elasticsearch. For this experiment, GBI shows a search time that is, on an average, 68% lower than Elasticsearch.

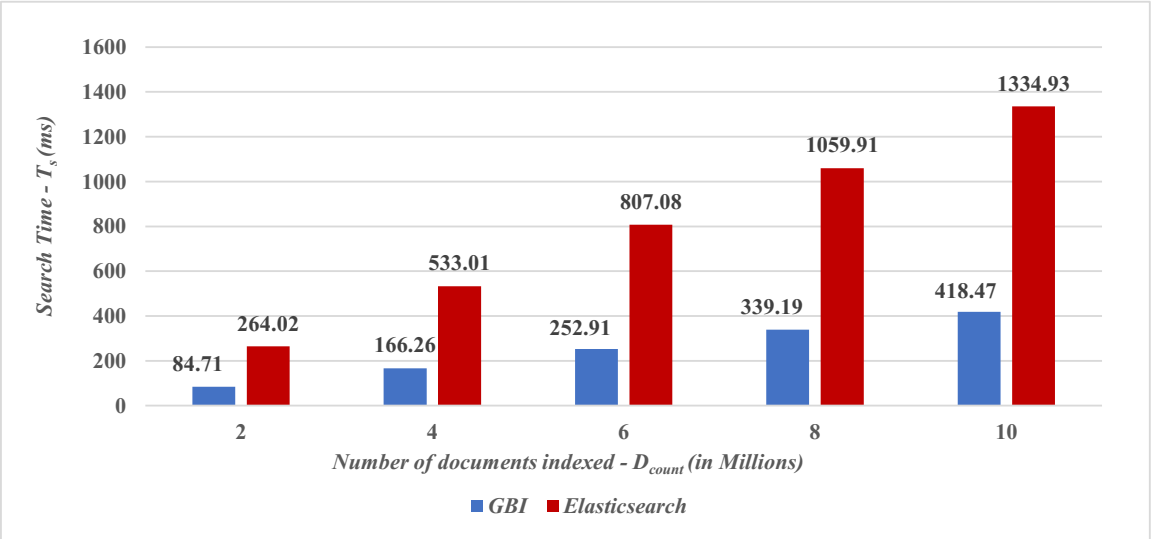


Figure 4.32: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean NOT query.

Figure 4.33 shows the search time (T_s) for GBI and Elasticsearch for the *Partial* relationship experiment. For a given D_{count}, T_s for Elasticsearch is higher than GBI in this experiment as well. GBI uses hash table lookup to check whether the search keywords (E_1, E_2, ..., E_k) are actually the neighbours of S_k. Among the keywords (E_1, E_2, ..., E_k), if only E_1 appear together with S_k then only the documents that are common between S_k and E_1 only are identified and removed. Thus, if none of the keywords in the set (E_1, E_2, ..., E_k) is connected to S_k, then the Boolean NOT operation is avoided. If some of the keywords from E_1, E_2, ..., E_k are connected with S_k, then only the common document ids of the connected keywords are removed. This setup leads to a lower search latency for GBI in comparison to that of Elasticsearch. The performance improvement observed in this experiment for GBI is on an average

85% over Elasticsearch.

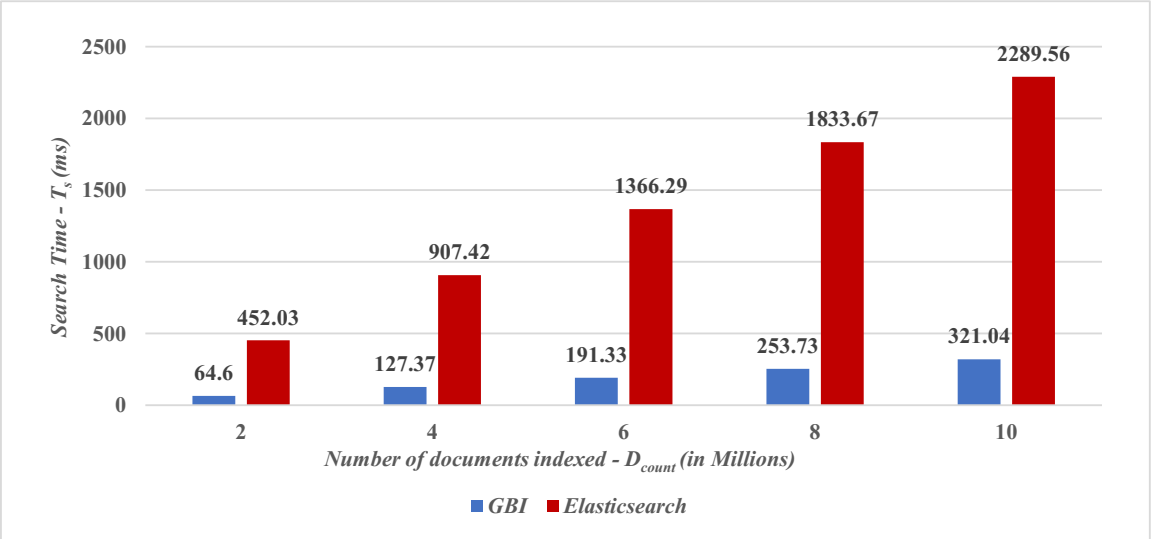


Figure 4.33: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Partial* relationship among search keywords for a Boolean NOT query.

Figure 4.34 shows the search time (T_s) for the special case of the *Full* relationship experiment in which all the search keywords occurs in all the documents indexed. Thus, the search keywords S_k and E_1, E_2, ..., E_k appear together in all the documents indexed. In this experiment as well, Elasticsearch shows a higher search time than GBI. Elasticsearch first performs a Boolean OR operation for E_1, E_2, ..., E_k. This Boolean OR operation results in all the document ids indexed. Then, the resultant document ids have to be removed from S_k's postings list. S_k's postings list also contains the ids of all the documents indexed. Thus, in this case, after the Boolean NOT operation, there will be no document ids left returning a NULL value. As can be seen, due to a lack of knowledge of search keywords relationships, Elasticsearch is

performing a large number of computations to produce the NULL result. But GBI is aware of the relationships among S_k and E_1, E_2, ..., E_k. Also, GBI avoids doing Boolean OR operation and directly removes the duplicate document ids between S_k and E_1 and so on. Thus, by avoiding additional computations, GBI gains a performance improvement of approximately 88% over Elasticsearch in this experiment for any given value of D_{count}.

