STATIC AND ADAPTIVE INDEXING FRAMEWORK FOR BIG DATA USING PREDICTOR LOGIC

AISHA SIDDIQA

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

STATIC AND ADAPTIVE INDEXING FRAMEWORK FOR BIG DATA USING PREDICTOR LOGIC

AISHA SIDDIQA

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

I would like to dedicate my work to my soulmate;

behind my success there is he…

UNIVERSITY OF MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: AISHA SIDDIQA

Registration/Matric No: WHA130025

Name of Degree: DOCTOR OF PHILOSOPHY:

Title of Thesis: Static and Adaptive Indexing Framework for Big Data Using Predictor Logic

Field of Study: BIG DATA INDEXING (COMPUTER SCIENCE)

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date:

Subscribed and solemnly declared before,

Witness's Signature Date:

Name:

Designation:

ABSTRACT

Big data with exponential growth come in various forms and require efficient data processing systems for fast retrieval. The disrupted features that are associated with big data have elicited attention from research and industry; the research efforts aim to explore viable solutions that can improve data retrieval performance for better insight. Indexing has undoubtedly contributed to increased search performance for big data sets; for big data indexing, researchers have used many indexing structures such as clustered and non-clustered. However, because of the continuous increase in data size, contemporary big data indexing mechanisms are inadequate to achieve efficiency in query responses. Clustered indexing approaches are constrained to number of replicas to offer indexing on a sufficient number of attributes, whereas non-clustered indexing implementation incurs high indexing overhead. Therefore, existing big data indexing structures are unable to achieve the maximum index hit ratio. The aim of this study is to expedite the data retrieval process with minimum indexing overhead and maximum index hit ratio against search queries for big data by using non-clustered indexing approach. Static indexes are created based on a user-provided list of index attributes before starting query execution, which are updated adaptively based on changing query workload to obtain an increased index hit ratio. We investigate contemporary big data indexing implementation and analyze its inefficiency in index creation time and index size. Furthermore, we observe that because of the limited number of indexes available with clustered indexing approaches, most queries are executed without using indexes. Thus, we propose a novel indexing framework for big data, named SmallClient, with minimized indexing overhead, improved search performance, and improved index hit ratio. SmallClient leverages B-Tree indexing structure and uses novel predictor logic for indexing. We collected data for indexing overhead (both in terms of indexing time and index size) as well as search performance and index hit ratio for static and adaptive indexing, respectively, to validate the performance of the framework. We use benchmarking and mathematical modeling for verification of SmallClient results. The results of indexing time prove that SmallClient has decreased indexing time overhead by up to 32% from 47%, taken by the Lucene indexing library. Similarly, index size overhead is 41% for large data sets where Lucene fails to create indexes. The results also prove that the search performance of SmallClient is more than 92% without intervening data uploading cost and that this framework achieves improved index hit ratio by adaptively updating indexes.

ABSTRAK

Big data dengan pertumbuhan eksponen datang dalam pelbagai bentuk dan memerlukan sistem cekap memproses data untuk capaian cepat. Ciri-ciri disrupted yang dikaitkan dengan big data telah elicited perhatian daripada penyelidikan dan industri; usaha-usaha penyelidikan ini bertujuan untuk meneroka penyelesaian yang berdaya maju yang boleh meningkatkan prestasi pencarian data wawasan yang lebih baik. Indeks tidak syak lagi telah menyumbang kepada prestasi meningkat Cari set big data; untuk big data Indeks, penyelidik telah menggunakan banyak struktur Indeks seperti berkelompok dan bebas berkelompok. Walau bagaimanapun, disebabkan oleh peningkatan berterusan dalam saiz data, mekanisme Indeks kontemporari big data adalah tidak mencukupi untuk mencapai kecekapan dalam jawapan pertanyaan. Pendekatan pengindeksan Berkelompok akan dikekang bilangan replika agar Indeks bilangan mencukupi sifat-sifat, manakala pelaksanaan Indeks-Berkelompok akan tinggi Indeks overhed. Oleh yang demikian, struktur Indeks big data yang sedia ada tidak dapat mencapai Indeks maksimum mencecah nisbah. Kajian ini bertujuan untuk mempercepatkan proses pencarian data dengan minimum pengindeksan overhed dan Indeks maksimum mencecah nisbah terhadap pertanyaan carian untuk big data dengan menggunakan pendekatan Indeks non-clustered. Statik Indeks dicipta berdasarkan pengguna-menyediakan senarai atribut Indeks sebelum memulakan pelaksanaan pertanyaan, yang akan dikemaskini Kurangkan berdasarkan perubahan beban kerja pertanyaan untuk mendapatkan Indeks meningkat mencecah nisbah. Kami menyiasat kontemporari big data Indeks pelaksanaan dan menganalisis dengan ketidakcekapan dalam Indeks penciptaan masa dan Indeks saiz. Tambahan pula, kita Perhatikan bahawa kerana bilangan terhad disediakan Indeks dengan pendekatan pengindeksan Berkelompok, kebanyakan pertanyaan dijalankan tanpa menggunakan Indeks. Oleh itu, kami mencadangkan rangka pengindeksan novel untuk big data, bernama SmallClient, dengan diminimumkan pengindeksan overhead, Cari peningkatan prestasi, dan peningkatan indeks hit nisbah. SmallClient memanfaatkan B-Tree pengindeksan struktur dan menggunakan novel faktor peramal logik untuk mengindeks. Kami mengumpul data untuk Indeks overhead (baik dari segi indeks masa dan Indeks saiz) dan Cari prestasi dan Indeks hit nisbah bagi statik dan mudah suai Indeks, masing-masing untuk mengesahkan prestasi rangka kerja. Kami menggunakan tanda aras dan model matematik untuk pengesahan keputusan SmallClient. Keputusan indeks masa membuktikan bahawa SmallClient telah menurun indeks masa overhed sehingga 32% daripada 47%, diambil oleh Lucene pengindeksan Perpustakaan. Begitu juga, Indeks saiz overhed adalah 41% bagi set data yang besar di mana Lucene gagal untuk mencipta Indeks. Keputusan juga membuktikan bahawa prestasi Cari SmallClient adalah lebih daripada 92% tanpa kematangan data naik kos dan bahawa rangka kerja ini mencapai peningkatan indeks hit nisbah dengan Kurangkan mengemas kini indeks adaptif.

ACKNOWLEDGEMENTS

I would like to acknowledge my supervisor Professor Dr. Abdullah Gani for his continual support and guidance throughout this research. His encouragement has made me produce such valuable piece of contribution in the form of this thesis. Besides that, working under his supervision was also a memorable experience of my life.

I also acknowledge Ministry of Higher Education, Malaysia for the financial support for this duration of study.

TABLE OF CONTENTS

ABSTRACT III

ABSTRAK V

ACKNOWLEDGEMENTS VII

TABLE OF CONTENTS VIII

LIST OF FIGURES XII

LIST OF TABLES XV

LIST OF SYMBOLS AND ABBREVIATIONS XVI

CHAPTER 1: INTRODUCTION 1

1.1 Research Motivation 1

1.2 Statement of Problem 3

1.3 Statement of Objectives 5

1.4 Proposed Methodology 6

1.5 Thesis Organization 8

CHAPTER 2: BIG DATA INDEXING TECHNIQUES: THE STATE-OF-THE-ART 11

2.1 Big Data Indexing Requirements 11

2.2 Indexing for Traditional Data 15

2.2.1 Classification of Indexing Techniques 16

2.2.2 Analysis of Indexing Techniques for Big Data Indexing Requirements 33

2.3 Indexing Implementation on Big Data 37

2.3.1 Clustered Indexing on Big Data 38

2.3.2 Non-clustered Indexing on Big Data 40

2.3.3 Analysis of Indexing Techniques Implemented on Big Data 41

2.4 Conclusion 45

CHAPTER 3: PERFORMANCE ANALYSIS OF INDEXING TECHNIQUES FOR BIG DATA 47

3.1 Experimental Setup and Data Collection 47

3.1.1 The Model 48

3.1.2 Performance Measures 50

3.1.3 Data Set Used 51

3.1.4 Data Collection Tools 52

3.2 Results and Discussion 53

3.2.1 Index Size Results 53

3.2.2 Indexing Time Results 58

3.2.3 Search Time Results 62

3.2.4 Index Hit ratio 66

3.3 Conclusion 67

CHAPTER 4: SMALLCLIENT FOR BIG DATA: PROPOSED INDEXING FRAMEWORK 69

4.1 SmallClient Indexing Framework 69

4.2 The Architecture 71

4.3 Framework Modules 73

4.3.1 Block Creation 74

4.3.2 Index Creation 76

4.3.3 Query Execution 81

4.4 Mathematical Model for SmallClient 84

4.5 Features of SmallClient 88

4.6 Conclusion 90

CHAPTER 5: EVALUATION 92

5.1 Test bed 92

5.2 Evaluation Measures 93

5.2.1 Data Upload Overhead 93

5.2.2 Indexing Overhead 95

5.2.3 Search Performance 96

5.2.4 Index Hit Ratio 98

5.3 Framework Design 99

5.4 Data Collection Tools 103

5.4.1 Data Collection for Experiment 103

5.4.2 Data Collection for Benchmarking 104

5.4.3 Data Collection for Mathematical Model 106

5.5 Conclusion 108

CHAPTER 6: RESULTS AND DISCUSSION 109

6.1 Validation 109

6.1.1 Experimental Results 110

6.1.2 Mathematical Modeling Results 120

6.2 Verification 123

6.2.1 Benchmarking 123

6.2.2 Mathematical Modeling 134

6.3 Conclusion 136

CHAPTER 7: CONCLUSION AND FUTURE WORK 138

7.1 Fulfillment of Aim and Objectives 138

7.1.1 Investigating the capability of existing indexing techniques to address the challenges of big data to establish potential research problem 138

7.1.2 Designing and implementing an indexing framework using non-clustered indexing structure incorporated with predictor function for adaptive index updating, which ensures the following: 139

7.1.3 Evaluating the effectiveness of proposed indexing framework with respect to overhead resulted by static and adaptive indexing, query execution and data retrieval time, and index hit ratio 140

7.1.4 Verifying the results of proposed indexing framework using benchmarking and mathematical modeling 140

7.2 Research Contributions 141

7.2.1 Taxonomy of State-of-the-Art Indexing Techniques 141

7.2.2 SmallClient: a novel indexing framework for big data 141

7.2.3 Algorithmic Design for SmallClient 142

7.2.4 Java Class Library for SmallClient 142

7.2.5 Mathematical Model of SmallClient 143

7.3 Limitations and Future Work 143

REFERENCES 145

APPENDIX A Data Uploading 152

APPENDIX B Indexing 155

APPENDIX C Query Execution 172

APPENDIX D Predictor 184

LIST OF PUBLICATIONS AND PAPERS PRESENTED 186

LIST OF FIGURES

Figure 1-1: Trends of Big Data and Indexing 2

Figure 1-2: Research Milestones and Methodology 7

Figure 1-3: Thesis Organization..... 9

Figure 2-1: Taxonomy of Indexing Techniques (Gani, Siddiqa, Shamshirband, & Hanum, 2015) 17

Figure 3-1: Experimental Model 48

Figure 3-2: Indexed Search Query Execution 49

Figure 3-3: Index Size comparison with Data Set size and Number of Index Attributes 56

Figure 3-4: Index Size Overhead for varying number of Index Attributes 57

Figure 3-5: Indexing Time comparison with Data Upload Time and Number of Index Attributes 60

Figure 3-6: Indexing Time Overhead for varying number of Index Attributes 61

Figure 3-7: Search Time Comparison between Full Scan and Indexed Search 64

Figure 3-8: Improved Search Performance with Index Search 64

Figure 3-9: Index Hit ratio Comparison with varying No. of Index Attributes 67

Figure 4-1: Proposed indexing framework, SmallClient 71

Figure 4-2: The Architecture for SmallClient 72

Figure 4-3: Block Creation Process 75

Figure 4-4: Index Creation Process 76

Figure 4-5: Sequence Diagram of Index Creation 78

Figure 4-6: Predictor Function 80

Figure 4-7: Sequence Diagram of Query Execution 83

Figure 4-8: Mathematical Model for SmallClient 85

Figure 6-1: Data Upload Time Experiment Results..111

Figure 6-2: Data Size Results from Experiments..112

Figure 6-3: Indexing Time Results from Experiments for up to five indexes 115

Figure 6-4: Index Size Resuts for five indexes 117

Figure 6-5: Query Execution Time Results from Experiments 118

Figure 6-6: Index Hit Ratio Results for Static and Adaptive Indexes 120

Figure 6-7: Data Upload Time Resuts using Mathematical Model 121

Figure 6-8: Indexing Time Resuts using Mathematical Model 122

Figure 6-9: Query Execution Time Resuts using Mathematical Model 122

Figure 6-10: Benchmarking on Data Uploading Time 124

Figure 6-11: Data Upload Time Overhead...125

Figure 6-12: Benchmarking on Data Size..126

Figure 6-13: Data Upload Size Overhead 126

Figure 6-14: Indexing Time Comparison for up to five indexes when created using Lucene vs SmallClient 128

Figure 6-15: Indexing Time Results using Lucene, SmallCLient and I-SmallClient...129

Figure 6-16: Indexing Time Overhead Results...130

Figure 6-17: Index Size Results using Lucene and SmallCLient 131

Figure 6-18: Index Size Overhead Results...132

Figure 6-19: Query Execution Time Comparison between full scan, Lucene and SmallClient...133

Figure 6-20: Search Performance results using SmallClient and Lucene....................133

Figure 6-21: Data Upload Time Mathematical Verification.......................................135

Figure 6-22: Indexing Time Mathematical Verification...136

Figure 6-23:s Query Execution Time Mathematical Verification 136

LIST OF TABLES

Table 2-1: Indexing Techniques for Traditional Data (Gani et al., 2015) 22

Table 2-2: Analysis of Indexing Techniques for Big Data Indexing Requirements (Gani et al., 2015).. 34

Table 2-3: Comparison of Clustered and Non-Clustered Indexing 38

Table 2-4: Analysis of Clustered Indexing Approaches for Big Data 42

Table 3-1: Data Sets 52

Table 3-2: Index Size Results 55

Table 3-3: Indexing Time Results... 59

Table 3-4: Search Time Results 62

Table 3-5: Index Hit Ratio Results 66

Table 4-1: Description of Places in SmallClient Mathematical Model 87

Table 4-2: Description of transitions in SmallClient Mathematical Model.................... 87

Table 6-1: Data Upload Time Results... 110

Table 6-2: Data Upload Size Results 111

Table 6-3: Indexing Time Results for up to five Indexes 114

Table 6-4: Index Size Results for up to five Indexes 116

Table 6-5: Query Execution Time and Search Performance Results............................ 118

Table 6-6: Query Log observations and Index Hit Ratio Results before and after Predictor Logic 119

LIST OF SYMBOLS AND ABBREVIATIONS

AI	:	Artificial Intelligent
CAI	:	Collaborative Artificial Intelligent
CKRR	:	Collaborative Knowledge Representation and Reasoning
CML	:	Collaborative Machine Learning
CPN	:	Colored Petri Nets
CSV	:	Comma Separated Values
ER	:	Entity Relationship
HDFS	:	Hadoop Distributed File System
NAI	:	Non Artificial Intelligent

CHAPTER 1: INTRODUCTION

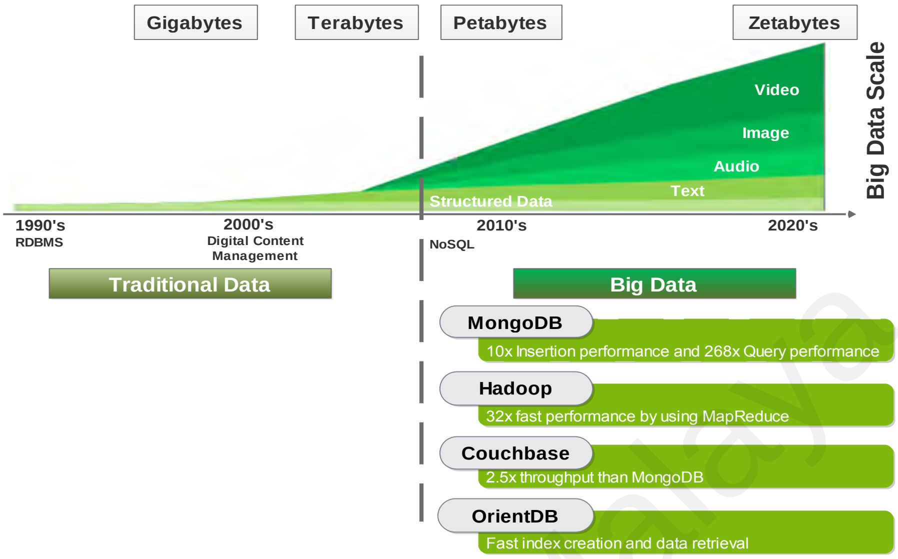
This chapter provides an overview of the current study. We explain the motivation for this thesis on big data indexing and present the research problem that we have addressed and investigated. Moreover, we present the aim and objectives of our research; the proposed methodology in undertaking the research process is also described in this chapter. In addition, we briefly outline the structure of this thesis in this chapter.

The rest of this chapter is organized as follows: Section 1.1 presents the background and motivation of our research. We present the statement of the problem in Section 1.2, followed by the research aim and objectives in Section 1.3. The proposed research methodology is illustrated in Section 1.4 and the thesis outline is presented in Section 1.5.

1.1 Research Motivation

Information technology has become a crucial part of today’s lifestyle. This role has resulted in the voluminous amount of structured and non-structured data, which is rapidly growing. This type of data is known as big data. Potential contributors of big data repositories are healthcare monitoring and diagnosing systems, social networks, e-science, and e-commerce. Big data management systems are also improving to cope with current challenges. However, big data is beyond the capabilities of these systems to fulfill its storage, processing, and visualization requirements. Therefore, there is still an urge for an efficient technology to handle big data.

Big data sets have crossed the boundaries of traditional data structures and comprise more than mere relational records such as text, audio, images, and videos in heterogeneous formats. Given the wider data type coverage and inclusion of every bit produced by information sensing systems, this pool of data is estimated to be doubled after every two years. The volume of the digital universe is further projected to grow by a factor of 300 and will be up to 40,000 exabytes from 2005 to 2020.



Gigabytes
Terabytes
Petabytes
Zetabytes
Big Data Scale
Video
Image
Audio
Text
Structured Data
1990's
RDBMS
2000's
Digital Content Management
2010's
NoSQL
2020's
Traditional Data
Big Data
MongoDB
10x Insertion performance and 268x Query performance
Hadoop
32x fast performance by using MapReduce
Couchbase
2.5x throughput than MongoDB
OrientDB
Fast index creation and data retrieval

Figure 1-1: Trends of Big Data and Indexing

Figure 1-1 depicts the shift of big data from traditional data in terms of volume, variety, and velocity. The figure chronologically presents the evolution of data management systems from Relational Database Management System (RDBMS) to recent NoSQL technologies. The figure also shows that during this evolution, a distinct change in volume and variety of data must be processed.