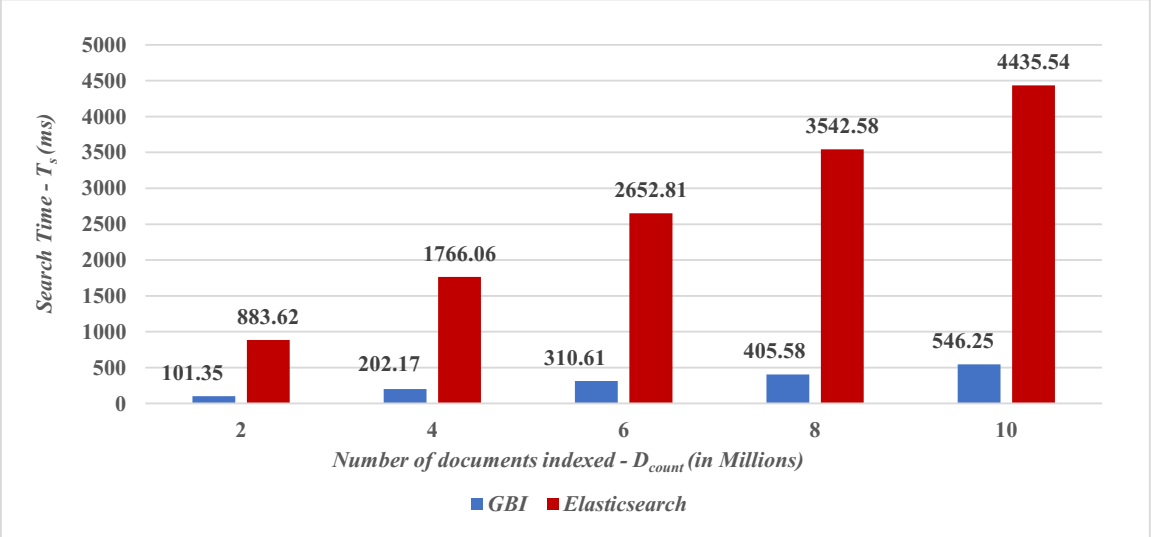


Figure 4.34: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean NOT query.

4.7.3.3 Analysis of the Search Latency for Boolean OR Queries

Figure 4.35 shows the search time (T_s) for GBI and Elasticsearch for Experiment A associated with random workload. The search time for Elasticsearch is much closer

to the search time of GBI. The presence of skip list (discussed in Section 2.1.2.1) in Elasticsearch might leads to an efficient traversal of postings lists for union operation. In GBI, knowing duplicates to be removed in advance while performing a union operation helps maintain lower search time for any given value of D_{count}. For this experiment, GBI shows a search time that is, on an average, 13% lower than Elasticsearch.

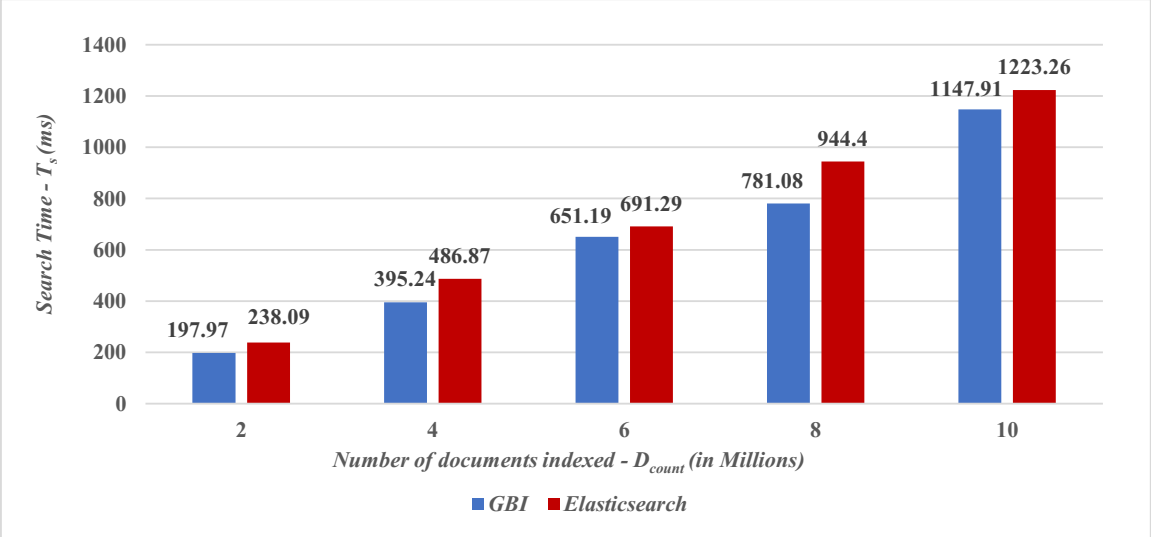


Figure 4.35: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch when the number of search keyword (SK_{count}) is kept at the constant value of 4 for a Boolean OR query.

Figure 4.36 shows the search time (T_s) for GBI and Elasticsearch for the *Partial* relationship experiment. For a given D_{count}, search time for Elasticsearch is higher than that for GBI. The hash table lookup for checking whether search keywords appear together enables GBI to perform union operation only between the search keywords that have a common document id. If there is no common document id between

some search keyword then the expensive union operation involving duplicate removal process is avoided between those search keywords. The performance improvement observed in this experiment for GBI is on an average, 41% over Elasticsearch.