Figure 1-1 also presents the attained performance improvement in data retrieval process by current NoSQL technologies through indexing implementation. Although the available big data management tools include well-defined standards and procedures, they are not capable of handling the challenges of emerging big data. Therefore, the research on big data storage and analytics aims to escalate the adaptability of data management architectures and operational models for forthcoming big data needs.

The main concern of big data analytics is to perform efficient search and retrieval operations on big data to obtain insights towards generating value. Undoubtedly, immense indexing procedures are available for traditional data retrieval and search systems that demonstrate competitive outcomes. However, in the era of big data, terabyte to petabyte scale records deteriorate the performance of traditional indexing procedures. Moreover, the complexity and velocity associated with big data also hinder their performance. Longer execution times and additional storage requirements are the immediate consequences of implementing traditional indexing procedure on big data. This inadequacy provides a motivation to undertake the research on an acceptable data retrieval solution; it encourages designing an improved search mechanism.

The emergence of new storage technologies to confront the needs of big data also reveals the need for quick responses to data search queries. Therefore, efficiency in query execution and data retrieval is highly important for faster decision-making. For instance, in the field of body sensor networks, the increasing costs of healthcare and ageing of population are major subjects that have critical information-retrieval requirements. Thus, in distributed and replicated big data storage systems, we conduct this research because of the need for fast data processing and timely query responses.

1.2 Statement of Problem

Performing fast data search and retrieval operations over big text data for data analytics and visualization is a challenging task because millions of records are located in a distributed replicated environment. Prevailing big data management systems propose numerous indexing mechanisms with the evolution of big data. However, these systems do not show satisfactory performance for search queries because indexes are not well-designed. We explain the limitations of various indexing approaches for big data and present the statement of the problem.

Clustered indexing approaches with static and adaptive features are applicable for big data only with certain limitations. These approaches physically re-order data records based on an attribute to create a single index and require a distinct number of replicas of a data set to create more than one index. Therefore, an increase in the number of indexes requires an increase in the number of replicas. This limitation imposes storage constraints that might prevent big data sets from having as many replicas as the number of indexes required.

Complex index updating is another limitation of clustered indexing approaches. Replacing an available index with a new index requires re-ordering the data records of that replica. Adding new indexes requires creating new replicas for data sets with a new physical order of data records that are based on new index attributes. Furthermore, the selection of attributes for indexing is also challenging for clustered indexing approaches. Given that the number of indexes is limited to the number of replicas for a data set and index updating is a complex process, determining a useful list of attributes for indexing is critical.

Non-clustered indexing approaches for big data have their own inadequacies. Unlike clustered indexing approaches, non-clustered approaches do not restrict the number of indexes with a number of replicas for a data set. These approaches allow as many indexes to be created as required on only a single replica of a data set. However, non-clustered indexing approaches incur high indexing overhead (i.e., additional index creation time and storage space) and increase the delay to start the query execution process.

The selection of attributes that are to be utilized to create indexes is highly critical for a well-designed indexing structure. Creating indexes on all attributes that are provided as schema of a data set is impractical. The reason is that performing full text indexing on big data that comprises an extensive number of records results in high indexing upfront cost in terms of size and time. Thus, a longer delay occurs between data uploading and executing first query when the indexing mechanism is not suitable for such data and indexing is performed on all data attributes. Moreover, these indexes consume sufficiently large storage space.

Indexing for specific attributes is relatively preferable to minimize upfront cost of indexing. However, these statically created indexes do not fulfill requirements for queries that have selection predicates on different attributes. Consequently, these queries are executed using full scan, which is a remarkably time-consuming activity for big data.

The possibility also exists that indexing on a predetermined set of attributes may only be efficient for a specific span of time. Adding/deleting indexes that have the changing trend of data retrieval and having different workloads of queries is needed so that up-to-date indexes are available. This index updating can be invoked by users and/or by systems that automatically predict the changing query workload.

An indexing mechanism is needed that has wisely selected attributes to create indexes, thereby resulting in minimized indexing overhead for big data. In the meantime, by considering the changing query workload, predicting future query workload is needed to create or destroy indexes.

1.3 Statement of Objectives

This study aims to expedite the data search and data retrieval process from the pool of big data and provide up-to-date indexes through a proposed novel indexing framework that introduces both static and adaptive indexing to minimize indexing overhead, improve data search performance, and ultimately improve index hit ratio. The proposed solution is deployable on contemporary big data storage and processing systems such as Hadoop. The following are the objectives that are needed to attain the aim of our research:

a. Investigate the capability of existing indexing techniques towards the challenges of big data to establish the potential research problem. Big data indexing requirements are defined to analyze their fulfillment by existing indexing techniques. Review and investigation on existing indexing techniques based on defined big data indexing requirements leads to clarify the research gap to further address in this research.

b. Design and implement an indexing framework that uses the non-clustered indexing structure incorporated with predictor function to adaptively update for adaptive index updating, thereby ensuring the following:

    - minimized indexing overhead in terms of index creation/updating time and the space consumed by indexes (index size) for large volume data,
    - reduced data retrieval time with faster query execution, and
    - maximized index hit ratio by predicting the future workload of incoming search queries.

c. Validate the effectiveness of the proposed indexing framework with respect to indexing overhead, query execution and data retrieval time, as well as index hit ratio. Existing approaches are used as benchmark to ensure that proposed indexing framework has achieved minimized indexing time, reduced data retrieval time and maximized index hit ratio.

d. Verify the results of the proposed indexing framework by comparing experimental results with mathematical modeling results. The results obtained from experiments and mathematical model are compared to ensure that proposed framework demonstrates same performance in both environments.

1.4 Proposed Methodology

This section presents the research methodology of our thesis. We describe the milestones and steps undertaken to accomplish each milestone. Figure 1-2 explains the methodology of our research. We have the following four milestones: establishment of research problem, modeling the solution, evaluating the solution, and validating the results.

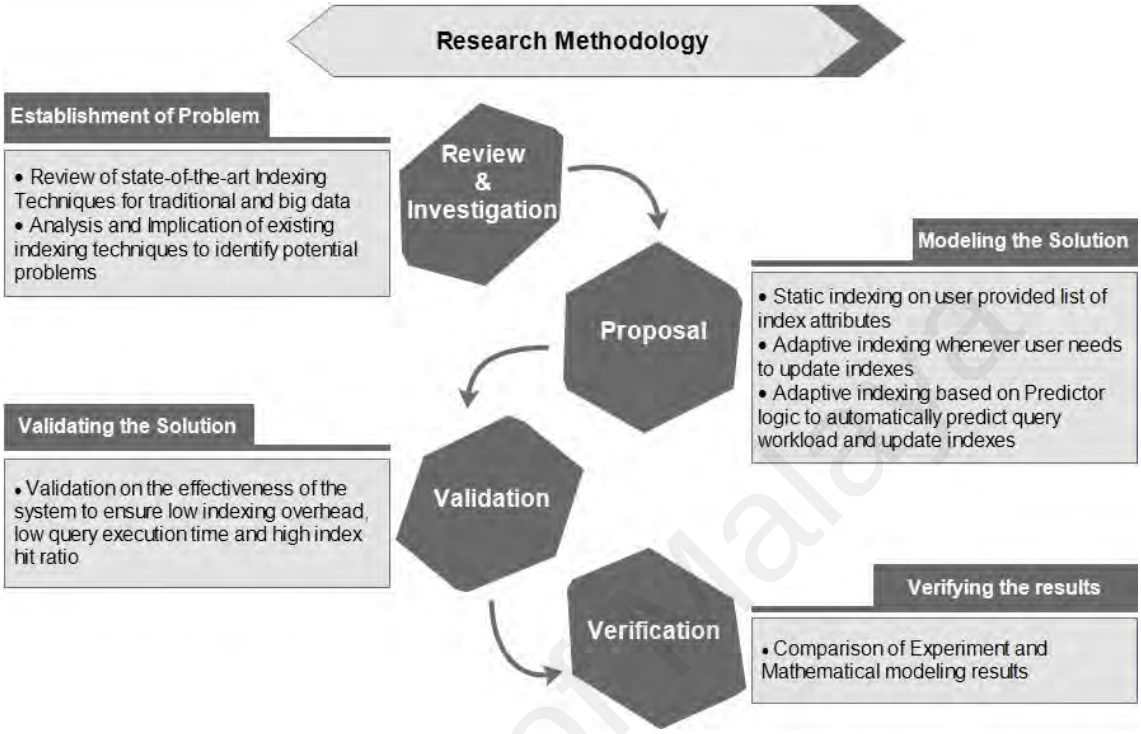

Research Methodology
Establishment of Problem
• Review of state-of-the-art Indexing Techniques for traditional and big data
• Analysis and Implication of existing indexing techniques to identify potential problems
Review & Investigation
Proposal
Modeling the Solution
• Static indexing on user provided list of index attributes
• Adaptive indexing whenever user needs to update indexes
• Adaptive indexing based on Predictor logic to automatically predict query workload and update indexes
Validating the Solution
• Validation on the effectiveness of the system to ensure low indexing overhead, low query execution time and high index hit ratio
Validation
Verification
Verifying the results
• Comparison of Experiment and Mathematical modeling results

Figure 1-2: Research Milestones and Methodology

We establish the research problem by reviewing and investigating the performance of existing indexing solutions for big data. We first identify big data indexing requirements and analyze recent indexing techniques for traditional data. We further performed review and investigation on indexing implementations for big data. Thereafter, we established the research problem by analyzing the performance of existing indexing implementations on big data.

We model the solution by proposing the indexing framework for big data named as SmallClient. SmallClient offers indexing procedure to create static and adaptive indexes. Users can provide a list of attributes for indexing either to be created during the data uploading process of SmallClient or at any random time whenever required. Furthermore, SmallClient introduces predictor logic to automatically predict future query workload based on which existing indexes are updated.

We validate the effectiveness of the solution by presenting the results for indexing overhead, query execution time, and index hit ratio. We show that SmallClient fulfills the objectives of this research and shows performance improvement in terms of minimized indexing overhead, minimized query execution cost, and increased index hit ratio.

We verify the results by using benchmarking and mathematical modeling. We collect data for benchmarking by using well-known big data search procedures such as Hadoop MapReduce and Hive for full scan and Apache Lucene indexing library for indexed search comparison. We use Petri nets to design a mathematical model for SmallClient and compare the experiment results with mathematical modeling results for verification.

1.5 Thesis Organization

The organization of the thesis is described in this section, and is presented in Figure 1-3.

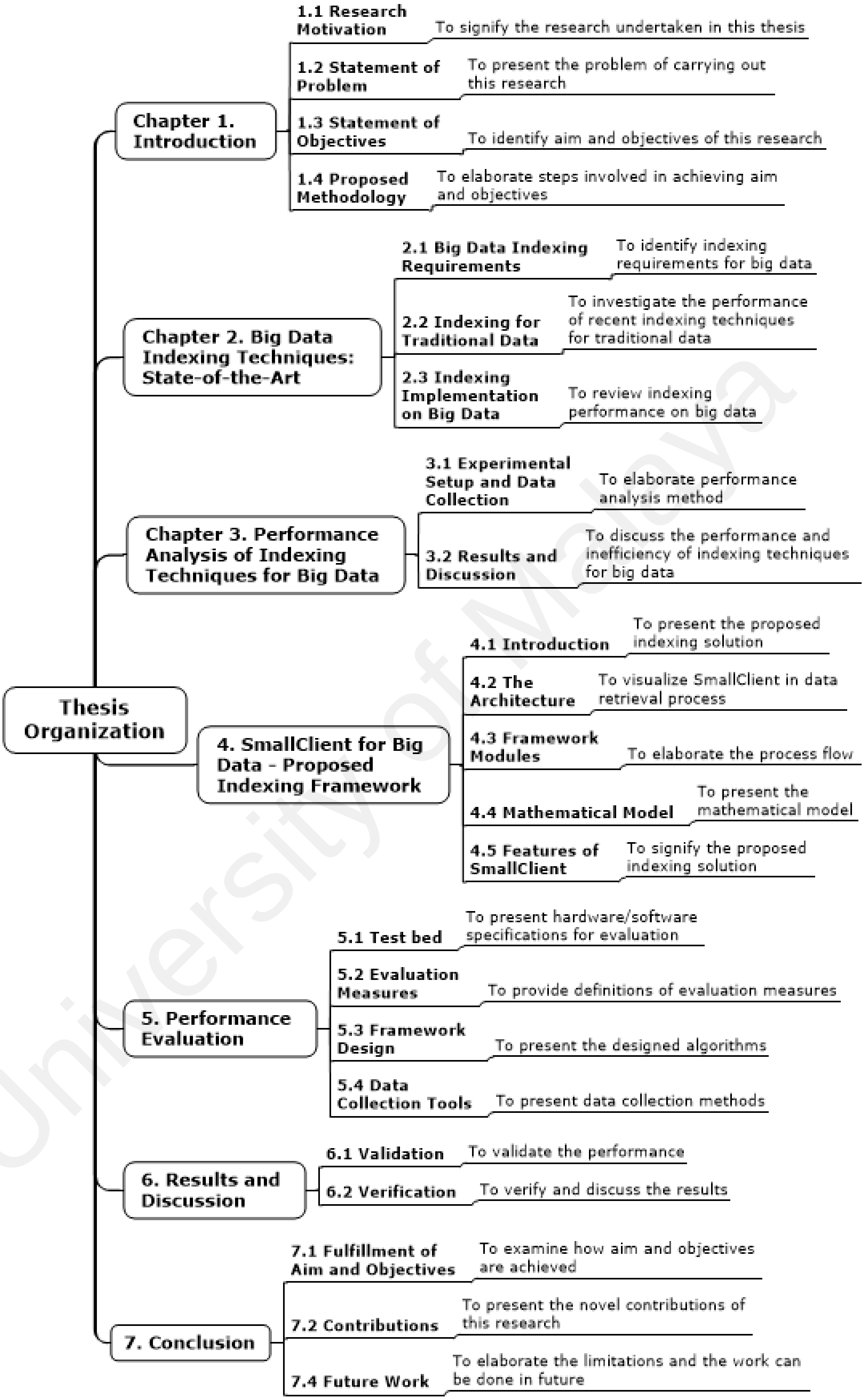

Thesis Organization
Chapter 1. Introduction
1.1 Research Motivation
To signify the research undertaken in this thesis
1.2 Statement of Problem
To present the problem of carrying out this research
1.3 Statement of Objectives
To identify aim and objectives of this research
1.4 Proposed Methodology
To elaborate steps involved in achieving aim and objectives
Chapter 2. Big Data Indexing Techniques: State-of-the-Art
2.1 Big Data Indexing Requirements
To identify indexing requirements for big data
2.2 Indexing for Traditional Data
To investigate the performance of recent indexing techniques for traditional data
2.3 Indexing Implementation on Big Data
To review indexing performance on big data
Chapter 3. Performance Analysis of Indexing Techniques for Big Data
3.1 Experimental Setup and Data Collection
To elaborate performance analysis method
3.2 Results and Discussion
To discuss the performance and inefficiency of indexing techniques for big data
4. SmallClient for Big Data - Proposed Indexing Framework
4.1 Introduction
To present the proposed indexing solution
4.2 The Architecture
To visualize SmallClient in data retrieval process
4.3 Framework Modules
To elaborate the process flow
4.4 Mathematical Model
To present the mathematical model
4.5 Features of SmallClient
To signify the proposed indexing solution
5. Performance Evaluation
5.1 Test bed
To present hardware/software specifications for evaluation
5.2 Evaluation Measures
To provide definitions of evaluation measures
5.3 Framework Design
To present the designed algorithms
5.4 Data Collection Tools
To present data collection methods
6. Results and Discussion
6.1 Validation
To validate the performance
6.2 Verification
To verify and discuss the results
7. Conclusion
7.1 Fulfillment of Aim and Objectives
To examine how aim and objectives are achieved
7.2 Contributions
To present the novel contributions of this research
7.4 Future Work
To elaborate the limitations and the work can be done in future

Figure 1-3: Thesis Organization

Chapter 2 presents a review of big data indexing techniques that enable identifying the potential problems that are related to indexing big data. It first investigates state-of-the-art indexing techniques for traditional data and identifies the indexing requirements related to big data. Then, indexing implementations on big data are reviewed and the potential problems are emphasized.

Chapter 3 presents the performance analysis of contemporary big data indexing techniques to establish the research problem of our thesis. Apache Lucene indexes are used to create indexes on different sizes of data sets to observe indexing overhead and search performance. Based on experimental results, the performance is analyzed and the problem is established by validating the results.

Chapter 4 introduces SmallClient, the proposed indexing framework for big data. The architecture of the data retrieval system and sequence flow of SmallClient to achieve the research objectives is presented in this chapter. The most appealing features of SmallClient are also emphasized in Chapter 4.

Chapter 5 elaborates the evaluation method of our research. Evaluation measures used to collect data and evaluate the solution are discussed in this chapter. Furthermore, the algorithms designed for the framework are presented.

Chapter 6 presents the performance results and their verification. We discuss the findings of collected data for indexing overhead, search performance, and index hit ratio. We use benchmarking and mathematical modeling to further verify the experiment results.

Chapter 7 concludes the thesis. It describes the mapping of the aim and objectives with the research findings. The main contribution of the study is presented in this chapter. Furthermore, the significance of the proposed research and future work is described.

CHAPTER 2: BIG DATA INDEXING TECHNIQUES: THE STATE-OF-THE-ART [1]

This chapter presents a review of indexing techniques for big data to identify potential problems. We first define indexing requirements for big data as investigation criteria and later analyze existing indexing techniques for traditional data by using these requirements for performance investigation. Furthermore, we review indexing mechanisms that have been recently implemented on big data in this chapter. The challenges and potential problems for big data indexing are also presented in this chapter by analyzing contemporary big data indexing implementations.

This chapter comprises four sections. Section 2.1 identifies and elaborates indexing requirements for big data. Section 2.2 presents investigation of contemporary indexing techniques for traditional data by using big data indexing requirements. Section 2.3 investigates indexing advancements in big data under clustered and non-clustered categories. Section 2.4 concludes the discussion.

2.1 Big Data Indexing Requirements

This section discusses the requirements of big data indexing. Accuracy and timeliness are the significant parameters in data retrieval operation performed using an indexing technique. Accuracy of results from data search operations deals with the consistency when same queries are applied whereas timeliness refers to the prompt response on submitted queries. However, knowledge of data requirements to develop an indexing

[1] The work presented in this chapter is partially obtained from the following research contribution:

Gani, Abdullah, Siddiqa, Aisha, Shamshirband, Shahaboddin, & Hanum, Fariza. (2015). A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowledge and Information Systems, 46*(2), 1-44. doi: 10.1007/s10115-015-0830-y

Siddiqa, Aisha, Karim, Ahmad, Gani, Abdullah, & Chang, Victor. On the analysis of big data indexing execution strategies (2016). *Journal of Intelligent and Fuzzy Systems* technique is essential. The literature reports the efficiency and effectiveness of existing indexing techniques when they are applied to traditional data sets (Raghavendra et al., 2016). However, our investigation is related to an analysis of the capability of these techniques to handle big data. We emphasize the indexing requirements that are specifically related to big data for investigation.

Big data refers to voluminous and exponentially growing data generated by heterogeneous resources for which existing technologies become incapable of handling and analyzing these data sets (Philip Chen & Zhang, 2014; Siddiqa, Karim, & Gani, 2016). Big data has its own structural and managerial features; traditional data management technologies are inadequate to deal with big data. Similarly, we analyze recent indexing techniques based on these big data features to prove their inefficiency. The following are the significant features of big data, which we introduce as big data indexing requirements:

- Volume

Volume is related to the size of big data, which is at present measurable in petabytes and is expected to reach zettabytes in the near future (Katal, Wazid, & Goudar, 2013). “Bigness” in the term “big data” refers to its volume. This extensive volume associated with big data is challenging for indexing and requires reduction in query execution time (Chen et al., 2013). Therefore, volume is the most important requirement to be considered when designing an indexing technique for big data.

- Velocity

Velocity refers not only to the rapid and exponential growth in data volume but also to the need to apply query processing (Hashem et al., 2015). With the emergence of big data, query processing trends are also transformed from batch processing to temporal (i.e., monthly, weekly, daily, and hourly) and now the speed requirement for big data analysis is in real time for certain applications. For instance, e-commerce requires management for both speed of data generation and real-time data retrieval for quick decision making (Kaisler, Armour, Espinosa, & Money, 2013).

- Variety

Another structural requirement of big data is handling data generated from various resources such as web pages, web log files, social media sites, e-mails, web documents, and sensor device data. These heterogeneous resources generate data in different formats and data types that bring forth the challenge of big data variety (Kaisler et al., 2013; Philip Chen & Zhang, 2014; Yang et al., 2014). An indexing techniques must be generic to support more than one data format and data type.

- Veracity

Accuracy, reliability, and trustworthiness define the veracity of big data. Big data with exponential generation rate from heterogeneous resources should ensure that data are in fact sufficiently accurate, and not spoofed, corrupted, or obtained from an expected source. This is an important issue known as big data veracity (X. Wang, Luo, & Liu, 2014). Accuracy of results for query execution is required by an indexing technique that addresses big data veracity.

- Variability

Variability handles inconsistencies in big data flow. Data loads become difficult to maintain, especially with the increasing usage of social media, which generally causes peaks in data loads when certain events occur (Katal et al., 2013). Variability brings the challenge for indexing techniques to ensure timeliness and accuracy of results for submitted queries.

- Value

Value refers to the insights and benefits that are obtained by keeping and managing such big data. Data usefulness in decision making defines its value. The accuracy and timeliness of results is significant to increase insight when insight is preferable to quantity (Kaisler et al., 2013; LaValle, Lesser, Shockley, Hopkins, & Kruschwitz, 2013).

- Complexity

Big data structures have a high degree of interconnectedness and dependencies (Kaisler et al., 2013). The challenges that are related to big data complexity are its linking, matching, cleansing, and transformation across systems (Siddiqa, TargioHashem, et al., 2016). However, connecting and correlating relationships, hierarchies, and multiple data linkages are also very important. If complexity in terms of these objectives is not considered, big data cannot be organized effectively (Barbierato, Gribaudo, & Iacono, 2014).

Volume, velocity, and variety are the structural features of big data, whereas the rest of the features are related to the managerial aspect. Structural features are the essential requirements for a system that is designed to handle big data. Therefore, one proposal is to consider the structural features of big data when designing a big data management system, specifically an indexing mechanism. An indexing technique is proven efficient when it satisfies requirements such as large volume, rapid growth, and heterogeneous data types along with efficiency of the indexing procedure itself.

By focusing on volume, velocity, and variety of big data, an indexing technique is efficient when indexing cost in terms of index creation time and index size is low. Meanwhile, query execution time must also be minimized to ensure effectiveness of an indexing technique for big data.

2.2 Indexing for Traditional Data

In this section, we present state-of-the-art indexing techniques that are found in the latest literature for traditional data. We review the recent indexing techniques for traditional data and categorize them based on their adopted procedures. We define each category and explain the characteristics of each technique in these categories. We also analyze the performance of these indexing techniques through big data indexing requirements that are elaborated in Section 2.1.

An indexing mechanism facilitates the data search and retrieval tasks when data sets comprise an enormous number of records and when scanning the whole data set incurs high operational cost (M. Wang, Holub, Murphy, & O'Sullivan, 2013). Indexing improves the performance of query operations and reduces data retrieval time for search queries over high-volume data sets (Chen et al., 2013). Therefore, indexing is an essential task for a data analysis system in terms of effectiveness of performing complex queries and accessing larger-sized data sets.

Recent research advancements indicate that various indexing mechanisms have been adopted based on the nature of data and type of data analysis. For instance, semantic indexing is used for enhanced search procedures for big data on cloud (Rodríguez-García, Valencia-García, García-Sánchez, and Samper-Zapater (2013), an inverted index method for event stream indexing on a large text collection in a distributed environment (Cambazoglu, Kayaaslan, Jonassen, & Aykanat, 2013), and R-Tree indexing on multi-dimensional data (J. Wang, Wu, Gao, Li, & Ooi, 2010). The requirement of consuming less time and cost to apply search operation has become critical for high-volume and continuously growing big data (Kadiyala & Shiri, 2008). Thus, the study shows continuous improvement in the implementation procedure of indexing.

2.2.1 Classification of Indexing Techniques

We categorize contemporary indexing techniques and devise a taxonomy comprising three categories: non-artificial intelligence (NAI), artificial intelligence (AI), and collaborative artificial intelligence (CAI). We present the taxonomy of existing indexing techniques in Figure 2-1. These techniques are discussed in this section.

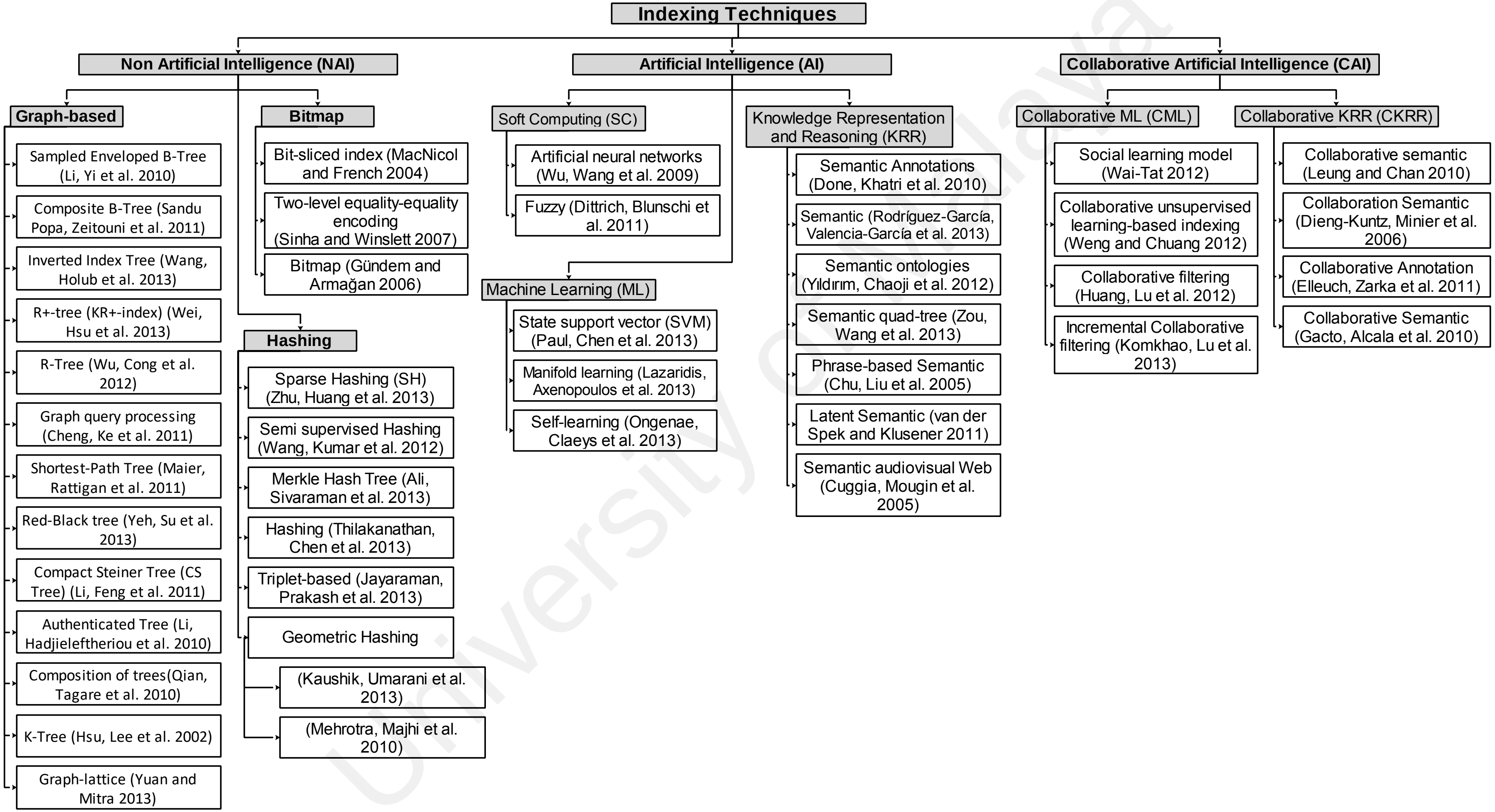

Indexing Techniques
Non Artificial Intelligence (NAI)
Artificial Intelligence (AI)
Collaborative Artificial Intelligence (CAI)
Graph-based
Bitmap
Soft Computing (SC)
Knowledge Representation and Reasoning (KRR)
Collaborative ML (CML)
Collaborative KRR (CKRR)
Sampled Enveloped B-Tree (Li, Yi et al. 2010)
Composite B-Tree (Sandu Popa, Zeitouni et al. 2011)
Inverted Index Tree (Wang, Holub et al. 2013)
R+-tree (KR+-index) (Wei, Hsu et al. 2013)
R-Tree (Wu, Cong et al. 2012)
Graph query processing (Cheng, Ke et al. 2011)
Shortest-Path Tree (Maier, Rattigan et al. 2011)
Red-Black tree (Yeh, Su et al. 2013)
Compact Steiner Tree (CS Tree) (Li, Feng et al. 2011)
Authenticated Tree (Li, Hadjieleftheriou et al. 2010)
Composition of trees(Qian, Tagare et al. 2010)
K-Tree (Hsu, Lee et al. 2002)
Graph-lattice (Yuan and Mitra 2013)
Bit-sliced index (MacNicol and French 2004)
Two-level equality-equality encoding (Sinha and Winslett 2007)
Bitmap (Gündem and Armağan 2006)
Hashing
Sparse Hashing (SH) (Zhu, Huang et al. 2013)
Semi supervised Hashing (Wang, Kumar et al. 2012)
Merkle Hash Tree (Ali, Sivaraman et al. 2013)
Hashing (Thilakanathan, Chen et al. 2013)
Triplet-based (Jayaraman, Prakash et al. 2013)
Geometric Hashing
(Kaushik, Umarani et al. 2013)
(Mehrotra, Majhi et al. 2010)
Artificial neural networks (Wu, Wang et al. 2009)
Fuzzy (Dittrich, Blunschi et al. 2011)
Machine Learning (ML)
State support vector (SVM) (Paul, Chen et al. 2013)
Manifold learning (Lazaridis, Axenopoulos et al. 2013)
Self-learning (Ongenae, Claeys et al. 2013)
Semantic Annotations (Done, Khatri et al. 2010)
Semantic (Rodríguez-García, Valencia-García et al. 2013)
Semantic ontologies (Yıldırım, Chaoji et al. 2012)
Semantic quad-tree (Zou, Wang et al. 2013)
Phrase-based Semantic (Chu, Liu et al. 2005)
Latent Semantic (van der Spek and Klusener 2011)
Semantic audiovisual Web (Cuggia, Mougin et al. 2005)
Social learning model (Wai-Tat 2012)
Collaborative unsupervised learning-based indexing (Weng and Chuang 2012)
Collaborative filtering (Huang, Lu et al. 2012)
Incremental Collaborative filtering (Komkhao, Lu et al. 2013)
Collaborative semantic (Leung and Chan 2010)
Collaboration Semantic (Dieng-Kuntz, Minier et al. 2006)
Collaborative Annotation (Elleuch, Zarka et al. 2011)
Collaborative Semantic (Gacto, Alcala et al. 2010)

Figure 2-1: Taxonomy of Indexing Techniques (Gani, Siddiqa, Shamshirband, & Hanum, 2015)

Figure 2-1 presents the detailed taxonomy and divides indexing techniques into the NAI, AI, and CAI categories. NAI is further categorized as graph-based, bitmap, and hashing. AI involves soft computing (SC), machine learning (ML), and knowledge representation and reasoning (KRR). CAI consists of subcategories, namely collaborative machine learning (CML) and collaborative knowledge representation and reasoning (CKRR).

We define each category of taxonomy and explain the indexing techniques under each category in the rest of this section. We also discuss the advantages and limitations of implementing these techniques. We summarize all indexing techniques later in Table 2-1.

- Non-artificial Intelligence (NAI) Techniques

The NAI category comprises indexing techniques that have straightforward procedures for index creation and query execution. Furthermore, these techniques are adaptable to fast data retrieval requirements. Therefore, NAI indexing techniques are widely used in the literature. Graph-based indexes develop a tree structure that improves traversal and data retrieval performance (Zhang et al., 2015). B-Tree and B+-Tree are used to index the under graph-based category of NAI. Sampled envelop B-Tree (F. Li, Yi, & Le, 2010) uses near-linear time to answer any top-k (t) query with optimal I/O cost expected.

Similarly, R+-Tree and R-Tree also reside in graph-based NAI indexing techniques that are used for efficient data retrieval on range and nearest-neighbor queries. KR+-index (Wei, Hsu, Peng, & Lee, 2013) is designed using R+-Tree for skewed spatial data. However, for keyword search queries, the compact Steiner tree (G. Li, Feng, Zhou, & Wang, 2011) outperforms other indexing mechanisms, thereby reducing the implementation cost. The Steiner tree can be seamlessly integrated with any existing RDBMS.

The second subcategory of NAI indexing techniques is bitmap, which is considered as an effective indexing mechanism for range queries on append-only data (K. Wu, Shoshani, & Stockinger, 2010). In bitmap indexing structure, bulk index data is stored as a sequence of bits; this bit sequence is used to answer queries. A bit-sliced index (MacNicol & French, 2004) adopts binary encoding, which reduces the number of bitmaps. However, compared with other methods, bit-sliced index still consumes more time in query execution.

Hashing methods of indexing for high-dimensional data uses the least time in executing approximate similarity search queries (Shang, Yang, Wang, Chan, & Hua, 2010). Hashing represents high-dimensional data into compact binary codes to increase query execution performance. Sparse hashing (Zhu, Huang, Cheng, Cui, & Shen, 2013) also performs better for approximate similarity search queries by converting the original feature space of data into low-dimensional data.

- Artificial Intelligence (AI) Techniques

The AI category of indexing techniques involves techniques that are highly technical and specialized; they use a knowledge base for efficient data retrieval. This category includes soft computing (SC), machine learning (ML), and knowledge representation and reasoning (KRR) methods of indexing. A prominent feature of AI indexing techniques is that each technique involves training the indexing model as a prerequisite for labeled data. This training process requires immense computational resources. Therefore, AI indexing needs more computational resources than does NAI indexing.

SC includes artificial neural networks and fuzzy based methods for AI indexing. The hierarchical tree (S. Wu, Wang, & Xia, 2009) is designed using the artificial neural network method for efficient indexing and data retrieval for human motion data.

Hierarchical tree indexing consumes more time in artificial neural network-based unsupervised learning. Fuzzy rule-based indexing is also efficient for indexing moving objects; index creation time is minimal (Dittrich, Blunschi, & Vaz Salles, 2011). However, dealing with unknown events in data is not possible in fuzzy rule base. Therefore, hybrid fuzzy classifiers (Bordogna, Pagani, & Pasi, 2006) are adopted to dynamically adjust the rule to ensure better detection ratio.

In the AI category, ML-based indexing introduces an iterative process of observing patterns in data to make predictions. Multi-model descriptor index (Lazaridis, Axenopoulos, Rafailidis, & Daras, 2013) uses manifold learning, which optimizes the search and retrieval process for large data sets. The initial state support vector machine network (Chen-Yu, Ta-Cheng, Jhing-Fa, & Li Pang, 2009), which models human behavior in surveillance situations, is followed by Paul et al. (2013). This study generates probabilistic scores that are based on input frames and computes transition probability using training data.

KRR indexing assigns tags to documents and semantics to data obtained from a user response on a system such as a social network. Semantic indexing (Y. Wang, 2008) stores annotations of a document by assigning them weights and finds closeness in scores of semantics. An enhanced approach in semantic indexing (Rodríguez-García et al., 2013) is applied on ontologies to retrieve information on cloud resources based on user needs. KRR indexing improves knowledge discovery and decision making for large data sets.

- Collaborative Artificial Intelligence (CAI) Techniques

Indexing techniques under the CAI category combine two or more indexing techniques to improve accuracy and search efficiency (Gacto, Alcala, & Herrera, 2010). This category includes collaborative machine learning (CML) and collaborative knowledge representation and reasoning (CKRR) methods.

CML-based indexing (Wai-Tat, 2012) for various types of data is designed using social learning that combines KRR to induce semantic indexing. This method of collaborative indexing assists with enhanced representation of semantics and makes them easily interpretable for users who may have different knowledge backgrounds or information needs. CML indexing also supports in developing recommendation systems to retrieve more accurate and precise information for keyword search queries in medical knowledge (Huang, Lu, Duan, & Zhao, 2012). These recommendation systems use expert suggestions and user profiles to ensure an accurate data retrieval process.

Under the CKKR category of CAI indexing techniques, the semantic indexing method (Dieng-Kuntz et al., 2006) combines the graph-based method to design a knowledge management model in medical ontology as a “virtual staff” tool that provides collaborative diagnosis.

We summarize the indexing techniques and emphasize their distinctive features in Table 2-1. The Table 2-1comprises indexing techniques under the NAI, AI, and CAI categories. These techniques include the application domain in which these indexing techniques are implemented, data set availability and type of data set used in evaluation, objectives of designing indexing techniques, and salient features.