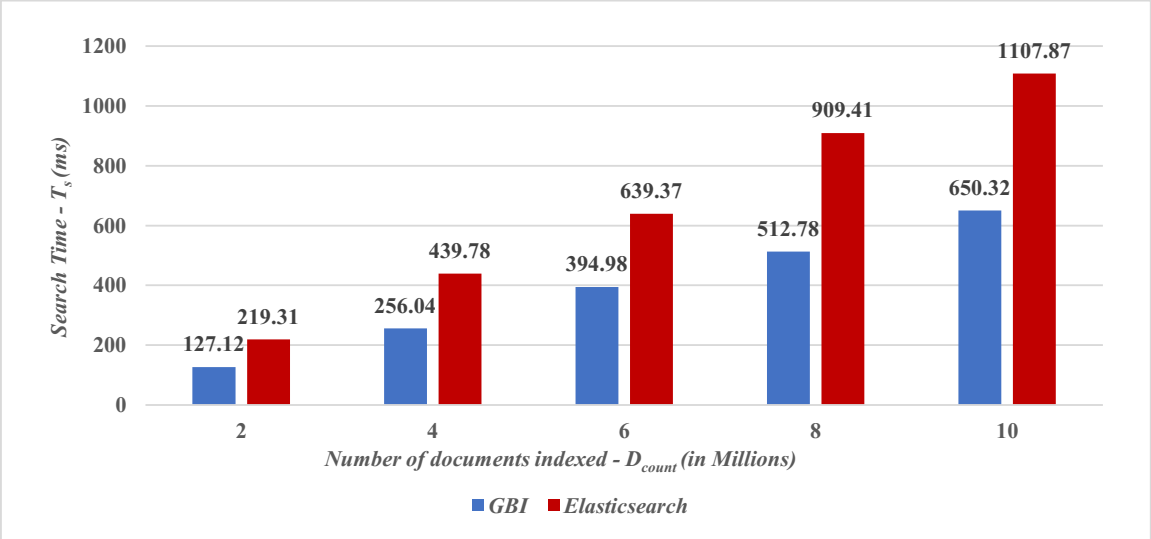


Figure 4.36: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Partial* relationship among search keywords for a Boolean OR query.

Figure 4.37 shows the search time (T_s) for the special case of the *Full* relationship experiment in which all the search keywords occurs in all the documents indexed. Similar to *Partial* relationship experiment, this experiment also leads to a much higher search time for Elasticsearch in comparison to GBI for any given values of D_{count}. This is because all the search keywords appear in all the documents indexed. This results in all the document ids to be identified as duplicates while performing union operation. Since there are 4 search keywords, three times all the document ids has to be removed. This result in higher search time in GBI and Elasticsearch.

Though the number of document ids and the number of times it has to be removed remains the same for both GBI and Elasticsearch, GBI still performs better than Elasticsearch. The possible reason could be Elasticsearch might be more optimized for retrieving small set of documents at a time. Since all the matching documents are retrieved at once, the performance suffers in Elasticsearch. In this experiment, the performance improvement for GBI over Elasticsearch is on an average around 59%.

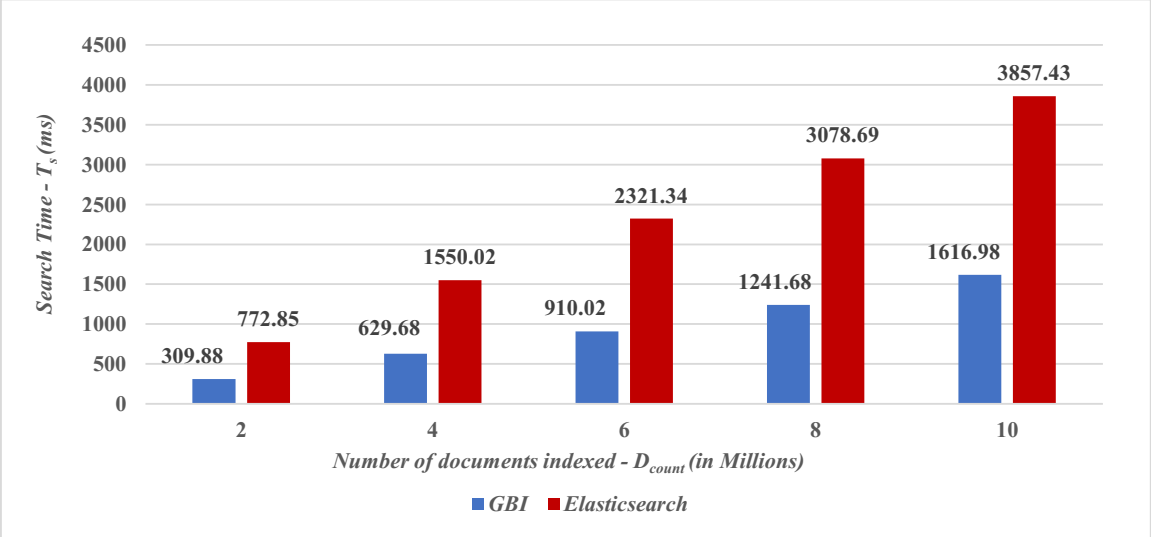


Figure 4.37: Effect of number of documents indexed (D_{count}) on the search time (T_s) in GBI and Elasticsearch for a workload with *Full* relationship among search keywords in which all the search keywords appear in all the documents indexed for a Boolean OR query.

4.7.4 Comparison of the Indexing Performance between GBI and Elasticsearch

For the comparison of indexing time and memory consumed by GBI and Elasticsearch, the random workload described in Section 4.2.1.1 is used.

Figure 4.38 shows the indexing time (T_i) for the random workload where the number of documents indexed (D_{count}) is increased from 2M to 10M in steps of 2M. For a given D_{count}, the indexing time for Elasticsearch is higher than GBI. In the proposed experimental prototypes, the time taken for indexing the keywords alone is considered. The indexing time value provided by Elasticsearch is the total time taken for indexing and does not provide the time consumed by the individual component of the indexing operation. Therefore, the indexing time reported by Elasticsearch may include time for processing the request, analyzing the keywords, and then indexing in its Inverted Index. These may contribute to a higher T_i for Elasticsearch. To the best of my knowledge, Elasticsearch does not provide the time for an individual component of the indexing operation. Thus, further investigation is warranted for detailed analysis of indexing time in Elasticsearch.