Table 2-1: Indexing Techniques for Traditional Data (Gani et al., 2015)

Method			Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
Non-Artificial Intelligent Methods (NAI)							
Graph-based	Tree-based	Sampled Envelope (SE) B-tree for top-k queries (F. Li, Yi, et al., 2010)	L	P_1/ P_2	Temporal	To design a simple and efficient indexing for ranking queries on temporal data	• Simple structure of indexing • Index takes less space • Its creation is faster • Small increase in creation cost when variance of data increases • Faster query response • Less update cost
		Graph partitioning and a composite B+-Tree (Sandu Popa, Zeitouni, Oria, Barth, & Vial, 2011)	L	P_1	Trajectory	To provide efficient indexing for trajectories of moving objects in a network	• Faster query response even when query size and data size is increased • Less update cost which increases gradually
		Inverted Index Tree (M. Wang et al., 2013)	C	P_1	Event Stream (Log)	To design an index for multiple keyword-based queries on generic stream data where bidirectional reference are created between leaf nodes and event indices so that	• Index take less space but takes more time to load in memory • Manageable query processing cost • Faster query response

i. [2]Performed **application** domain: Cloud (C), Network (N), and Local data on a single computer (L)

ii. [3]Type of **applied** data set: Public (P_1), Private (P_2), and Unspecified (U)

Method			Application domain^2	Applied data set^3	Data set Type	Objectives	Features
						CPU cost is reduced and efficient indexing is achieved	
		R+-tree (KR+-index) (Wei et al., 2013)	C	P_1	Spatial	To present a novel multidimensional key design index based on an R+-tree KR+ index for efficient search and retrieval of skewed spatial data	• Index takes more space • Query response time depends upon query size and data size • Scalable for large data
		R-Tree (D. Wu, Cong, & Jensen, 2012)	N	P_1	Spatial	To design a hybrid inverted file R-Tree to retrieve text and query spatial proximity	• Index takes more space • Query response depends upon buffer size • Less query processing cost
		A graph query processing index system (Cheng, Ke, Fu, & Yu, 2011)	L	P_1/ P_2	Graph	Design a graph querying system that achieves both fast indexing and efficient query processing	• Index take less space • Faster index creation • Faster query response • Less update cost • Scalable for large data and query response time remains the same
		A network structure index with Shortest-Path Trees (Maier, Rattigan, & Jensen, 2011)	L	P_1/ P_2	Graph (network path)	To present and design an indexing technique for auxiliary data structures that provides fast look-ups for common operations	• Index takes linear space • Efficient search for common operations • Accurate query results • Applicable on real data sets • More computational costs for large networks
		A framework of Red–Black tree as an efficient and	C	U	Text	To present a Red–Black tree framework for big data cloud collaborative editing	• Less index creation cost • Less update time • Reduced data encryption overhead

Method			Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
		secure approach (Yeh, Su, Chen, & Lin, 2013)					• Efficient encryption compared to 3DES encryption and AES encryption
		Compact Steiner Tree (CS Tree) (G. Li et al., 2011)	L	P_1	Graph	To optimize the Steiner tree to answer keyword queries more efficiently, to effectively implement keyword search, and to utilize DBMS capabilities	• Accurate query results • Faster query response
		Authenticated Tree- Based Index Structures (F. Li, Hadjieleftheriou, Kollios, & Reyzin, 2010)	N	P_1/ P_2	Spatial	To develop efficient index structures for authenticating aggregation queries over large data sets	• More accurate query results • Less query execution cost • Dynamic index updating
		Composition of Coordinate tree, metric tree, and *kd*-tree (Qian, Tagare, Fulbright, Long, & Antani, 2010)	L	P_1	Image	To present an optimal shape embedding procedure to index shapes for complete and partial shape similarity retrieval	• Less index computational cost • Fast query response • Less query execution cost
		K-Tree (Hsu et al., 2002)	L	P_2	Image	To develop a new indexing method called K-tree to process RkNN	• Faster query response • Accurate query results

Method		**Application domain**[2]	**Applied data set**[3]	**Data set Type**	**Objectives**	**Features**
					(Reverse k-Nearest Neighbors) queries efficiently	
	A graph-lattice-based index (Yuan & Mitra, 2013)	L	P_1	Graph	To describe indexing techniques based on sub-graphs	• Fast query response • Index creation is faster and easy • Index update is faster • Faster query results for sub-graph-querying • False graphs can be filtered easily
Bitmap	Bit-sliced index (MacNicol & French, 2004)	N	U	Transactional	To present a multi-component bitmap index created from three basic encoding schemes	• Index takes less space • Less query processing cost • Querying is slower than multi-level indexes
	Two-level equality-equality encoding (Sinha & Winslett, 2007)	L / N	P_1	Hierarchical Data Format	To propose multi-resolution and parallelizable bitmap indexes	• Index takes more space • Faster query response • Better results for range queries • Index is scalable in cluster environment
Hashing	A novel Sparse Hashing (SH) method (Zhu et al., 2013)	L	P_1/ P_2	Image, Text	To develop a novel sparse hashing (SH) method for fast approximate similarity searches	• Accurate query results for large data sets • More index computational cost • More training cost for large data set • Fast encoding
	Merkle Hash Tree (Ali, Sivaraman, & Ostry, 2013)	C	P_2	Real-time	To design an authentication scheme to detect loss in data using the Merkle hash tree	• Accurate query results (90%) • Less creation cost
	Hashing (Thilakanathan, Chen,	C	P_2	Medical (ECG)	Design a system to ensure fast healthcare data download using a hash function	• Efficient query response for large data set • More initial setup time

Method			Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
	Nepal, Calvo, & Alem, 2013)						
	Triplet-based Hashing (Jayaraman, Prakash, & Gupta, 2013)		L	P_1	Medical (ECG)	To propose an indexing technique for a biometric image database	• Index takes less space • Less computational cost • Invariant to scaling
	Geometric hashing	(Kaushik, Umarani, Gupta, Gupta, & Gupta, 2013)	L	P_1	Image (face)	To present an efficient scheme to index a database of facial images	• Index takes less space • Less computational cost • Accurate query results
		(Mehrotra, Majhi, & Gupta, 2010)	L	P_1	Image (Iris)	To propose an efficient indexing scheme for searching a large iris biometric database	• Index takes less space • Fast query response • More accurate query results • Robust in similarity transformations as well as occlusion • Capable of localizing iris images with change in gaze, occlusion, and illumination
HubRank (Chakrabarti, Pathak, & Gupta, 2011)			L	P_1	Graph	To design an efficient index for consistent results of PageRank query	• Index takes less space • Accurate query results • Less index creation time • Fast query response • Efficient query processing
A novel term-based inverted index partitioning model that relies on			N	P_1	Text	To minimize the communication overhead that will be incurred by future queries	• Index takes less space • More computational cost • Scalable index

Method		Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
hypergraph partitioning (Cambazoglu et al., 2013)						
A Compressed Permuterm Index (CPI) (Ferragina & Venturini, 2010)		L	P_1	Graph	To propose a Compressed Permuterm Index which supports fast queries	• Index takes less space • Fast query results • Easy updating
Three-level Indexing Hierarchy (TIH) (C.-H. Wang et al., 2010)		L	P_2	Multimedia (video)	To present a novel indexing architecture in order to support a range of smart playback functions in collaborative telemedicine systems	• Simple index • Index takes less space • Less computational cost • Accurate query results
Artificial Intelligent Methods (AI)						
Soft Computing (SC)	A hierarchical tree based on artificial neural networks (S. Wu et al., 2009)	L	P_1	Motion data	To develop an efficient indexing and retrieval approach for human motion data	• Fast query response • Accurate query results • More time consuming for artificial neural network-based unsupervised learning
	Fuzzy (Dittrich et al., 2011)	N	P_2	Road Network	To design an indexing technique for such application where objects are moving at a high update rate	• Index takes less space • Less index creation time • Faster index update • Faster query response time • Scalable • Index image is created frequently so it is time consuming

Method		Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
Machine Learning (ML)	State support vector (SVM) (Paul et al., 2013)	L	P_1	Multimedia	To present a video search and indexing system based on the state support vector (SVM) network, video graph, and reinforcement agent	• Less index creation time • Accurate query results • Time consuming at Learning stage
	Multimodal descriptor indexing based on manifold learning (Lazaridis et al., 2013)	L	P_1	Multimedia (Audiovisual)	To propose a complete solution for search and retrieval of rich multimedia content over modern databases	• Less index creation time • Less index creation cost • Faster query response • Scalable • Time consuming for manifold learning method
	Self-learning (Ongenae et al., 2013)	L	P_1	Temporal	To propose a self-learning, probabilistic, ontology-based framework which allows healthcare context-aware applications to adapt their behavior to run-time	• Fast query response • Accurate query results
Knowledge Representation and	Semantic Annotations (Done, Khatri, Done, & Draghici, 2010)	L	P_1/ P_2	Annotated	Design a technique to detect Gene Ontology annotations with the help of finding relationships between genes and functions	• Accurate query results
	Semantic (Rodríguez-García et al., 2013)	C	P_1	Annotated	Offer a platform to facilitate the retrieval and selection of cloud resources on the basis of keyword search query meeting the users' needs	• Automatic index updating • Fast query response • Accurate query results • Applicable to unstructured documents • More information required to provide enough accuracy

Method		Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
						• It supports only keyword-based queries • To ensure accuracy it needs more knowledge
	Scalable reachability index (GRAIL) based on semantic ontologies (Yıldırım, Chaoji, & Zaki, 2012)	L	P_1/ P_2	Graph	Propose randomized interval labeling based on the graph theory.	• Simple Index • Fast query response for large graphs • Scalable • Comparatively low performance for small graphs
	semantic quad-tree and Chord ring (Zou, Wang, Cao, Qu, & Wang, 2013)	N	P_2	Spatial	To present a novel semantic overlay network for large-scale multi-dimensional spatial information indexing	• Scalable • Supports complex range queries
	Phrase-based Semantic (Chu, Liu, Mao, & Zou, 2005)	L	P_1	Text	To present a new knowledge-based approach to support scenario-specific retrieval applicable in healthcare monitoring	• Fast query response in real time • Accurate query results
	Latent Semantic (van der Spek & Klusener, 2011)	L	P_1	Text	To apply a dynamic threshold to improve cluster detection of LSI (Latent Semantic Indexing)	• Applicable to large document sets • Fully automated
	Semantic audiovisual Web indexing (Cuggia, Mougin, & Beux, 2005)	N	U	Multimedia (Video)	To propose an audiovisual Web indexing system for medical audiovisual resources	• Simple index • Demonstrates possibilities of conceptual indexing based on medical ontologies
Collaborative Artificial Intelligent Methods (CAI)						
Colla	Social learning model utilized in	N	P_2	Folksonom y	A machine learning based approach to present a social learning model	• Faster query response • Supports structuring of information

Method		Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
	collaborative indexing (Wai-Tat, 2012)				which is, in collaboration with knowledge representation, applied as collaborative indexing for retrieval of relevant documents and knowledge exploration	• Scalable for large data • Efficient in semantic representation • Efficient human-system integration • Learning is time consuming
	Collaborative unsupervised learning-based indexing via matrix factorization (Weng & Chuang, 2012)	L	P_1	Multimedia (Video)	To present a recommendation system of unsupervised video re-indexing developed based on collaborative filtering approach which refines and improves the indexing scores generated by concept classifiers	• Faster Index creation • More query response cost • Accurate in query results
	Collaborative filtering based Medical Recommendation System (Huang et al., 2012)	N or L	P_2	Clinical	To develop a collaborative filtering based medical knowledge recommendation system so that clinicians can retrieve trust-based accurate knowledge	• Faster query response • Accurate in query results • More human effort is required in recommendation recording • Motivation is required in recording recommendation
	Incremental Collaborative Filtering based Recommender System (Komkhao, Lu, Li, & Halang, 2013)	L	P_1	Text	To design a model-based collaborating filtering technique to improve the accuracy and scalability of recommender system	• More accurate query results • Scalable and the performance is improved for larger training data set
Colla	Collaborative semantic (Leung & Chan, 2010)	L	U	Multimedia (music)	To design a collaborative semantic indexing and metadata based	• Accurate in query results • Accuracy increases as the index is updated

Method	Application domain[2]	Applied data set[3]	Data set Type	Objectives	Features
				retrieval for music information so that accurate results are available to users To design an approach for deep content-based music information retrieval	• Index size is gradually increasing • Fault tolerant • Resilient, community validated structure • Eliminates inappropriate index terms
Collaboration-based Semantic Indexing (Dieng-Kuntz et al., 2006)	N	P_1/ P_2	Cognitive (concept based)	To present a method for reconstituting a medical ontology by translating a medical database into RDF language in the context of a healthcare network. A virtual staff is developed where more number of healthcare members are involved for better diagnosis	• Guaranteed knowledge management • Useful for a healthcare network dedicated to heavy pathology • Accurate in query results
Collaborative Annotation (Elleuch, Zarka, Ammar, & Alimi, 2011)	L	P_1	Multimedia (video)	To improve the semantic concept detection process through collaboration of fuzzy with ontology	• Improvement in accuracy of query results • Improvement in precision of context and concept detection • More relevant query results
Collaborative Semantic (Gacto et al., 2010)	L	P_1	Regression	To design an index for natural language context preserving to make it simple and more interpretable	• More accurate query results • Results are more interpretable

i. [1]Performed application domain: Cloud (C), Network (N), and Local data on a single computer (L)

ii. [2]Type of applied data set: Public (P1), Private (P2), and Unspecified (U)

Table 2-1 shows that graph-based NAI indexing techniques are usually adopted to create indexes for graph, temporal, spatial, image, and text types of data and result in reduced index creation time and small index size. Graph-based NAI indexing techniques provide an indexing structure that aims to fasten query execution and data retrieval process. Bitmap indexes in the NAI category are designed for transactional and hierarchical data formats. Bitmap indexes do not always guarantee a low index creation cost in terms of index creation time and index size. Most of the applications of hashing are for image data indexing. Although some hashing implementations for indexing reduce the index size, the computational cost for index creation or initialization is not guaranteed with the hashing method. Therefore, graph-based indexing techniques are widespread in efficient data retrieval systems for various types of data sets.

AI indexing techniques are usually adopted to index multimedia, motion, and temporal data. However, the KRR subcategory of AI indexing techniques is mostly used for annotation and text data. SC and ML indexing techniques ensure less query execution time and need less time in index creation, whereas initial learning for these mechanisms is time consuming, thereby increasing the delay to start query execution. Similarly, KRR indexing methods also ensure less query execution time. However, KRR implements semantic logic for indexing; thus, it is unsuitable for schema-based data.

CAI indexing techniques offer collaborations of more than one mechanism for better indexing solution. In CML, learning methods usually adopt collaborative filtering and KRR methods to increase the accuracy of query results. Most CML indexing mechanisms are scalable, but additional computational cost is required. Similarly, CKRR-based indexing methods are also designed to increase the accuracy of results. For example, collaborative annotation (Elleuch et al., 2011) is a CKRR approach that integrates fuzzy soft computing with ontology for improved semantic detection and ensures more relevant results for queries.

2.2.2 Analysis of Indexing Techniques for Big Data Indexing Requirements

In this section, we investigate indexing techniques that are discussed in Section 2.2.1. We analyze the fulfillment of indexing requirements related to big data by investigating the support of the discussed indexing techniques for each derived criterion, i.e., volume, velocity, veracity, variability, value, and complexity (see Table 2-2). The analysis leads to an assessment of the viability of these indexing techniques for big data.

Table 2-2: Analysis of Indexing Techniques for Big Data Indexing Requirements (Gani et al., 2015)

Indexing Method		**Authors**	**Big data indexing requirements**[4]						
			Volume	**Velocity**	**Variety**	**Veracity**	**Variability**	**Value**	**Complexity**
Non- artificial intelligence Indexing (NAI)									
Graph-based	B-Tree	Li, Yi et al. (2010)	√	NA	NA	NA	√	√	NA
	R+-tree	Wei, Hsu et al. (2013)	√	NA	×	NA	√	NA	NA
	Suffix Tree	Russo, Navarro et al. (2008)	√	×	NA	NA	NA	NA	NA
	Graph Query Tree	Cheng, Ke et al. (2011)	√	√	NA	NA	NA	NA	√
	Shortest Path Tree	Maier, Rattigan et al. (2011)	√	NA	√	√	√	√	NA
	Red–Black tree	Yeh, Su et al. (2013)	√	√	×	√	NA	√	√
Bitmap		Wu, Shoshani et al. (2010)	√	√	×	NA	√	×	√
Hashing	Hashing	Zhu, Huang et al. (2013)	√	×	√	×	NA	NA	NA
	Geometric hashing	Mehrotra, Majhi et al. (2010)	√	√	×	√	√	NA	√
Inverted index		Cambazoglu, Kayaaslan et al. (2013)	√	NA	×	NA	NA	NA	√
Lazy Indexing		Richter, Quiané-Ruiz et al. (2012)	×	NA	×	√	NA	√	NA
Artificial Intelligence (AI)									
Semantic Indexing		Rodríguez-García, Valencia-García et al. (2013)	√	NA	√	√	√	√	NA

[4] Big Data Indexing Requirements: √ = Satisfied, × = Not Satisfied, NA = Not Applicable

Indexing Method	Authors	Big data indexing requirements[4]						
		Volume	**Velocity**	**Variety**	**Veracity**	**Variability**	**Value**	**Complexity**
	Done, Khatri et al. (2010)	√	×	√	√	NA	NA	NA
Manifold Learning	Lazaridis, Axenopoulos et al. (2013)	√	√	√	√	NA	NA	√
Fuzzy	Dittrich, Blunschi et al. (2011)	√	√	×	×	NA	√	√
Support Vector Machine	Paul, Chen et al. (2013)	NA	NA	×	√	NA	√	NA
Randomized interval labeling	Yıldırım, Chaoji et al. (2012)	√	√	×	√	√	NA	√
Hierarchical Tree	Wu, Wang et al. (2009)	√	√	×	√	NA	NA	√
Collaborative Artificial Intelligence (CAI)								
Collaborative Semantic	Leung and Chan (2010)	√	NA	×	√	NA	√	NA
	Dieng-Kuntz, Minier et al. (2006)	√	NA	×	NA	NA	√	NA
	Gacto, Alcala et al. (2010)	NA	NA	NA	√	NA	NA	NA
Collaborative filtering technique	Weng and Chuang (2012)	√	√	√	√	NA	NA	NA
	Huang, Lu et al. (2012)		NA	×	√	×	√	×
Incremental Collaborative Filtering	Komkhao, Lu et al. (2013)	√	NA	NA	√	NA	×	NA
Collaborative learning	Wai-Tat (Wai-Tat, 2012)	√	NA	√	NA	NA	NA	NA
Collaborative Annotation	Elleuch, Zarka et al. (2011)	√	NA	NA	√	NA	NA	NA
Big Data Indexing Requirements: √ = Satisfied, × = Not Satisfied, NA = Not Applicable								

In Table 2-2, we present an investigation of NAI, AI, and CAI indexing techniques based on big data indexing requirements. We show that an indexing technique satisfies one or more indexing requirements of big data, thereby leading a researcher to select a technique based on the preference for analyzed data set.

NAI indexing techniques mostly satisfy a large volume requirement of big data. The suffix tree, for instance, ensures efficiency when implemented on large volume data supports, thereby revealing its feasibility for big data (Russo et al., 2008). Hashing (Zhu et al., 2013), inverted index (Cambazoglu et al., 2013), and other NAI methods also show their support to large volumes of data. B-Tree is capable of dealing with volume, variability, and value of data. Furthermore, query execution time is very low. Thus, the efficiency of B-Tree for big data is confirmed (F. Li, Yi, et al., 2010).

AI-based indexing is more concerned about accuracy of results. The techniques such as semantic (Rodríguez-García et al., 2013) and support vector machine (Paul et al., 2013) show improved performance in terms of veracity and value. Rodríguez-García et al. (2013) obtained up to 300 different services of various data types from the ICT domain to validate the semantic technique results. They found an 88% precision value for accuracy. Therefore, we recommend choosing an AI mechanism for indexing when the data analysis procedure aims to show accurate results.

The CAI methods for indexing fulfill volume, veracity, and value requirements of big data (Leung & Chan, 2010). The results obtained from semantic KRR indexing (Leung & Chan, 2010) implementation on large digital music data demonstrate the robustness, fault tolerance, and data retrieval efficiency of this method.

Precisely, an indexing technique is considered efficient for big data, thereby satisfying volume, velocity, and variety requirements. Minimum indexing cost in terms of both index creation time and index size, and minimum query execution and data retrieval time also prove the effectiveness of the indexing technique for big data.

2.3 Indexing Implementation on Big Data

In this section, we review indexing techniques that are designed to support big data indexing requirements as well as to facilitate query execution and search performance for big data. We have previously described that contemporary technologies are becoming inefficient to meet capture, preparation, analysis, and visualization requirements of big data (Kwon, Lee, & Shin, 2014). Thus, big data bring new challenges in processing such as quick and up-to-date responses to search queries and in-time data availability. The need for fast data processing and timely responses associated with big data are used to evaluate the performance of indexing and search process so that challenges revealed by the emergence of big data can be emphasized.

Numerous solutions have been proposed by researchers to improve the efficiency of the search and data retrieval process for voluminous data records. Some examples are vertical partitioning (Jindal, Quiané-Ruiz, & Dittrich, 2011), clustered attribute-based indexing (Dittrich et al., 2012; Dittrich et al., 2010) for distributed parallel processing systems, and clustered adaptive indexing (Richter et al., 2012) for changing query workload. Likewise, in medical research, a batch processing-based image retrieval system (Zhuang, Jiang, Li, Chen, & Ju, 2015) contributes in scheduling multiple query requests; minimized response time is achieved when large distributed image data sets face the problem of multi-query optimization. Consequently, for distributed and replicated big data storage systems, an efficient indexing technique is needed to serve a larger number of queries for improved search performance.

We categorize big data indexing techniques as clustered and non-clustered approaches to illustrate them effectively. Chaudhuri, Datar, and Narasayya (2004) define these categories as follows: the clustered approach of indexing physically reorders data according to the values of indexed column(s), whereas the non-clustered approach creates a redundant index structure for a data set. Clustered approaches use sorting algorithms to reorder data, whereas non-clustered approaches use the indexing techniques as discussed in Section 2.1. We provide a comparative chart for both indexing categories in Table 2-3.

Table 2-3: Comparison of Clustered and Non-Clustered Indexing

Features	Clustered Indexing	Non-Clustered Index
Process	Physically sorts and stores data rows	Separate structure containing key-value. Key is the content of indexed attribute and the value is pointer to the row/ indexed attribute
No. of Indexes	One replica can have one index	Single replica can have as many indexes as required
Index Size	Less size	Separate structure needs more space. However, creating index is less costly than creating separate replica
Index Updating	Index rebuilding needs re-ordering whole data	Index rebuilding is easy (delete and create new)
Data write	Slow (requires re-ordering)	Each index of last data block is updated
Data read	Fast (searches in sorted list)	First traverses index then jumps to record

Fast query execution and data retrieval are the main challenges for big data that are distributed over clusters of heterogeneous machines. Researchers are interested in accepting these challenges and they have focused on exploiting various methods to optimize search performance for such big data. To date, many indexing approaches are available to perform fast search operations on big data on distributed parallel systems. However, these approaches have unaddressed challenges. We describe both clustered and non-clustered indexing implementation on big data and emphasize their potential problems in the following subsections.

2.3.1 Clustered Indexing on Big Data

This section presents clustered indexing approaches for big data. Clustered indexing approaches are implemented over Hadoop, a framework for big data processing. These approaches are further categorized as static and adaptive based on the invocation of index creation process and the ability to update indexes. More explicitly, static indexes are created at data upload time and they do not allow index updating once created. By contrast, adaptive indexes are the result of query execution with the flexibility to create as many indexes as index attributes fed by incoming queries.

Clustered static indexes that are developed for big data offer single-attribute indexing, such as the Trojan index (Dittrich et al., 2010), or a varying number of index attributes, such as HAIL (Dittrich et al., 2012). Indexes are created on the entire data set in parallel with data uploading. Thus, the query execution process can be conducted immediately when a query is submitted because it does not invoke index creation or updating. However, selection of attributes to be indexed should be well considered because these are the only indexes that are available throughout the data search process; they cannot be updated later. Based on anticipated query workload knowledge, better indexes are created. Queries that have the same selection predicate can be executed using static indexes; otherwise, full scan can be performed. In the Trojan index, only one particular attribute is selected for indexing, whereas HAIL can extend the number of indexes up to the available number of replicas. We elaborate this concept in Equations 2-1 and 2-2 for Trojan index and HAIL, respectively.

$$No. of\ Indexes = 1 \qquad \textbf{2-1}$$

$$No. of\ Indexes = \ No. of\ Replicas \qquad \textbf{2-2}$$

In contrast to static indexes, adaptive indexes do not offer pre-created indexes to serve incoming new queries. These indexes continue updating with new queries and are being used by repeated queries. Data blocks are replicated for each new index attribute. Lazy indexing (LIAH) is proposed by Richter et al. (2012) as adaptive indexing using clustered approach. LIAH uses an offer rate to minimize indexing I/O cost and creates as many indexes as suggested by incoming queries. However, future utilization of these indexes remains unpredictable. Similarly, from the offer rate perspective, a better tradeoff exists to minimize index creation overhead when the offer rate value is set to low. Nevertheless, to completely index all data blocks, a low offer rate requires a larger number of MapReduce jobs.

Thus, LIAH must compromise either the indexing overhead or the number of MapReduce jobs, thereby motivating a dynamically adapting offer rate (Richter, Quiané-Ruiz, Schuh, & Dittrich, 2014). Query workload prediction is not required and, unlike static indexing, no replication factor dependency is used to consider the number of index attributes in both of these approaches. However, performing a full scan for each new query and replicating the data block for each new index attribute are the performance bottlenecks of LIAH. Therefore, the proposal by Schuh and Dittrich (2015) is to drop the indexes from existing replicas and use these replicas to create new indexes based on the changing query workload.

2.3.2 Non-clustered Indexing on Big Data

The non-clustered indexing category encompasses all indexing techniques discussed in Section 2.2. Among these techniques, any indexing technique that is suitable and efficient for the required indexing mechanism can be chosen to design an indexing solution for big data. To date, Apache Lucene (Białecki, Muir, & Ingersoll, 2012) is a great achievement in full-text indexing and searching big data with high performance. Apache Lucene is an open-source Java-based library that was initially introduced by Gospodnetic and Hatcher (2005) to create indexes for big data using the inverted index, which is a non-clustered indexing structure. Indexes are created using mapping of attributes in a document along with their location, and a pluggable mechanism is later applied to code and store indexes.

Indexes created with the Lucene library are capable of being incrementally updated based on a user-provided list of index attributes. Thus, static indexes and adaptive indexes can be created any time using the Lucene indexing library. Lucene indexing is implemented on Twitter data to create a breaking news detection system (Phuvipadawat & Murata, 2010) and to develop a social web search engine (Bouadjenek, Hacid, & Bouzeghoub, 2013). Implementing a non-clustered approach of indexing, Apache Lucene allows as many indexes for data as the number of attributes in a data set. Lucene indexes are very fast in query execution and take only a few seconds in processing (Kelley et al., 2015). However, longer index creation time and separate index structure are the time and space overhead of Lucene indexing.

2.3.3 Analysis of Indexing Techniques Implemented on Big Data

In this section, we analyze clustered and non-clustered indexing implementations on big data. We summarize static and adaptive clustered indexing approaches in Table 2-4. Their method, success points, and weaknesses are detailed in this table. Furthermore, index hit ratio (defined in Chapter 3), which is a significant efficiency measure for indexing, is also described for each approach.

Table 2-4: Analysis of Clustered Indexing Approaches for Big Data

Approach		**Method**	**Achievements**	**Problems/Un-addressed**	**Index Hit Ratio**
Static	Trojan Index (Dittrich et al., 2010)	One particular attribute is indexed and stored on all replicas	• Index is created at data uploading time, no indexing cost at each query • Full scan option is still valid for queries on non-indexed attributes • Same or improved query execution performance as shared-nothing databases	• One particular index is not sufficient • Indexing upfront cost is higher than running a full scan query • Index Miss ratio is very high • Index may be unused, increasing indexing overhead • Anticipated query workload knowledge is required before index creation • No mechanism for changing query workload	• Only one attribute is indexed that is why all queries having selection predicates other than index attributes are missed
	Aggressive (Dittrich et al., 2012)	Change physical data layout on each replica based on index attributes	• Reduced Index Miss Ratio up to number of replicas • Upload cost is negligible by utilizing un-used CPU cycles • Full scan option is still valid for queries on non-indexed attributes	• High index upfront cost • No knowledge about query workload • Index Miss Ratio is still high • Indexes are replica dependent • Indexes may be unused by queries	• In order to improve Index Hit Ratio, more number of replicas are required
Adaptive	Lazy Indexing (LIAH) (Richter et al., 2012)	Indexing is the effect of query execution. Records in data block are reordered during scan and pseudo data block is created if required.	• Adaptive to query workload • Query can be executed right after data upload • No Indexing upfront cost • Reduced indexing overhead because of selective block indexing • No additional I/O cost • Quick convergence to complete index	• Every first time query faces full scan • Each new index replicates the data block and increases space consumption • Data block replicas are continuously growing with index creation process • Not all data blocks are indexed during one time query execution	• Every first time query faces full scan (index hit ratio is NULL) • In order to improve Index Hit Ratio more number of block replicas are required

Approach		Method	Achievements	Problems/Un-addressed	Index Hit Ratio
				• Constant offer rate either supports indexing overhead or number of MapReduce jobs to completely index all data blocks	
	Adaptive indexing - replace indexes (Schuh & Dittrich, 2015)	Adaptively create and delete un-used indexes	• Query may not result in index creation and help in dropping index • Number of continuously growing index replicas is reduced	• Physical restructuring for each index is required to replace index • Data blocks are still replicated for new index and consume disk space	• Index Hit Ratio is same as Lazy Indexing Approach
Hybrid	Eager Adaptive Indexing (Richter et al., 2014)	Introduce cost model for LIAH with varying offer rate. Missing indexes of HAIL are created adaptively	• Static HAIL adapts to new query workload • Indexing cost is not over burdened • Adaptive indexing overhead is less than full scan • Quick convergence to complete index	• Data block replicas are continuously growing with index creation process	• Index Hit Ratio is improved from HAIL as new indexes are created runtime

Table 2-4 shows that clustered indexing approaches, whether static or adaptive, allow one index per replica and replicating the extensive volume of data for more indexes does not seem practical. Similarly, a better predictor to future query workload is lacking in to-date clustered indexing advancements. Predicting future query workload may assist in deciding attributes to be indexed so that the costly procedure of later index updating and frequent data re-ordering can be avoided.

When comparing clustered and non-clustered indexing approaches, we observe that although clustered indexing implementation on big data has high performance gains, index management and updating is not as straightforward as it should be. Changing query workload and requiring more indexes during the search process is natural. However, replicating voluminous data to create indexes is not a practical approach, which is the only option with clustered indexing.

Similarly, non-clustered indexes have their own limitations. Taking extra time to create indexes and separate storage space comprise the overhead that is associated with non-clustered indexing. However, many indexing techniques are available in the literature (Section 2.1), thereby reducing the indexing overhead along with improving query execution and search performance. The Apache Lucene indexing library, for example, uses the non-clustered indexing approach and results in faster query execution time with smaller index size and least index creation time.

We are able to identify current challenges in big data indexing from Table 2-3 by presenting the comparison of clustered and non-clustered indexing and in Table 2-4 by highlighting the problems of clustered indexing under static and adaptive methods. These challenges provide insights to develop an optimum indexing solution for big data. The following findings from the preceding review are the milestones to formulate new research objectives toward the development of an improved indexing mechanism. Thus, efficiency in search operations over big data can be achieved with reduced index storage consumption and index creation time

- Indexing is a significant process to improve data search and query execution performance for relatively large and growing data sets.
- Clustered indexes are proven to result in less indexing overhead, whereas non-clustered indexes require separate index storage space and indexing time. However, index updating and adding new indexes using the clustered approach is less convenient than non-clustered approach of indexing.
- Overhead from indexing process is one time and becomes negligible when clear improvement in search performance is obtained.
- Overall indexing overhead is somehow inversely proportional to the size of the data set but is directly proportional to the number of index attributes.
- The more the number of attributes considered in indexing, the greater the overhead is, although the index hit ratio increases.
- A wise selection of attributes for indexing provides a better tradeoff between indexing overhead and hit ratio.
- Adaptive to changing query workload index updating also supports our prior claim.

2.4 Conclusion

This chapter reviews recent indexing advancements in the field of big data and emphasizes their potential problems. We define indexing requirements for big data and use these requirements as criteria to investigate the adequacy of contemporary indexing techniques for big data. We also present a review of indexing mechanisms that are implemented on big data to analyze their efficiency.

The investigation of recent indexing techniques by using big data indexing requirements shows that each category, i.e., NAI, AI, and CAI has distinct adaptability. The NAI indexing techniques support the volume of big data. As we have presented in Table 2-2, all NAI indexing techniques except lazy indexing have support to volume. The performance of NAI indexing is also justifiable for velocity and variety of big data. However, the AI indexing category signifies robustness and accuracy of results, and CAI indexing fulfils the volume requirements of big data as well. The spectrum of NAI indexing to support big data indexing requirements is wider than the AI and CAI indexing categories.

The investigation of indexing implementation on big data shows that both clustered and non-clustered approaches result in a significant improvement in search and data retrieval performance for big data. However, both have their own design constraints. The clustered approach results in minimum indexing overhead because indexes are not separate structures. By contrast, adding new indexes needs data to be replicated, thereby indicating that the number of indexes for a data set is subject to availability of storage space to create the same number of replicas. The non-clustered approach of indexing creates separate index structures and requires additional storage. However, index storage consumption caused by the non-clustered approach remains less than the replicating data for adding new indexes.

We found that indexing techniques under the NAI, AI, and CAI categories are non-clustered approaches from which the NAI indexing techniques are more inclined to fulfil big data indexing requirements. We also found that the non-clustered approaches offer increased flexibility to create and update indexes regardless of constrained storage for big data. Therefore, the non-clustered approach, specifically NAI indexing, has been proven more effective than other methods for big data analysis environment.

CHAPTER 3: PERFORMANCE ANALYSIS OF INDEXING TECHNIQUES FOR BIG DATA [5]

This chapter aims to establish the research problem by examining the performance and identifying the limitations of indexing mechanisms when implemented for big data. We have elaborated in Chapter 2 (Table 2-2 and Table 2-3) that the non-clustered approach of indexing has more options to create indexes with minimized indexing overhead and improved search performance. Thus, in this chapter, we implement non-clustered indexing to explore its limitations and deficiencies. We obtain results from experiments and present the overhead caused by performing indexing in terms of indexing time and index size. Moreover, we examine the search and data retrieval time for MapReduce jobs and indexed jobs.

This chapter consists of three sections. Section 3.1 presents the experimental setup and data collection method used in problem analysis. Section 3.2 presents the results and discusses the reported results. Section 3.3 concludes the chapter.

3.1 Experimental Setup and Data Collection

In this section, we describe the experimental setup and data design. We investigate the performance of indexing implementation on big data. The experimental setup to conduct analysis consists of hardware and software specifications of used devices, whereas data design includes performance metrics, description of data set, and data collection process.

We implement the indexing mechanism on the big data processing framework and analyze the effect of indexing on data retrieval performance. We verify the query

[5] The work presented in this chapter is partially obtained from the following research contribution:

Siddiqa, Aisha, Karim, Ahmad, Gani, Abdullah, & Chang, Victor. On the analysis of big data indexing execution strategies (2016). *Journal of Intelligent and Fuzzy Systems* processing latencies that are observed in current big data processing systems and identify the overhead and inefficiency of indexing structures for big data.

3.1.1 The Model

We present the experimental model that we have used for performance evaluation. We select Hadoop, a big data processing framework that comprises the MapReduce programming model, to execute a job in a distributed parallel manner that supports a tremendous amount of big data. We use the distributed file system incorporated with Hadoop (HDFS) for storing and managing data sets in the form of files.

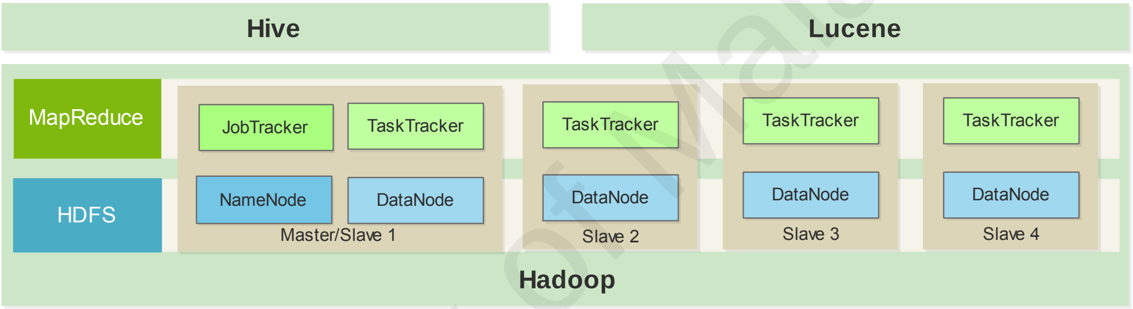


Figure 3-1: Experimental Model

We demonstrate our experimental model in Figure 3-1. In this analysis, we establish a test bed that comprises four physical commodity servers and configure the Hadoop four-node cluster. The master–slave cluster comprises four slave nodes; one of these slaves also acts as a master. We configure the respective MapReduce and HDFS daemons that are required for data processing and storage.

As shown in Figure 3-1, we use Hive and Lucene to execute queries on full scan and indexed search environments, respectively. We configure Hive warehouse over Hadoop, which offers full scan execution of SQL-like queries on big data using Hive Query Language (HQL). We use the Apache Lucene indexing library to execute queries with indexes. The two query execution environments of our experiment, i.e., full scan and indexed search, are explained as follows:

- Full Scan

The full scan environment for query execution over the Hadoop framework is the process of executing the MapReduce job on slave nodes to perform search and retrieval operations on big data. We use the Hive warehouse to execute queries in the full scan environment.

MapReduce divides a job into small tasks that use map, combine, and reduce functions. Each map function generates the results in the form of <key, value> pairs for the records that match the query. The combine function merges the records based on query requirements. Finally, the reduce function uses the output generated by map and combine functions across all TaskTrackers and concatenates the results. The output of the reduce task is the overall output that contains the retrieved data for a query.