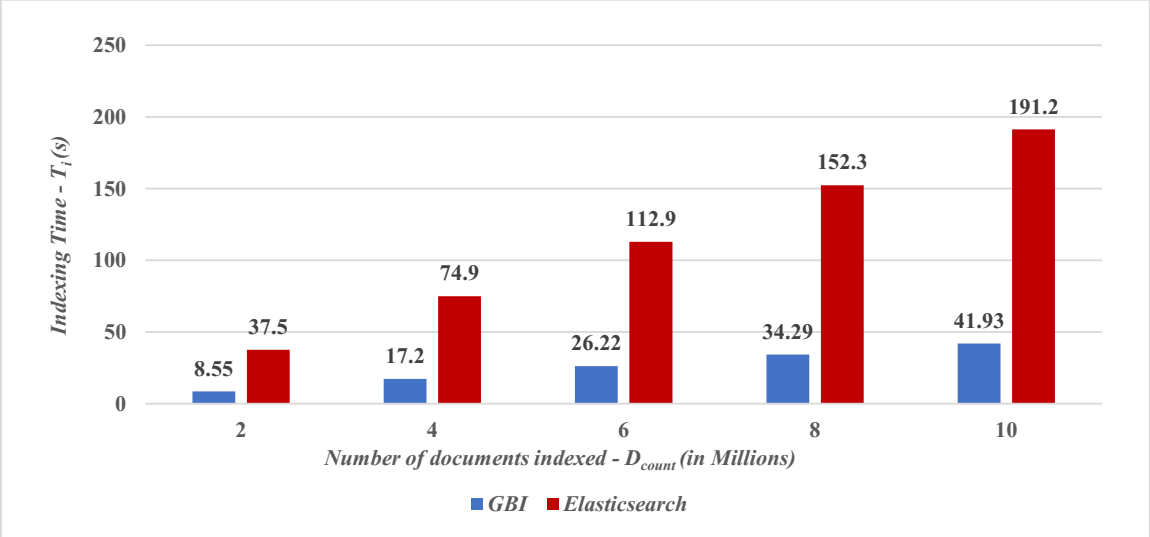


Figure 4.38: Effect of number of documents indexed (D_{count}) on the indexing time (T_i) for GBI and Elasticsearch.

Figure 4.39 shows the memory (M_i) consumed by Elasticsearch and the compressed GBI index file. It can be inferred from the figure that for any given value of D_{count}, M_i for Elasticsearch is much lower in comparison to GBI. This is because Elasticsearch compresses and stores its index in a disk. Elasticsearch uses a lossless compression algorithm called LZ4 to compress its data[7] [8]. We tried using the LZ4 compression algorithm (with default options) on the index file containing GBI data in JSON format. The result of compression on GBI index file for different values of D_{count} can be seen in Table 4.5. On average, approximately a 50% reduction in GBI file size is achieved by using the LZ4 compression algorithm. Although the memory consumed by the compressed GBI index file is almost half of the original file size, it is still

[7] https://www.elastic.co/blog/store-compression-in-lucene-and-elasticsearch
[8] https://en.wikipedia.org/wiki/LZ4_(compression_algorithm)

Table 4.5: Original and Compressed GBI File Size

Number of Documents Indexed	Original GBI File Size	Compressed GBI File Size
2M	708.2 MB	361.11 MB
4M	1463.54 MB	722.71 MB
6M	2220.24 MB	1084.91 MB
8M	2975.77 MB	1447.26 MB
10M	3729.81 MB	1808.67 MB

around 10 times greater than the equivalent Elasticsearch index file. To the best of my knowledge, in Elasticsearch, the format of the data, the strategy for compressing the data stored in the index file is not described in detail. Thus, these factors might be some of the reasons for the large difference in memory consumption between GBI and Elasticsearch. Further investigation is warranted for more detailed analysis of memory consumption by Elasticsearch.

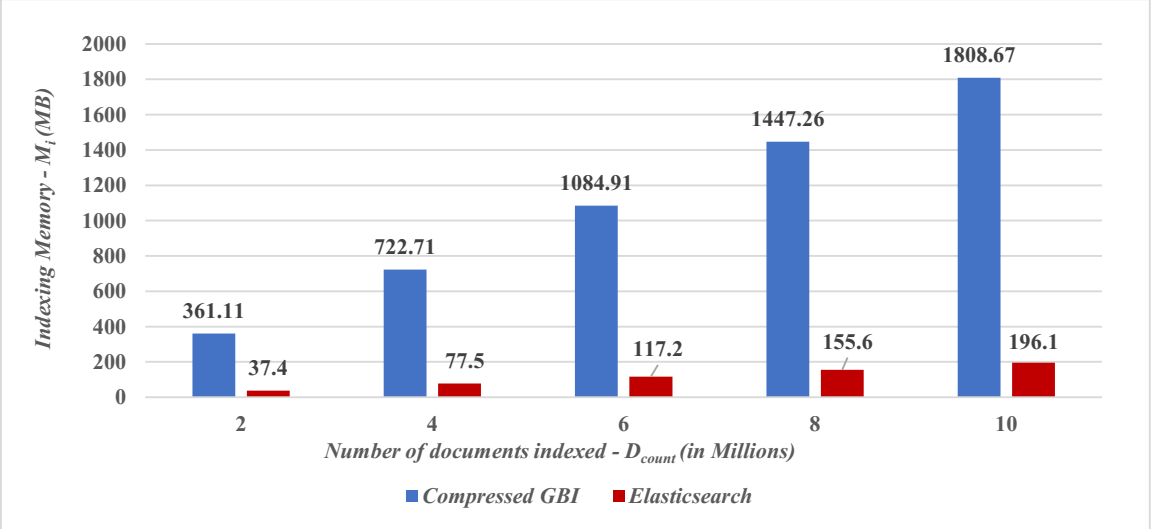


Figure 4.39: Effect of number of documents indexed (D_{count}) on the indexing memory (M_i) for compressed GBI and Elasticsearch.

Chapter 5

Conclusions and Future Work

In this chapter, the synopsis of the concepts presented in this thesis, and a summary of the performance comparison between GBI and Inverted Index are provided. Finally, the limitations of the presented research and directions for future work are discussed.