- Indexed Search

The indexed search environment uses indexes to perform query search operation. We use the Apache Lucene library, which is highly efficient in creating indexes and performing data retrieval operations on big data using indexes. Apache Lucene utilizes inverted indexing that is a non-clustered NAI indexing approach. We use index creation and query execution program codes written in Java when importing the library.

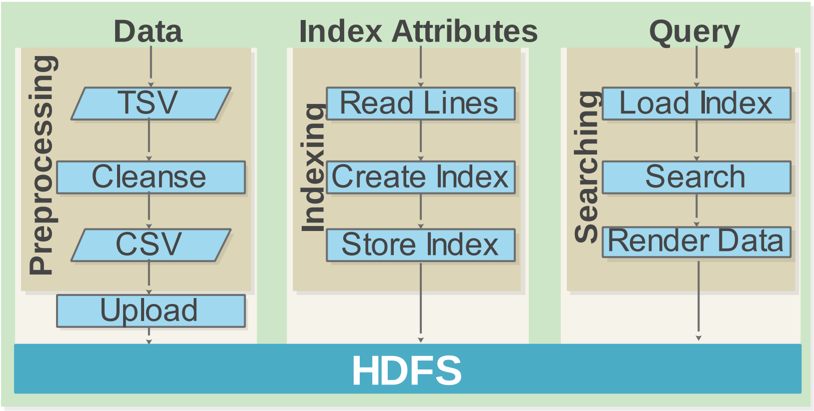


Figure 3-2: Indexed Search Query Execution

The process of indexed search using Lucene is elaborated in Figure 3-2. We apply necessary pre-processing on data before storing the data in HDFS. For instance, attributes or columns in records may have different separators such as space, comma, tab, or others. During pre-processing, we convert the separators into commas (CSV).

We perform index creation on data that we have uploaded in HDFS as data files. The process begins by creating an object in the memory where indexes are created. The data file is then read from HDFS and the indexes are stored as index files. We store these index files in HDFS for further utilization.

We perform indexed search for query execution by loading index files in a memory object from HDFS. Search operations to obtain required data are performed based on the selection predicate that is specified in query. In Lucene indexes, index files store the pointers only for the records for which the indexes are created. Therefore, data can be retrieved only for those data attributes that are included in index files. Thus, all attributes are required to be indexed to retrieve the entire record from the file.

3.1.2 Performance Measures

This subsection presents the metrics that we have used to analyze the performance of full scan and indexed search operations. Index size, indexing time, and search performance are the conventional metrics used in research to analyze indexing. However, index hit ratio is also useful for big data indexing when creating indexes on all attributes is not feasible. The description of performance metrics is provided as follows:

- *Index Size (MB):* Index size is the space that is required by the index in memory and/or on storage. Many factors affect index size, such as the number of non-empty values for an index attribute and the size of these values. Index size is an overhead on actual size of big data sets and therefore, index size is recommended to be very small.

- *Indexing Time (sec):* Indexing activity takes time when applied to big data sets. However, users appreciate minimum delays. This delay is calculated as indexing time, known as index creation time. Similar to a small index size, short indexing time is appreciated.
- *Search Time (sec):* Search time measures the time required to execute a query and retrieve data. The search space that is offered by indexing is less and more structured than the actual data search space, which is used by full scan. Therefore, a minimized time to execute queries and retrieve data is achieved.
- *Index Hit Ratio:* We introduce index hit ratio to examine the rate of incoming queries that are served by indexes. Index hit ratio is the ratio of queries that hit the index during execution. In attribute-based indexing, where all attributes of data are not indexed because of increased indexing overhead (i.e., index size and indexing time), the index hit ratio is significant to ensure that maximum incoming queries are executed with these indexes. We calculate the index hit ratio using the following equation:

$$Index\ Hit\ Ratio = \frac{No.of\ Index\ Attributes}{No.of\ Attributes} \quad \textbf{3-1}$$

3.1.3 Data Set Used

We use varying size data sets in our experiment to determine the behavior of index operations on different workloads. For this purpose, we obtain TIGER data sets (Eldawy & Mokbel, 2015), which contain the spatial features of geographical areas. These data sets offer flexibility of observation as they vary in size, No. of records, No. of attributes and No. of created blocks and allow to analyze the impact of varying characteristics of a data set on obtained results. Later, the results have shown that the characteristics other than data set size also influence the performance of indexing and search operations. Table 3-1 summarizes the data sets.

Table 3-1: Data Sets

Data Sets	Data Size	No. of Records	No. of Attributes	No. of Blocks
Primary Roads	77.1	13373	10	2
Area Landmark	406	121960	15	7
Tabulation Area	1,600	33144	15	25
Area Hydrography	6,460	2298808	16	104
All Edges Combined	16,220	19291957	37	260
Linear Hydrography	18,270	5857442	11	293

We leverage six different data sets from the TIGER database to collect results for data sets with varying sizes. Each data set differs in size and number of records. As shown in Table 3-1, data sets have varying numbers of blocks depending on data set size. The primary road data set (77.1MB) is the smallest in our experiment. It comprises 13,373 records and HDFS has stored the primary road data set in two blocks. Table 3-1 summarizes the information of each data set in which linear hydrography (18,270) is the largest with 5,857,442 records and comprises 293 blocks in HDFS.

3.1.4 Data Collection Tools

To obtain accurate results for each parameter, we design data collection tools carefully. We collect data for index size and data set size from the user interface to browse the file system, whereas data for time is collected from the console and stopwatch Apache API.

We upload each data set in Hive warehouse from the local disk by running the HQL command in the console; the time taken to run this command is the data upload time. We perform full scan operation using HQL command in the console and note the time as the full scan search time. The stopwatch Apache API returns the time for indexed search operations (i.e., indexing time and searching time).

3.2 Results and Discussion

In this section, we present and discuss the results of our experiments. We provide the results of indexing overhead in terms of index size and indexing time. We perform search operation on both full scan and indexed environment to observe the effect of indexing for big data. The experiment incurs an out-of-memory error when creating indexes for the all-edges combined data set, thereby showing that the indexing code uses more memory than available at the physical machine during this experiment. Therefore, we are unable to obtain results for this set.

We compare the results of index size with the data set size, indexing time with data uploading time, and indexed search time with full scan search time. Given that we have implemented non-clustered indexing, we compare the index hit ratio of non-clustered indexing with the clustered indexing approach.

An interesting observation from the execution of Apache Lucene indexing is that the success of index creation process depends upon the available main memory size (i.e. RAM). We have shown in Figure 3-2 that data set is loaded into main memory to create indexes. The experiment returns out of memory error whenever heap size cannot accommodate a data set. For instance, Apache Lucene library fails to create indexes for All Edges Combined data set (see Table 3-2 and Table 3-3). Therefore, our performance analysis is limited to six data sets where data set size remains under heap size.

3.2.1 Index Size Results

We present the results of index size and index size overhead. As shown in Table 3-2, two observations are related to index size. First, index size grows with data set size. We have chosen data sets that grow in size. The results show that the index size is also growing. Index size overhead demonstrates the percentage of growth in data set size because of indexing. Second, index size grows with the number of attributes to be considered for indexing (i.e., index attributes). We consider up to five index attributes for each data set to observe the effect of increased number of index attributes.

Table 3-2: Index Size Results

Data Sets	Data Size (MB)	Index Size for varying No. of Index Attributes					Index Size Overhead (%)				
		1	2	3	4	5	1	2	3	4	5
Primary Roads	77.1	0.36	0.67	0.96	1.24	1.53	0.46	0.86	1.23	1.58	1.94
Area Landmark	406	2.39	5.55	6.27	8.11	9.05	0.58	1.34	1.51	1.95	2.17
Tabulation Area	1,600	0.92	1.78	2.60	3.47	4.30	0.05	0.11	0.16	0.22	0.27
Area Hydrography	6,460	28.51	39.92	57.95	82.57	185.17	0.43	0.61	0.88	1.26	2.78
All Edges Combined	16,220	Out of memory Error					-	-	-	-	-
Linear Hydrography	18,270	114.25	136.64	294.78	329.55	369.93	0.62	0.74	1.58	1.77	1.98

The results for the primary road data set (77.1 MB) show that index size are 0.36, 0.67, 0.96, 1.24, and 1.53 MB for one to five indexes, respectively (Table 3-2). Thus, index size overhead for the primary road data set increases from 0.46% to 1.94%. The area landmark data set (406MB) has an initial 2.39MB index size for one index attribute; this size grows with the number of index attributes to 5.55, 6.27, 8.11, and 9.05MB for five index attributes, and the overhead reaches 0.58% to 2.17%. We also present the index size results for up to five attributes for tabulation area (1600MB), area hydrography (6460MB), and linear hydrography (18270MB) in Table 3-2.

Table 3-2 also shows that the index size overhead from the primary road data set (77.1MB) to linear hydrology data set (18,270MB) is almost similar and less than 3%. However, the index size overhead is smallest (~0.3%) for the tabulation area data set (1600MB). The reason is that the tabulation area data set contains the smallest number of records, as presented in Table 3-1. Fewer records in larger size data set is also significant in index size; thus, the index size overhead becomes very low. Collectively, index size overhead results show that the memory-based indexes created by Lucene are very small and that the size overhead does not reach more than 3%.

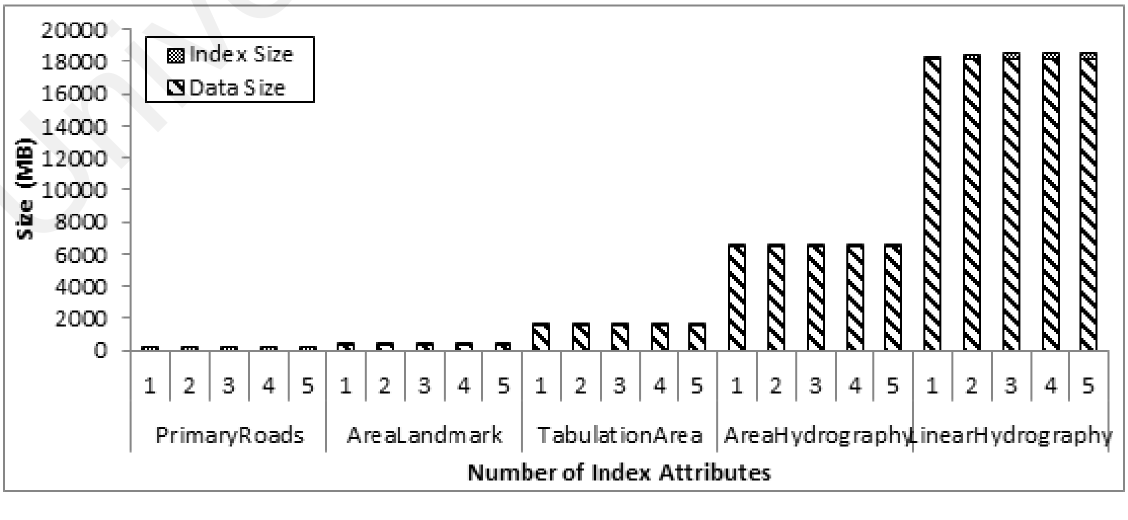


Figure 3-3: Index Size comparison with Data Set size and Number of Index Attributes

We also present index size results in plotted form for better elaboration for an increasing number of index attributes in Figure 3-3. The bars show that index size is considerably less than the data set size; for area landmark and tabulation area data sets, the index size is almost invisible. Figure 3-3 also shows that the index size for one or two index attributes is almost invisible and slightly grows when the number of index attributes is increased.

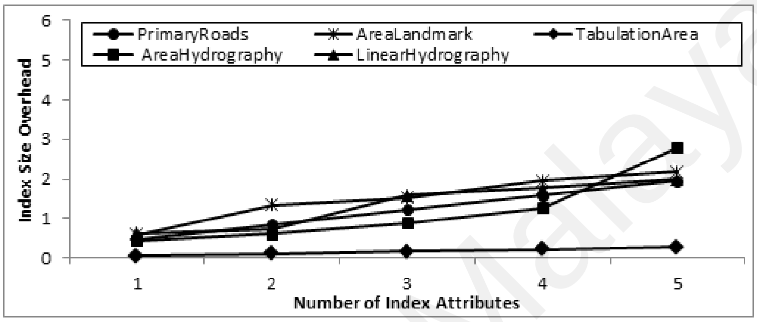


Figure 3-4: Index Size Overhead for varying number of Index Attributes

We present the percentage growth of index size overhead in Figure 3-4. The least index size overhead is for the tabulation area data set and is almost parallel to x axis. The overhead for other data sets is also very low and the effect of adding more attributes in index creation is not very high. Figure 3-4 also shows that for each data set, although the overhead increases with the number of index attributes, it is still less than 3% of the actual data set size for five index attributes.

The index size increases with data set size. The results show that indexing activity increases the storage space requirements for data sets. The index size mainly depends on the size of the data set. The results have proven that for large data sets, the index size is also large.

Another important factor that affects index size is the number of records in a data set. A data set may be very large while having a certain number of records, i.e., the tabulation area data set (1600MB) comprises 33,144 records. By contrast, the area landmark data set (406MB), which is smaller than the tabulation area data set, contains 121,960 records. Given the small number of records in the tabulation area data set, the index size is always observed as less than the area landmark data set (Table 3-2).

The content size and number of occurrences of an index attribute also contribute to the index size. The results for area hydrography data set (6460MB) show a slight increase in index size when created for the first four index attributes. The index size grows almost linearly for up to four index attributes, i.e., 28.51, 39.92, 57.95, and 82.57MB. However, when a fifth index attribute is added, an abnormal increase in index size occurs, i.e., 185.17MB. The increase in index size with the fifth index attribute shows that only increasing the number of index attributes is not the reason. The reason is that either the content size of the fifth index attribute is higher than that of the first four index attributes or that the number of occurrences is very high.

The factors that affect the index size for a data set are not completely under the control of an indexing mechanism. The index size mostly depends on the nature of the data set. However, an efficient indexing mechanism that offers a structure with reduced index sizes even when more index attributes are considered, is preferable.

3.2.2 Indexing Time Results

We present the results of indexing time and overhead in this section. Indexing activity increases the delay to start query execution. We indicate this delay as indexing overhead.

Table 3-3: Indexing Time Results

Data Sets	Data Upload Time	Indexing Time for varying No. of Index Attributes					Indexing Time Overhead (%)				
		1	2	3	4	5	1	2	3	4	5
Primary Roads (77.1 MB)	7.71	10.00	10.27	10.39	10.44	10.68	56.47	57.11	57.40	57.53	58.07
Area Landmark (406 MB)	39.28	405.04	409.94	410.55	414.91	415.29	91.16	91.25	91.27	91.35	91.36
Tabulation Area (1,600 MB)	151.88	152.79	153.21	154.01	154.21	154.57	50.15	50.22	50.35	50.38	50.44
Area Hydrography (6,460 MB)	703.41	720.30	726.58	732.99	733.70	739.90	50.59	50.81	51.03	51.05	51.26
All Edges Combined (16,220 MB)	1773.66	Out of memory Error					-	-	-	-	-
Linear Hydrography (18,270 MB)	1984.72	1695.56	1696.85	1757.67	1759.03	1764.55	46.07	46.09	46.97	46.99	47.06

Table 3-3 shows the indexing time and indexing overhead results. Indexing on the primary road data set (77.1 MB) takes 10, 10.27, 10.39, 10.44, and 10.68 s for one to five index attributes, respectively. Thus, indexing time overhead for the primary road data set increases from 56.47% to 58.07%. Area landmark data set (406MB) initially takes 405.04 s for one index attribute that respectively grows with the number of index attributes to 409.94, 410.55, 414.91, and 415.29 s for five index attributes; the overhead reaches 91.16% to 91.36%. Similarly, the indexing time for up to five attributes for tabulation area (1600MB), area hydrography (6460MB), and linear hydrography (18270MB) are also shown in Table 3-3.

Table 3-3 also shows that indexing time overhead for all data sets is almost similar, which ranges from 45%–60%, except for the area landmark data set (406 MB). Indexing time overhead for the area landmark data set with varying number of index attributes is very high (~92%). The reason is that the area landmark data set contains a large number of records (i.e., 121,960), as shown in Table 3-1. Thus, creating indexes using the Lucene library results in very long delays (up to 60%), which may increase when a large number of records are present in a data set.

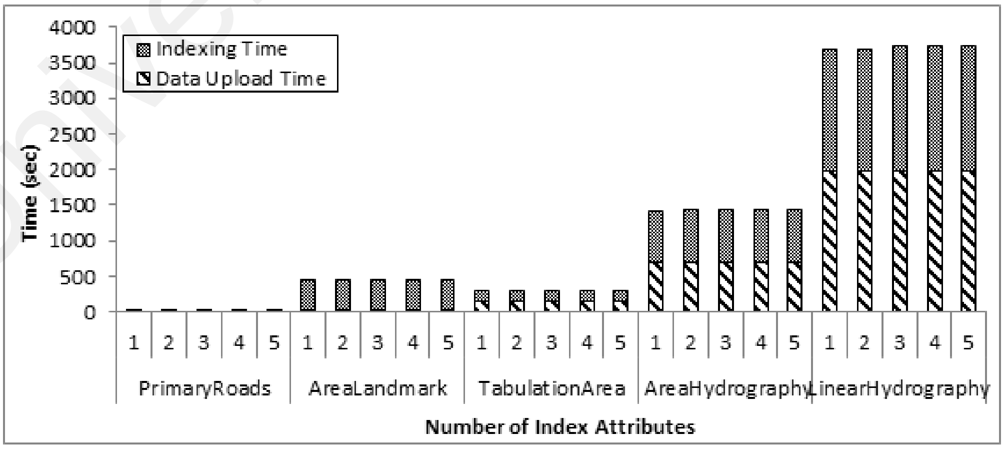


Figure 3-5: Indexing Time comparison with Data Upload Time and Number of Index Attributes

We visualize the indexing time results in a bar chart. Figure 3-5 presents the results of indexing time with respect to data uploading time and number of index attributes. The bars show that index time is slightly higher than data upload time for primary roads, tabulation area, and area hydrography data sets. By contrast, for the linear hydrography data set, indexing time is slightly less than data upload time. However, for all these data sets, indexing takes almost the same time as data upload time. Indexing time is only high for the area landmark data set. Figure 3-5 also shows that indexing time slightly increases as the number of index attributes for each data set increases.

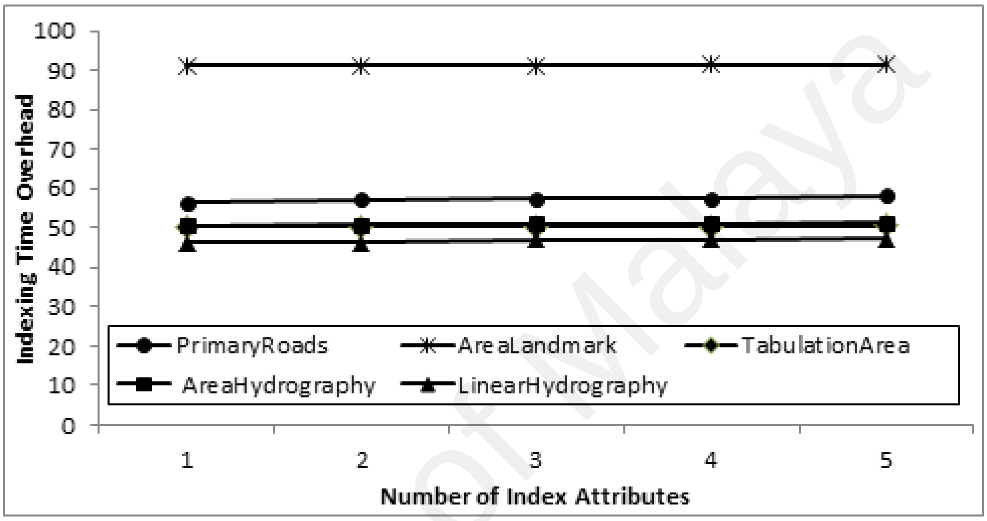


Figure 3-6: Indexing Time Overhead for varying number of Index Attributes

Figure 3-6 presents the effect of increasing number of index attributes on indexing time overhead where the indexing overhead for the area landmark data set is highest. The indexing overhead for other data sets ranges from 40%–60%. The lines that indicate indexing time overhead for each data set are almost linear, thereby indicating that the increasing number of index attributes does not have a significant effect on indexing time overhead. Therefore, this analysis indicates that considering more index attributes has a linear increase in indexing time overhead.

Two observations that relate to indexing time are found. First, indexing takes more time for larger data sets. To prove this observation, we chose data sets with different sizes; the indexing overhead shows that the percentage increased the delay with a growing volume of data sets. Second, indexing time increases with the number of index attributes.

However, the rate of increase in indexing time caused by the number of index attributes is very low.

Indexing time results also show that indexing activity increases the delay between uploading data and starting query execution. Creating indexes using the Lucene library needs almost same time as does uploading data. Thus, the delay is two times higher than the full scan to perform indexed search operation.

Another significant observation from Figure 3-6 is that the number of records in a data set also affects indexing time. Indexing time overhead is less for a data set that has fewer records even when the size of the data set is larger (i.e., a tabulation area data set of 1600MB that comprises 33,144 records) than indexing time overhead caused by area landmark data set (Table 3-3). The indexing time overhead increases because of a higher number of records in the area landmark data set. The reason is that the indexing process reads records individually in the index creation process (Figure 3-2).

The discussed factors affect the indexing time. A larger data set size, a higher number of index attributes and a higher number of records in a data set increase the indexing time. Therefore, an indexing mechanism is needed that considers these factors and offers minimized time consumption in the index creation process.

3.2.3 Search Time Results

In this section, we present the results of search time that were taken by full scan and indexed search. We show the query execution time results obtained from full scan and indexed search and compare the improved search performance of indexed search.

Table 3-4: Search Time Results

Data Sets	Search Time		Search Performance (%)
	Full Scan	Indexed	
Primary Roads (77.1 MB)	9.60	1.29	86.57

Area Landmark (406 MB)	31.85	1.40	95.60
Tabulation Area (1,600 MB)	21.84	2.54	88.37
Area Hydrography (6,460 MB)	63.69	2.97	95.34
All Edges Combined (16,220 MB)	-	No index	-
Linear Hydrography (18,270 MB)	183.00	2.98	98.37

Table 3-4 presents the results of query execution and search performance. For the primary road data set (77.1 MB), full scan takes 9.60 s in searching, whereas indexed search time is decreased to 1.29 s. Thus, search performance improvement with indexing is 86.57% for the primary road data set. Similarly, search time results with full scan are 31.85, 21.84, 63.69, and 183.00 s for other data sets such as area landmark (406 MB), tabulation area (1600 MB), area hydrography (6460 MB), and linear hydrography (18270 MB), respectively. Search time with indexed search results are 1.40, 2.54, 2.97, s and 2.98s, respectively, for these data sets.

Table 3-4 also shows that search time is significantly reduced for each data set when indexed search is applied and search performance is improved to 95.60%, 88.37%, 95.34%, and 98.37%. Search performance improvement is more than 86% with indexed search; it gradually increases with larger size data sets except for tabulation area data set (1600 MB), which is 88.37%. Although performance is high, it is not as much as that obtained with other data sets. The reason is that the tabulation area data set takes significantly less search time with full scan because of fewer records, whereas indexed search takes a normal amount of time. Thus, overall search performance has not increased. The search time is significantly reduced with Lucene indexes and search performance increases with larger size data sets such as indexed search takes 2.98s for Linear Hydrography data set that is largest data set in our experiment.

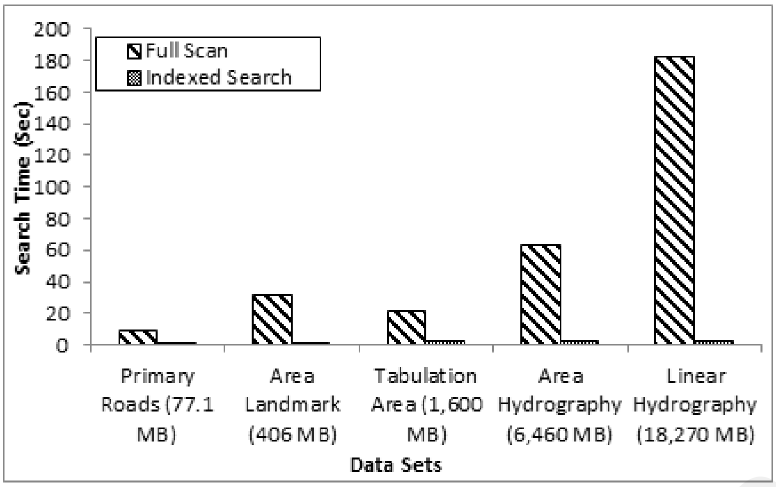


Figure 3-7: Search Time Comparison between Full Scan and Indexed Search

Figure 3-7 shows the search time results in both full scan and indexed search cases for all data sets. Full scan bars show that search time increases with data set size. However, full scan search time is 21.84 s for tabulation area data set (1600 MB), which is less than the full scan search time taken by a smaller size data set, i.e., area landmark (406 MB) because of the small number of records in tabulation area data set. Indexed search bars are very small in Figure 3-7, thereby showing that search time with indexed search is very low for all data sets. The size of data set has a very small effect on indexed search time.

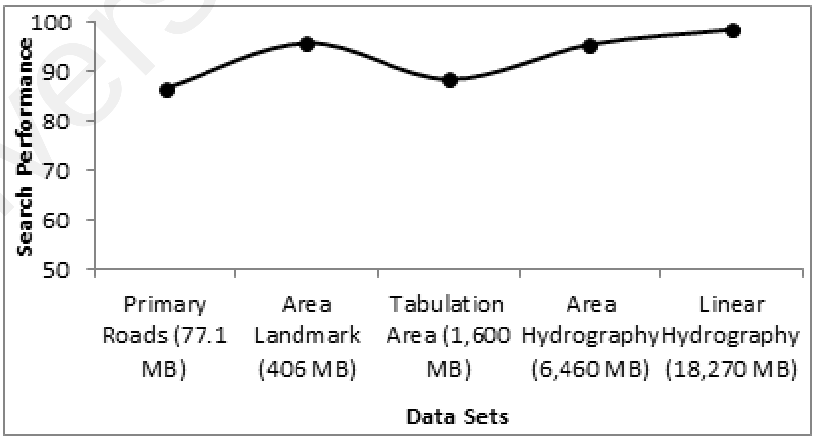


Figure 3-8: Improved Search Performance with Index Search

Figure 3-8 shows that search performance improvement with indexed search is more than 86% of full scan. This performance increases with data set size. The primary road data set (77.1 MB) is smallest in our experiment, for which the search performance gains are 86.57%. Search performance is respectively improved to 95.60%, 88.37%, 95.34%, and 98.37% with growth of data set size, i.e., area landmark (406 MB), tabulation area (1600 MB), area hydrography (6460 MB), and linear hydrography (18270 MB).

One interesting observation from query execution and search performance is that search time increases with data set size in both full scan and indexed search cases. However, the increase in indexed search time is less than the full scan search time. We chose data sets with varying sizes to prove the indexed search performance improvement.

The second observation is related to performing queries to retrieve data for non-indexed attributes. Lucene performs search for only indexed attributes; Lucene is unable to retrieve data for non-indexed attributes. Therefore, we executed queries that have indexed attributes as both selection predicates and search predicates. When entire records are retrieved using Lucene indexing, all attributes must be indexed.

Search time results show that indexing activity minimizes query execution time, thereby improving overall search performance. Performing indexed search via Lucene indexing takes significantly less time than does full scan search. A large data set that comprises a higher number of blocks needs more map and reduce jobs to perform data search and retrieval operation. However, indexing reduces the search space, thereby resulting in very quick responses. The results have proved that the search time is more than 86% improved with indexed search.

The results also show that performance improvement is more than indexing overhead, which is a positive aspect of Lucene indexing. Indexing overhead results show that indexing increases roughly 3% size of a data set and takes 40%–60% more time than data uploading. However, the search performance improvement of indexing is more than 86%, thereby proving that usefulness of indexing is more than the cost of indexing.

3.2.4 Index Hit ratio

In this section, we present the results of index hit ratio for varying numbers of index attributes. We show that regardless of the data set size, considering a higher number of attributes in index creation increases the index hit ratio.

Table 3-5: Index Hit Ratio Results

Data Sets	**Data Size (MB)**	**No. of data Attributes**	**Index Hit Ratio for No. of Index Attributes**				
			1	**2**	**3**	**4**	**5**
Primary Roads	77.1	10	0.1	0.2	0.3	0.4	0.5
Area Landmark	406	15	0.07	0.13	0.20	0.27	0.33
Tabulation Area	1,600	15	0.07	0.13	0.20	0.27	0.33
Area Hydrography	6,460	16	0.06	0.13	0.19	0.25	0.31
Linear Hydrography	18,270	11	0.09	0.18	0.27	0.36	0.45

Table 3-5 shows the results of index hit ratio for all data sets. The primary road data set (77.1 MB) comprises 10 data attributes and the index hit ratio for up to five index attributes is 0.1, 0.2, 0.3, 0.4, and 0.5, respectively. The index hit ratios for both area landmark (406 MB) and tabulation area (1600 MB) data sets is the same for the five attributes, which are 0.07, 0.13, 0.20, 0.27, and 0.33, respectively. The reason is that both data sets have an equal number of data attributes. The index hit ratio for area hydrography (6460 MB) and linear hydrography (18270 MB) are also shown in Table 3-5 with respect to an increase in the number of indexed attributes.

The index hit ratio for each data set linearly increases with the number of index attributes. The index hit ratios for primary roads and linear hydrography data sets are higher than those of the other data sets. Meanwhile, the index hit ratios are the same for both area landmark and tabulation area data sets, whereas area hydrography has a slightly higher index hit ratio. The index hit ratio only considers the number of index attributes and total number of attributes in a data set. Therefore, the ratio increases with a higher number of index attributes and decreases with a higher number of total attributes in a data set.

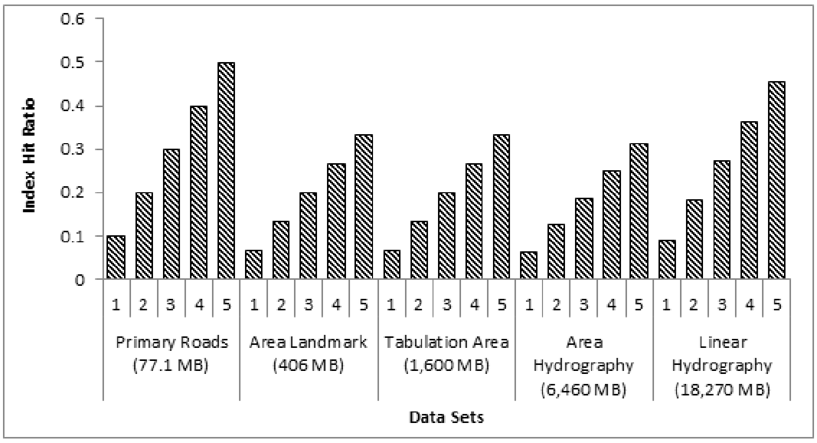


Figure 3-9: Index Hit ratio Comparison with varying No. of Index Attributes

We visualize the index hit ratio for all data sets with varying number of index attributes in Figure 3-9. The index hit ratio of the primary roads data set for each index attribute is the highest among all data sets. The bars for the area landmark and tabulation area data sets show equal index hit ratios at each number of index attributes, whereas this ratio is slightly low for the area hydrography data set. Similarly, the index hit ratio of linear hydrography at each number of index attribute is slightly lower than the primary roads data set.

Index hit ratio depends on the number of attributes in a data set instead of size or other features of a data set. We used a fixed number of index attributes for each data set in our evaluation. The number of index attributes for a data set can be increased to achieve higher index hit ratios. However, an increasing number of index attributes has an obvious effect on index size and indexing time. Therefore, the selection of attributes to be considered in indexing is crucial.

3.3 Conclusion

In this chapter, we investigate the effect of clustered indexing for big data in terms of indexing overhead and search time performance. We found that indexing improves search performance when increasing overhead on data size and data uploading time.

The results of investigation proved that indexes created using the Lucene library improve data search performance by at least 86% and that the performance is improved for larger data sets. However, adding an attribute to be indexed increases overall index size and index size overhead on the data set size. Moreover, increasing the number of index attributes also increases indexing time as well as indexing time overhead on data upload time. Furthermore, the results indicate that indexing time depends on the number of records in a data set and that indexing time overhead is very high (i.e., 40–60%) compared with index size overhead (i.e., ~3%). Therefore, indexing time reduction can make the indexing process more appealing for big data users because the delay to start query execution is minimized.

We also identified that creating more indexes increases the index hit ratio if incoming query workload is supposed to be equal for all attributes. The results have proved that adding to index attributes linearly increases the index hit ratio. However, assuming that the query workload is equal for all attributes is unrealistic. Query workload may be irregular and may vary from time to time. Thus, creating indexes only before starting query execution is not practical. Therefore, an adaptive to query workload indexing along with one-time static indexing can increase the index hit ratio.

CHAPTER 4: SMALLCLIENT FOR BIG DATA: PROPOSED INDEXING FRAMEWORK [6]

This chapter presents the proposed indexing framework for big data. The framework aims to attain minimized indexing overhead in terms of index creation/updating time and index size, reduced data retrieval time with faster query execution, and maximum index hit ratio by predicting the future workload of incoming search queries. Our indexing framework, named SmallClient, uses a non-clustered NAI indexing approach to perform efficient indexing and query execution process on big data sets. The implemented indexing approach is efficient in reducing indexing overhead with an increase in the number of index attributes. Furthermore, SmallClient offers predictor logic to adaptively update indexes, thereby achieving an improved index hit ratio.

This chapter consists of the following six sections: Section 4.1 introduces the proposed indexing framework and Section 4.2 presents the system architecture and explains the process flow of how SmallClient interacts with the user and the underlying file system. Section 4.3 describes each module of SmallClient. Section 4.4 presents the mathematical model for the proposed framework. Section 4.5 highlights its prominent features and Section 4.6 concludes the chapter.

4.1 SmallClient Indexing Framework

We propose SmallClient for indexing big data. SmallClient is a generalized compact framework to address data search and query execution operations over contemporary

[6] The work presented in this chapter is partially available in following research papers:

Siddiqa, A., Karim, A., & Chang, V. (2016). SmallClient for big data: an indexing framework towards fast data retrieval. *Cluster Computing*, 1-16. doi:10.1007/s10586-016-0712-4

Siddiqa, A., Karim, A., & Chang, V. Modelling SmallClient indexing framework for big data analytics. *Supercomputing* (Under Review) distributed file systems such as HDFS. We adopt B-Tree, which is a non-clustered NAI indexing approach to create separate, easily manageable, and updatable index structure. SmallClient deals with the problems of both clustered and non-clustered approaches that are identified in Chapter 3. SmallClient offers maximum possible indexes for big data sets. SmallClient also enables creating indexes for each data block instead of the entire data set, thereby making the indexes manageable as data volume grows.

SmallClient solves the problems of contemporary clustered indexing approaches for big data. As identified in Chapter 3, the number of indexes that use the clustered indexing approach is constrained to the number of replicas. However, available storage capacity may not allow creating many replicas for big data sets. Thus, only a limited number of indexes can be created. As a result, all incoming queries may not use available clustered indexes. In addition, the overall index hit ratio is very low. The clustered approach performs physical data reordering to update the index list, thereby resulting in high computational cost. SmallClient implements the non-clustered indexing approach in which indexes are separately manageable structures. Separate indexes that are created via SmallClient are independent from the number of replicas for a data set. Therefore, when using SmallClient, increasing the number of attributes for indexing has no limit.

SmallClient considers the limitations of recently deployed non-clustered indexing approaches. The problem with existing implementation of non-clustered indexing on big data is that indexing time overhead and index size overhead are very high. SmallClient uses specialized procedures to reduce the indexing overhead. Existing approaches use the data set as a whole and incur out-of-memory errors, whereas SmallClient offers index creation for separate data blocks. Thus, memory utilization for index creation is very low.

The proposed big data indexing framework comprises procedures that are related to creating static indexes based on user-specified lists of attributes, offering a query execution platform, and adaptively updating pre-created indexes with the passage of query execution. In addition to indexing, SmallClient offers a block creation procedure that creates blocks for data sets to avoid tuple rupturing during the filling of a block container. Consequently, novel procedures to perform data retrieval operations enable SmallClient to outperform existing big data indexing mechanisms.

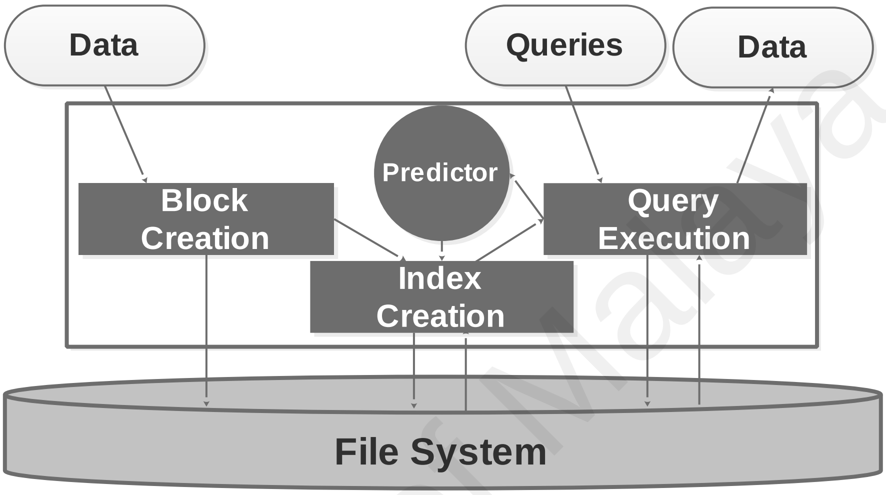


Figure 4-1: Proposed indexing framework, SmallClient

Figure 4-1 presents the framework of our proposed indexing client, SmallClient. We decompose our client into three modules: the first is designed to create data blocks that split data into smaller manageable chunks and uploads these series of chunks as data blocks to a file system. We present index creation design as a second module that uses the B-Tree NAI technique of non-clustered indexing. The predictor function is used to update indexes adaptively based on query workload. The third module offers query execution procedure to retrieve required data and shows improved search performance for large data sets.