5.1 Synopsis of the Concepts Presented

Indexing is one of the efficient techniques used for accessing the data faster. The challenge involved in constructing an index structure depends on the type of search query that is being executed and the data that is being indexed.

Inverted Index is a popular text indexing technique used in web search engines like Google [9] and enterprise search engines like Elasticsearch [10]. The Inverted Index structure is described in detail in Section 2.1.1. Section 2.1.2.1 describes various optimization techniques used for executing Boolean queries using Inverted Index.

Most of the works are done on retrieving top K results only by using ranking or scoring methods. A hash table is the data structure that provides constant lookup time irrespective of the number of elements stored in it. The construction and working of the hash table are described in Section 2.2. In general, not all the words found in a document are used for indexing. The different strategies used for selecting potential keywords for efficient indexing are discussed in Section 2.3.

The primary goal of the proposed research is to use a graph-based indexing technique to improve the search performance of various queries. The proposed index structure (GBI) uses a directed graph to represent keywords that appear together in a document. A brief overview of graph theory and its application in textual indexing is discussed in sections 2.4 and 2.4.2.1 respectively. The structure of the directed graph used in the proposed indexing technique is discussed in Section 3.1. Section 3.3 details how the graph structure represented by a modified adjacency matrix is built using the hash table.

The proposed graph-based indexing technique aims to retrieve all the matching results at once which is the primary motivation for the proposed approach. Apart from document ids, other details such as word frequency, the position of the word in the document can also be stored in the postings lists of GBI. Thus, the different details stored for a document in Inverted Index can also be stored in GBI. This allows an effective algorithm for executing Boolean queries using Inverted Index to be applied to systems using GBI as well. Also, due to the graph structure of GBI, queries like neighbours of a keyword and exclusive keyword (described in Section 3.6.1

and Section 3.6.2) are executed efficiently.

The search latency for various query execution is analyzed for GBI and Inverted Index by varying different workload parameters described in Section 4.2.1. GBI is also compared with Elasticsearch, a real-world Inverted Index based system in Section 4.7.

5.2 Performance Insights

A thorough performance analysis of the prototype systems (GBI, Pruned GBI, and Inverted Index) is described in Chapter 4. A number of synthetic workloads, as described in Section 4.2.1, are used in the performance analysis. The key insights from the experiments are summarized next.

- For the random workload in which the number of documents indexed is increased from 2 to 10 million in steps of 2 million while the number of search keywords is kept at a constant value of 4, the search time obtained using GBI is 70%, 28%, and 28% lower than those achieved with Inverted Index for Boolean AND, NOT and OR queries respectively (see sections 4.3.1.1, 4.3.2.1 and 4.3.3.1). Thus, GBI is able to maintain its superior performance over Inverted Index when the number of the documents indexed increases.

- For the random workload in which the number of search keywords is increased from 2 to 10 while the number of documents indexed is kept at a constant value of 6 million, the search time obtained using GBI is 62%, 28%, and 28%

lower than that observed for Inverted Index for Boolean AND, NOT and OR queries respectively (see sections 4.3.1.1, 4.3.2.1 and 4.3.3.1), Thus, GBI is able to maintain its superior performance over Inverted Index when the number of search keywords used in the query increases.

- If the search keywords do not occur together in any document indexed (i.e. *No* relationship), the search latency obtained for GBI is 99.9% lower than that obtained with Inverted Index for all the Boolean queries discussed (see sections 4.3.1.2, 4.3.2.2 and 4.3.3.2). Thus, if a set of documents whose contents are highly diverse is indexed, then the Boolean search queries using GBI show significant improvement over Inverted Index.

- If some of the search keywords do not occur together in any document indexed (i.e. *Partial* relationship), the search latency obtained for GBI is on average 99.9%, 30%, and 29% lower than that obtained with Inverted Index for Boolean AND, NOT and OR queries respectively (see sections 4.3.1.2, 4.3.2.2 and 4.3.3.2). Thus, if a set of documents whose contents slightly overlap is indexed, then performing the Boolean search queries using GBI shows a significant improvement over Inverted Index.

- If all of the search keywords occur together in some or all of the document indexed (i.e. *Full* relationship), the search time is 67% better than Inverted Index for Boolean AND queries. For Boolean NOT and OR queries, a comparable search time between GBI and Inverted Index is observed (see sections 4.3.1.2, 4.3.2.2 and 4.3.3.2). Thus, if a set of documents whose contents completely

overlap is indexed, then GBI shows superior performance in the case of Boolean AND queries and comparable performance for Boolean NOT and OR queries with respect to Inverted Index.

- For neighbour of the keyword queries, the search time for GBI depends on how many keywords indexed are lexicographically before and after the search keyword. If more keywords are alphabetically after the search keyword, then finding its neighbour will have a constant search time. Even if there is a large number of keywords occurring before the search keyword, the search time is significantly lower than Inverted Index as GBI uses hash table lookups. On the other hand, with Inverted Index, the search time depends on how early the search keyword and the neighbouring keywords occur together in the documents indexed (see Section 4.4).

- For exclusive keyword queries, the search operation shows a constant time in GBI with an overall improvement of 99.9% in comparison to Inverted Index (see Section 4.5.1). Thus, it is faster to access the exclusive contents of a keyword using a single hash table lookup in GBI.

- For Boolean AND queries and a given set of system and workload parameters, GBI shows a performance improvement of 50%, 99%, and 70% over Elasticsearch for random, *Partial*, and a special case of *Full* relationship workloads respectively (see Section 4.7.3.1).

- For Boolean NOT queries and a given set of system and workload parameters, GBI shows a performance improvement of 68%, 85%, and 88% over Elastic-

search for random, *Partial*, and a special case of *Full* relationship workloads respectively (see Section 4.7.3.2).

- For Boolean OR queries and a given set of system and workload parameters, GBI shows a performance improvement of 13%, 41%, and 59% over Elasticsearch for random, *Partial*, and a special case of *Full* relationship workloads respectively (see Section 4.7.3.3).

- Indexing latency of Inverted Index is on average 4.2 and 5.2 times lower than Pruned GBI and GBI respectively (see Section 4.6). Also, memory consumption by Inverted Index is 2.5 and 3.5 times lower than Pruned GBI and GBI respectively. Establishing links between different keywords extracted from a document incurred an overhead in indexing latency in GBI and Pruned GBI.

5.3 Limitations and Recommendations

The proposed solution was primarily designed with Boolean queries in mind and further changes would be required to optimize GBI for different types of queries. The proposed GBI structure provides an arena for these future improvements.

The main limitations of GBI are its indexing time and memory consumption. Based on the indexing performance analysis described in Section 4.6, GBI can be used for applications that involve indexing once and searching multiple times throughout document lifetime. Examples include indexing large scale historical archival data

and indexing data for web search engines in Google and Twitter for example. Also, when more keywords are extracted from a document for indexing, more time and more memory are consumed by GBI for creating a relationship between the extracted keywords during the indexing process. Thus, GBI is most suitable for indexing documents with short textual content because they usually result in a smaller number of keywords when the documents are subjected to the keyword extraction process. As a result, this thesis also recommends that GBI will be most suitable for indexing a large volume of short textual documents. One example of documents with the short textual content is tweets (micro-blogs) generated on the social networking website Twitter.

5.4 Future Work

Directions for further research include the following:

- The performance analysis in this thesis is based on synthetic workloads. Evaluating the performance of GBI with workloads of real systems can form an important direction for future research.
- Using various concepts of graph theory, analyzing the structure of GBI in terms of node degree (in/out) could provide important information about the set of documents indexed such as the keywords that occur the most, and the keyword that has maximum connectivity with other keywords. This information could

further be used to improve the search performance and observe the sentiment of the context found in the indexed documents. Further research is required.

- One research direction is to make the indexing operation in GBI consume less time and memory so that it can be adopted to a wider variety of scenarios. Since GBI also indexes all the keywords extracted from the document in pairs, the same document id will be stored multiple times and thus consuming more time and memory. A possible solution is to optimize the storage of the postings list by storing document ids once and devising a way to refer to it multiple times wherever needed. Further investigation is warranted.

- Another research direction is to extend GBI support in optimizing the execution of other types of queries (such as prefix, wildcard, and phrasal) that will allow GBI to offer a wide range of searching capabilities.

- Comparing the performance of GBI with other textual indexing techniques (such as Suffix and Radix tree) could provide a more complete evaluation and more insights.

References

[1] D. Sayce, "The Number of tweets per day in 2020," [Available online at]: https://www.dsayce.com/social-media/tweets-day/, last accessed on Apr 23, 2020.

[2] T. Shay, "Top 5 Ways To Improve Your Database Performance," [Available online at]: https://www.eversql.com/5-easy-ways-to-improve-your-database-performance, last accessed on Apr 23, 2020.

[3] M. Drake, "Understanding Database Sharding," [Available online at]: https://www.digitalocean.com/community/tutorials/understanding-database-sharding, last accessed on Jul 2, 2020.

[4] A. K. Mahapatra and S. Biswa, "Inverted indexes: Types and techniques," *International Journal of Computer Science Issues (IJCSI)*, vol. 8, no. 4, pp. 384–392, 2011.

[5] University of Alaska, Fairbanks, "Boolean Searching," [Available online at]: https://library.uaf.edu/ls101-boolean, last accessed on Apr 23, 2020.

[6] A. Barysevich, "How to Use Boolean Search for Social Media Monitoring (and Why You Want to)," [Available online at]: https://

www.socialmediatoday.com/news/how-to-use-boolean-search-for-social-media-monitoring-and-why-you-want-to/548054/, last accessed on Apr 23, 2020.

[7] K. Wenzel, "Query Results using Boolean Logic," [Available online at]: https://www.essentialsql.com/get-ready-to-learn-sql-4-query-results-using-boolean-logic/, last accessed on Apr 23, 2020.

[8] C. D. Manning, H. Schütze, and P. Raghavan, *Introduction to Information Retrieval*. Cambridge university press, 2008.

[9] B. Slawski, "Caffeine: Google's Indexer," [Available online at]: https://gofishdigital.com/googles-indexer-caffeine/, last accessed on Apr 23, 2020.

[10] A. Brasetvik, "Elasticsearch from the Bottom Up, Part 1," [Available online at]: https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up, last accessed on May 6, 2020.

[11] R. Sharma, A. Kandpal, P. Bhakuni, R. Chauhan, R. H. Goudar, and A. Tyagi, "Web Page Indexing through Page Ranking for Effective Semantic Search," in *7th International Conference on Intelligent Systems and Control (ISCO)*, 2013, pp. 389–392.

[12] R. Nagmoti, A. Teredesai, and M. De Cock, "Ranking Approaches for Microblog Search," in *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2010, pp. 153–157.

[13] A. K. Mohideen, S. Majumdar, M. St-Hilaire, and A. El-Haraki, "A Graph-Based Indexing Technique to Enhance the Performance of Boolean AND

Queries in Big Data Systems," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 677–680.

[14] A. Kumar, S. Johari, and S. Saha, "Reduction in Searching Time of Inverted Index Using Bloom Filter," in *International Conference on Emerging Research in Computing, Information,Communication and Applications (ERCICA-14)*, 2014, pp. 239–245.

[15] H. Wu, G. Li, and L. Zhou, "Ginix: Generalized Inverted Index for Keyword Search," *Tsinghua Science and Technology*, vol. 18, no. 1, pp. 77–87, 2013.

[16] A. Moffat and J. Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval," *ACM Transactions on Information Systems (TOIS)*, vol. 14, no. 4, pp. 349–379, 1996.

[17] S. Pohl, A. Moffat, and J. Zobel, "Efficient Extended Boolean Retrieval," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 6, pp. 1014–1024, 2012.

[18] V. N. Anh and A. Moffat, "Pruned Query Evaluation Using Pre-Computed Impacts," in *29th annual international ACM SIGIR conference on Research and development in information retrieval*, 2006, pp. 372–379.

[19] V. N. Anh, O. de Kretser, and A. Moffat, "Vector-Space Ranking with Effective Early Termination," in *24th annual international ACM SIGIR conference on Research and development in information retrieval*, 2001, pp. 35–42.

[20] M. Fontoura, M. Gurevich, V. Josifovski, and S. Vassilvitskii, "Efficiently Encoding Term Co-occurrences in Inverted Indexes," in *20th ACM international conference on Information and knowledge management*, 2011, pp. 307–316.

[21] Apache Software Foundation, "Apache Lucene," [Available online at]: https://lucene.apache.org/, last accessed on Oct 25, 2019.

[22] Z. Zhaofeng, "Analysis of Lucene — basic concepts," [Available online at]: https://medium.com/@Alibaba_Cloud/analysis-of-lucene-basic-concepts-5ff5d8b90a53, last accessed on Aug 06, 2020.

[23] S. Lakhara and N. Mishra, "Desktop Full-Text Searching based on Lucene: a Review," in *IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, 2017, pp. 2434–2438.

[24] L. Qian and L. Wang, "An Evaluation of Lucene for Keywords Search in Large-scale Short Text Storage," in *International Conference On Computer Design and Applications*, 2010, pp. V2–206–V2–209.

[25] H. Cai, Z. Huang, D. Srivastava, and Q. Zhang, "Indexing Evolving Events from Tweet Streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 11, pp. 3001–3015, 2015.

[26] C. Chen, F. Li, B. C. Ooi, and S. Wu, "TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets," in *ACM SIGMOD International Conference on Management of data*, 2011, pp. 649–660.

[27] L. Wu, W. Lin, X. Xiao, and Y. Xu, "LSII: An Indexing Structure for Exact Real-Time Search on Microblogs," in *29th IEEE International Conference on Data Engineering (ICDE)*, 2013, pp. 482–493.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[29] F. Zhao, J. Liu, J. Zhou, H. Jin, and L. T. Yang, "LS-AMS: An Adaptive Indexing Structure for Realtime Search on Microblogs," *IEEE Transactions on Big Data*, vol. 1, no. 4, pp. 125–137, 2015.

[30] L. Jiang, S.-I. Yu, D. Meng, Y. Yang, T. Mitamura, and A. G. Hauptmann, "Fast and Accurate Content-based Semantic Search in 100M Internet Videos," in *23rd ACM international conference on Multimedia*, 2015, pp. 49–58.

[31] Z. Wen, X. Liu, H. Cao, and B. He, "RTSI: An Index Structure for Multi-Modal Real-Time Search on Live Audio Streaming Services," in *34th IEEE International Conference on Data Engineering (ICDE)*, 2018, pp. 1495–1506.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[33] N. Popov, "Understanding PHP's internal array implementation," [Available online at]: https://nikic.github.io/2012/03/28/Understanding-PHPs-internal-array-implementation.html, last accessed on Apr 20, 2020.

[34] G. L. McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*. Palo Alto, CA : CareerCup, LLC, 2015.

[35] N. Ram, R. Ranjan, S. Chakrabarti, and D. Samanta, "Application of Data Structure in the field of Cryptography," in *International Conference on Computational Systems for Health & Sustainability, RV College of Engineering*, 2015, pp. 65–68.

[36] A. Gutmans and Z. Suraski, "Zend hash," [Available online at]: https://github.com/php/php-src/blob/PHP-5.6.10/Zend/zend_hash.h#L237, last accessed on Apr 20, 2020.

[37] N. Popov, "PHP's new hashtable implementation," [Available online at]: https://nikic.github.io/2014/12/22/PHPs-new-hashtable-implementation.html, last accessed on Apr 20, 2020.

[38] Wikipedia, "Hash table," [Available online at]: https://en.wikipedia.org/wiki/Hash_table#Key_statistics, last accessed on Sep 12, 2020.

[39] Yourbasic, "Hash tables explained," [Available online at]: https://yourbasic.org/algorithms/hash-tables-explained/#resizing-in-constant-amortized-time, last accessed on Apr 20, 2020.

[40] S. Shah and A. Shaikh, "Hash Based Optimization for Faster Access to Inverted Index," in *International Conference on Inventive Computation Technologies (ICICT)*, 2016, pp. 1–5.

[41] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-Time Search at Twitter," in *28th IEEE International Conference on Data Engineering*, 2012, pp. 1360–1369.

[42] H. Huang, J. Li, R. Zhang, W. Yu, and W. Ju, "LiveIndex: A distributed online index system for temporal microblog data," in *17th IEEE International Conference on High Performance Computing and Communications, 7th IEEE International Symposium on Cyberspace Safety and Security, and 12th IEEE International Conference on Embedded Software and Systems*, 2015, pp. 884–887.

[43] S. I. Mohammed, F. A. Omara, and H. M. Sharaf, "Information retrieval using dynamic indexing," in *9th International Conference on Informatics and Systems*, 2014, pp. PDC–93–PDC–101.

[44] D. G. Lee, Y. J. Jung, Y. W. Lee, and K. H. Ryu, "Hashed Multiple Lists: A Stream Filter for Processing Continuous Query with Multiple Attributes in Geosensor Networks," in *8th IEEE International Conference on Computer and Information Technology Workshops*, 2008, pp. 104–109.

[45] S. Siddiqi and A. Sharan, "Keyword and Keyphrase Extraction Techniques: A Literature Review," *International Journal of Computer Applications*, vol. 109, no. 2, pp. 18–23, 2015.

[46] S. Beliga, "Keyword extraction: a review of methods and approaches," *University of Rijeka, Department of Informatics, Rijeka*, pp. 1–9, 2014.

[47] P. D. Tumey, "Learning to extract keyphrases from text," *NRC Technical Report ERB-l 057. National Research Council, Canada*, pp. 1–43, 1999.

[48] F. Eibe, G. W. Paynter, I. H. Witten, C. Gutwin, and C. G. Nevill-Manning, "Domain-Specific Keyphrase Extraction," in *16th International Joint Conference on Artificial Intelligence*, 1999, pp. 668–673.

[49] M. Krapivin, A. Autayeu, M. Marchese, E. Blanzieri, and N. Segata, "Keyphrases Extraction from Scientific Documents: Improving Machine Learning Approaches with Natural Language Processing," in *International Conference on Asian Digital Libraries*, 2010, pp. 102–111.

[50] C. Zhang, "Automatic Keyword Extraction from Documents Using Conditional Random Fields," *Journal of Computational Information Systems*, vol. 4, no. 3, pp. 1169–1180, 2008.

[51] Y. HaCohen-Kerner, "Automatic Extraction of Keywords from Abstracts," in *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, 2003, pp. 843–849.

[52] C. Pasquier, "Single Document Keyphrase Extraction Using Sentence Clustering and Latent Dirichlet Allocation," in *5th international workshop on semantic evaluation*, 2010, pp. 154–157.

[53] N. Pudota, A. Dattolo, A. Baruzzo, and C. Tasso, "A New Domain Independent Keyphrase Extraction System," in *Italian Research Conference on Digital Libraries*, 2010, pp. 67–78.

[54] Z. Yang, J. Lei, K. Fan, and Y. Lai, "Keyword extraction by entropy difference between the intrinsic and extrinsic mode," *Physica A: Statistical Mechanics and its Applications*, vol. 392, no. 19, pp. 4523–4531, 2013.

[55] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," in *31st International Conference on Machine Learning (ICML)*, 2014, pp. 1188–1196.

[56] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv preprint arXiv:1301.3781*, 2013.

[57] J. H. Lau and T. Baldwin, "An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation," in *1st Workshop on Representation Learning for NLP*, 2016, pp. 78–86.

[58] J. Li, G. Huang, C. Fan, Z. Sun, and H. Zhu, "Key word extraction for short text via word2vec, doc2vec, and textrank," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 27, no. 3, pp. 1794–1805, 2019.

[59] A. Chakraborty, T. Dutta, S. Mondal, and A. Nath, "Application of Graph Theory in Social Media," *International Journal of Computer Sciences and Engineering*, vol. 6, no. 10, pp. 722–729, 2018.

[60] M. S. El Bazzi, D. Mammass, T. Zaki, and A. Ennaji, "A graph based method for Arabic document indexing," in *7th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)*, 2016, pp. 308–312.

[61] V. T. Kesavan and B. S. Kumar, "Graph Based Indexing Techniques for Big Data Analytics: a Systematic Survey," *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 7, no. 6S5, pp. 641–647, 2019.

[62] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao *et al.*, "A1: A Distributed In-Memory Graph Database," in *2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 329–344.

[63] K. Aoyama, A. Ogawa, T. Hattori, T. Hori, and A. Nakamura, "Graph Index Based Query-by-Example Search on a Large Speech Data Set," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8520–8524.

[64] K. M. Kyu and A. N. Oo, "Graph-based Indexing Method for Searching in RDF Data," in *International Conference on Advanced Information Technologies (ICAIT)*, 2019, pp. 96–101.

[65] K. M. Hammouda and M. S. Kamel, "Phrase-based Document Similarity Based on an Index Graph Model," in *IEEE International Conference on Data Mining*, 2002, pp. 203–210.

[66] A. Schenker, M. Last, H. Bunke, and A. Kandel, "Classification of Web Documents Using a Graph Model," in *7th International Conference on Document Analysis and Recognition*, 2003, pp. 240–244.

[67] P. Das, A. K. Das, J. Nayak, D. Pelusi, and W. Ding, "A Graph Based Clustering Approach for Relation Extraction From Crime Data," *IEEE Access*, vol. 7, pp. 101 269–101 282, 2019.

[68] S. Yata, K. Morita, M. Fuketa, and J. Aoe, "Fast String Matching with Space-efficient Word Graphs," in *International Conference on Innovations in Information Technology*, 2008, pp. 79–83.

[69] Y. Ohsawa, N. E. Benson, and M. Yachida, "KeyGraph: Automatic Indexing by Co-occurrence Graph based on Building Construction Metaphor," in *IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL-98)*, 1998, pp. 12–18.

[70] R. Mihalcea and P. Tarau, "TextRank: Bringing Order into Texts," in *conference on empirical methods in natural language processing*, 2004, pp. 404–411.

[71] W. D. Abilhoa and L. N. De Castro, "A keyword extraction method from twitter messages represented as graphs," *Applied Mathematics and Computation*, vol. 240, pp. 308–325, 2014.

[72] M. Kumar, P. Aggarwal *et al.*, "A Graph based Keyword Extraction from Twitter using Node and Edge Weight," in *International Conference on Data Science and Engineering (ICDSE)*, 2019, pp. 35–39.

[73] S. Anjali, M. Nair, and M. Thushara, "A Graph based Approach for Keyword Extraction from Documents," in *2nd International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, 2019, pp. 1–4.

[74] J. Barbay, A. López-Ortiz, and T. Lu, "Faster Adaptive Set Intersections for Text Searching," in *International Workshop on Experimental and Efficient Algorithms*, 2006, pp. 146–157.

[75] J. S. Culpepper and A. Moffat, "Efficient Set Intersection for Inverted Indexing," *ACM Transactions on Information Systems*, vol. 29, no. 1, pp. 1:1–1:25, 2010.

[76] Wikipedia, "JSON," [Available online at]: https://en.wikipedia.org/wiki/JSON, last accessed on Jul 09, 2020.

[77] Carleton University, "Research Computing and Development Cloud (RCDC)," [Available online at]: https://carleton.ca/rcs/research-computing-and-development-cloud-rcdc/, last accessed on Jun 16, 2020.

[78] C. Bond, "27 Google Search Statistics You Should Know in 2019 (+ Insights!)," [Available online at]: https://www.wordstream.com/blog/ws/2019/02/07/google-search-statistics, last accessed on Oct 25, 2019.

[79] G. P. Dritto, "An Overview on Elasticsearch and its usage," [Available online at]: https://towardsdatascience.com/an-overview-on-elasticsearch-and-its-usage-e26df1d1d24a, last accessed on Jul 09, 2020.

[80] Oracle, "Heap Tuning Parameters," [Available online at]: https://docs.oracle.com/cd/E19900-01/819-4742/abeik/index.html, last accessed on Jul 09, 2020.

[81] Opster, "Elasticsearch Shards and Replicas getting started guide," [Available online at]: https://opster.com/blogs/elasticsearch-shards-and-replicas-getting-started-guide/, last accessed on Jul 21, 2020.