4.2 The Architecture

In this section, we present the system architecture and process of a distributed file system in which SmallClient facilitates data retrieval operation as an intermediate layer between user interface and file system. In this section, we describe the process of each layer and present the interaction of SmallClient with other layers. The systematic layered visualization of data retrieval system is depicted as Figure 4-2.

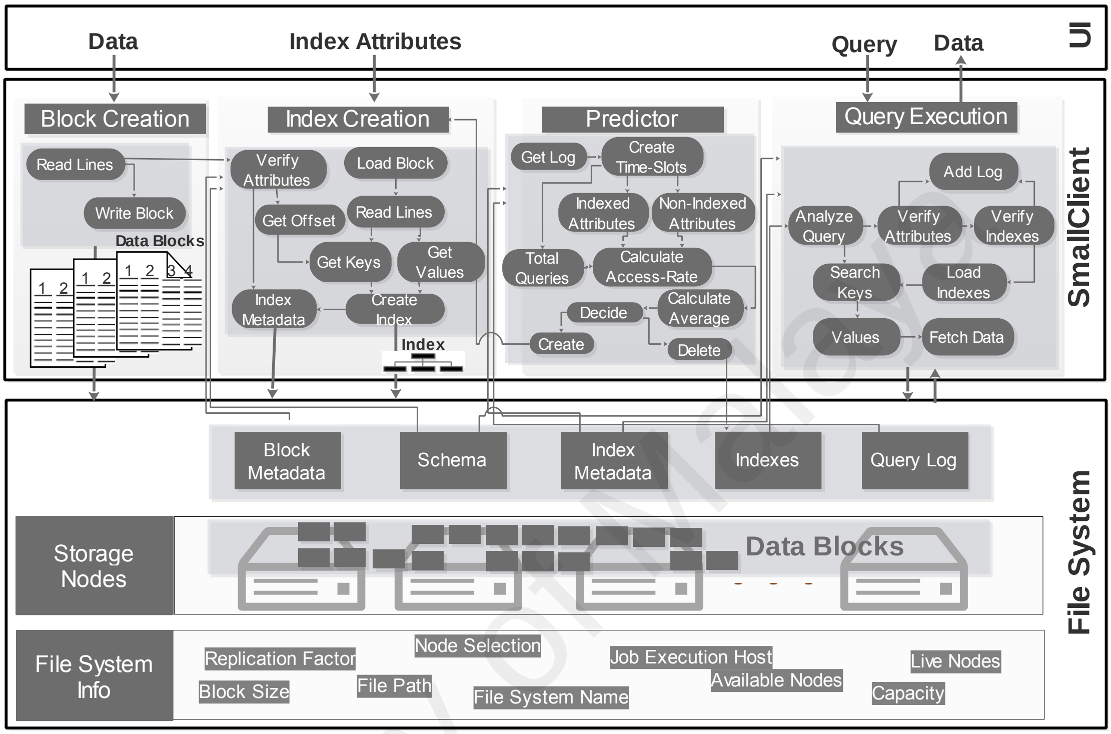


Figure 4-2: The Architecture for SmallClient

We present a file system data retrieval architecture for SmallClient in Figure 4-2. SmallClient is an intermediate layer between user input/output and file system layers to support the data retrieval process by offering indexes on attributes that are most expected to be the selection predicates of incoming queries. SmallClient receives queries from user as input, loads relevant indexes from the file system, traverses these indexes to find the location of data, and retrieves required data from the location returned by indexes.

The first layer is the user interface layer from which the user interacts with SmallClient. The user provides the data set location and its schema and invokes the block creation module to upload this data. The user can also provide index attributes along with data to create static indexes during data uploading. However, index creation can be invoked any time if the data are already in the file system. The user submits a query to SmallClient for data retrieval and obtains the required data from the user interface layer.

The second layer of the architecture contains the procedures of the SmallClient framework. This layer performs block creation, index creation, and query execution processes. The block creation module takes data as user input, reads data records individually, and fills them in the block buffer and then uploads the blocks to the file system. The index creation module takes data blocks and index attributes as input either from the user or from the file system to extract <key, value> pairs, generates metadata for each index, and uploads indexes and index metadata to the file system. The query execution module takes queries as input, uses indexes for data search if indexes are available, and returns requested data to the user if found in the data set. The query execution module updates the query log after processing each query. The predictor uses the query log to analyze past query trends and decides to update indexes by predicting future query workload.

The third layer is a file system layer that comprises files and file system information that are involved in different SmallClient processes. Files contain data blocks and supporting content for a data set such as block metadata, schema, index metadata, indexes, and query log. These files reside in storage nodes. The file system replicates files for a data set based on its available replication factor information. SmallClient also uses file system information to access the file system. This information provides basic statistics of nodes and directories in a file system.

4.3 Framework Modules

This section presents a brief description of all components of the SmallClient framework. As described earlier in this chapter, the SmallClient indexing framework has three execution modules: block creation, index creation, and query execution. In this section, we describe each module in detail. We further explain static and adaptive indexing and predictor logic offered by SmallClient.

4.3.1 Block Creation

In this section, we elaborate the block creation and data uploading process module of SmallClient. Contemporary big data processing systems offer distributed storage for big data when voluminous data are handled as small chunks. These chunks or data blocks are manageable pieces of large data sets. Each big data storage system has its own data splitting mechanism in which data block size and location to store each block is decided. Data uploading time should be very low for an efficient block creation process with minimum storage overhead on actual size of data set.

HDFS also has a unique block creation policy. HDFS uses contiguous bits from data to create blocks. The last record in a block usually faces breakage when HDFS splits data into fixed size blocks. In HDFS, the storage of two blocks that contain this broken record on a single site is not expected. Therefore, HDFS incurs high processing cost to access more than one node to retrieve the broken record.

We propose our own block creation method to avoid the processing costs that are associated with accessing multiple sites for a single broken record. Our proposition to place distinct records on a single site, thereby decreasing the time required to access the resulting records. We introduce block creation such that the last record in a block is never split.

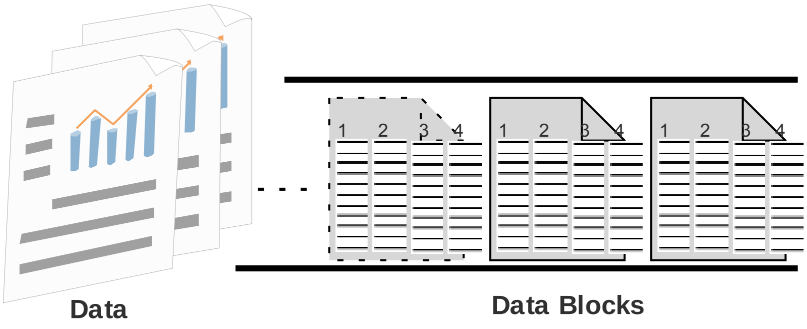


Figure 4-3: Block Creation Process

Figure 4-3 presents the block creation process of SmallClient. The block creation module starts reading records and stores these records in a temporary container with a pre-defined block size until the container does not have any capacity to store more bytes with larger record size. This container is uploaded to a file system as a block of a data set. All blocks are created and uploaded one by one as a data set to the file system. For the last block, the container continues to store records until the end of the file, and this container is handed over to the file system for storage.

SmallClient offers adjustable block size and replication factor for a data set during block creation. The block size should be adjusted such that a minimum space remains in a block after the records are kept. The size of records may be sufficiently large. When the default block size of HDFS is utilized, a significantly large unused space may exist in each block and the overall size of uploaded data for a data set increases. Therefore, having an adjustable block size reduces unused space in each block.

SmallClient also offers an adjustable replication factor so that users can specify the number of replicas for a data set. A better tradeoff between available storage nodes and data size should be achieved by adjusting the replication factor for a data set based on storage capacity and data availability requirements. The block creation module incurs data size and uploading time overhead unlike the HDFS data uploading process. The overhead on data size is due to the additional unused space in blocks caused by storing variable size records in fixed-sized blocks, whereas uploading time overhead is incurred with additional time required to read records of a data set. However, this overhead is minimized for larger data sets, which requires a longer time for HDFS to upload.

4.3.2 Index Creation

This section discusses the index creation process module of SmallClient. Indexes are relatively a small search space for a data set to determine the location of required data. To improve query execution and data search performance for big data, fast traversable indexes with minimum indexing overhead must be created.

The SmallClient index creation module solves the problems of both clustered and non-clustered approaches, as highlighted in Chapter 3. SmallClient uses B-Tree structure for indexing, which is a non-clustered NAI indexing technique to overcome the replica dependency problem of clustered approach, thereby resulting in maximized index hit ratio. The B-Tree structure ensures less indexing time and size overhead than the existing inverted index approach, which is implemented by Lucene library for big data indexing. Thus, indexing overhead is minimized, query execution performance is improved, and index hit ratio is increased.

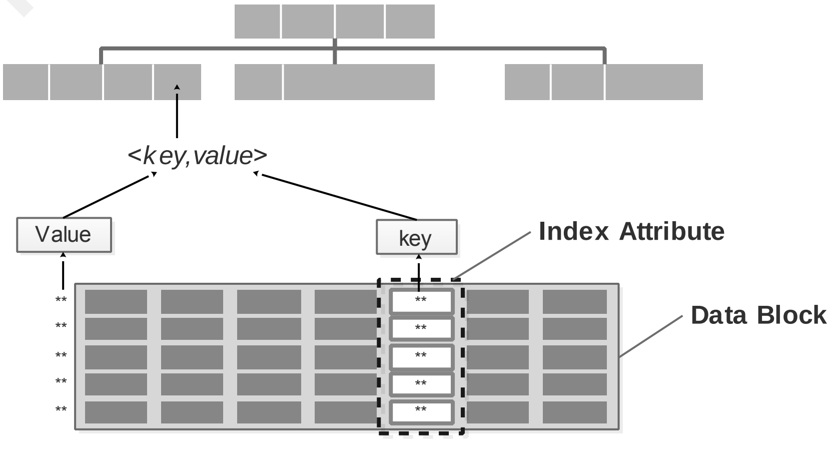


Figure 4-4: Index Creation Process

The index creation process is presented in Figure 4-4. The process begins by obtaining the list of attributes (i.e., index_attr_list) either from users or from predictors for which indexes should be created. This list is further verified with a schema of data set to ensure that the provided attribute names are correct and to remove unmatched attribute names. The system obtains the offset address from the schema for each of the verified index attributes during verification to determine the position of the index attribute in the records. These offset addresses are further utilized to access keys. All $< \text{key}, \text{value} >$ pairs from a data block are added to the B-Tree structure. The indexes of a data block and index metadata are stored in a file system. The sequence flow of index creation is presented in Figure 4-5.

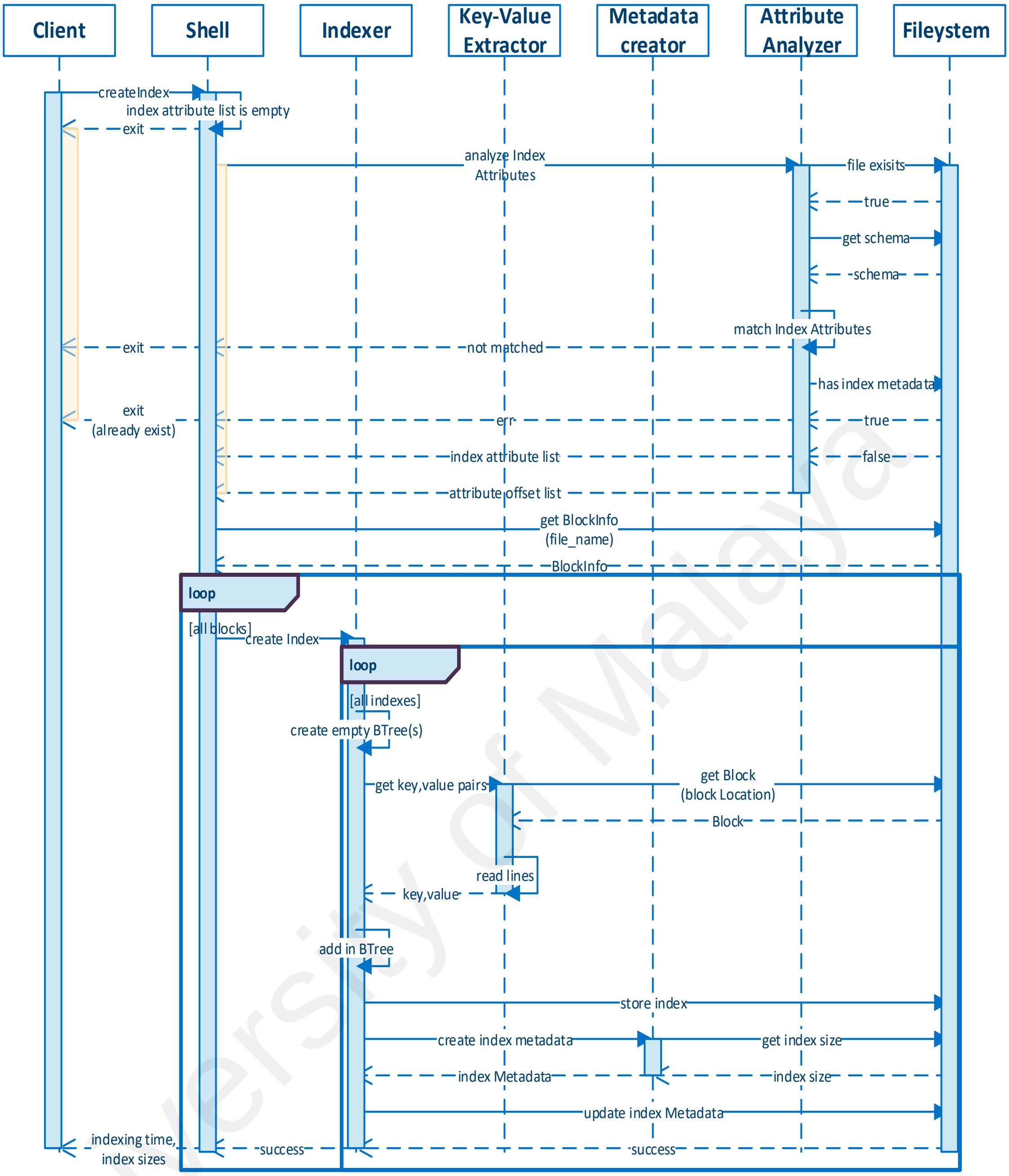


Figure 4-5: Sequence Diagram of Index Creation

SmallClient offers index creation at three stages of data handling in a file system: (a) index creation at data uploading time and (b) any time a user feels the need for additional indexes and adaptive index creation upon the recommendation of predictor logic. These three index creation options are elaborated as follows:

- Index Creation during data uploading (Static):

SmallClient offers index creation during the block creation process of data uploading. Users specify a list of index attributes along with the data. Figure 4-1 shows that the block creation module reads data records and adds these records to a block container. If the list of index attributes is not empty, these records are input to the index creation process, which extracts key(s) from records and calculates the offset of a record. Performing index creation with data uploading is very efficient because indexes are created in parallel with the block creation process. The delay to start query execution is significantly less than invoking index creation after the data uploading process. Index hit ratio is maximized because users initially know the attribute data to be queried and provide these attributes as a list of index attributes. However, users are free to perform index creation in parallel or separate from data uploading. Static indexes may not always be useful when query workload changes.

- Index Creation/Deletion when required (Adaptive):

Users can invoke index creation any time. Figure 4-1 shows that the index creation module loads each block for a data set from a file system into memory, reads records from a block one by one, extracts $< \text{key}, \text{value} >$ pairs, and adds these pairs to new indexes. Users can specify the list of indexes to be deleted, which will not be further utilized by incoming queries. Indexes require significant space to be stored. Therefore, when a user feels the need for new indexes or knows that available indexes will never or rarely be utilized by incoming queries, the user invokes index creation or deletion methods.

- Predictor Logic for Adaptive Indexing (Adaptive):

SmallClient offers automatic index updating based on predictor logic decision. Users have a choice to invoke an index creation or a deletion process when required. However, users can only invoke index updating when incoming queries are predictable. The predictor function of SmallClient offers adaptive indexing for unpredictable and changing query workload, which works based on historical data obtained from query log and automatically updates indexes. Query log retains the information of past queries, i.e., selection predicates of queries and query submission time of both hit and missed queries. Hit queries were executed using indexes, whereas missed queries occur when indexes were not available for any or all selection predicates. Predictor logic decides to create new indexes based on hit queries and to delete existing indexes based on missed queries.

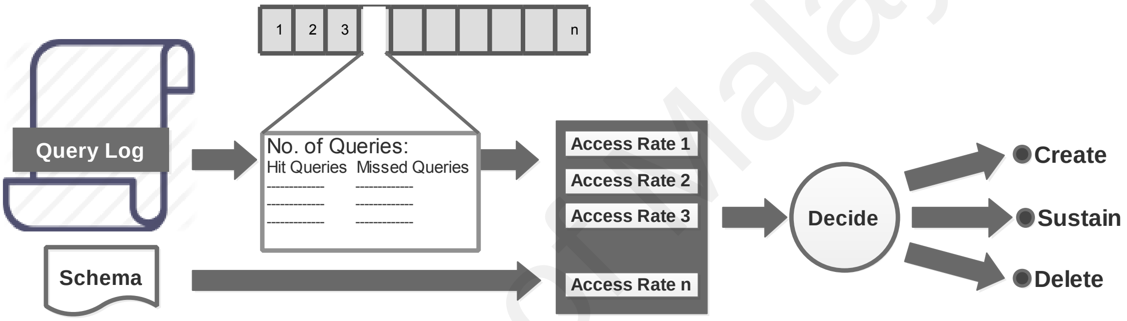


Figure 4-6: Predictor Function

Figure 4-6 presents the detailed process of the predictor function. Query log is loaded to the main memory to analyze queries in n time slots for prediction. Using only some recent queries is impossible when query workload changes, which indicates that most incoming queries do not have the same selection predicates. Query log is divided in equal time slots, and 10 recent time slots are considered for prediction (n=10). The number of total queries, hit queries, and missed queries may vary in each time slot.

All the attributes of a data set from a schema are obtained, and their access rates in the respective time slots are calculated based on available information in time slots. The access rate is zero for an attribute when it is never submitted as a selection predicate in queries (calculation of access rate is defined as Equation 5-20 in Section 5.3). When *n* values of the access rate for each attribute are obtained, the average access rate is calculated. The average access rate is used to make decisions.

The predictor function decides to create indexes for non-indexed attributes when the Average Access Rate is greater than the pre-set threshold value (i.e., *create_threshold_value*). Predictor function decides to remove an index for indexed attributes when Average Access Rate is less than the pre-set threshold value (i.e., *remove_threshold_value*). The *remove_threshold_value* is lower than the *create_threshold_value* because deleting an index is highly critical. Low accuracy in prediction may result in the deletion of indexes, which may be utilized by incoming queries.

4.3.3 Query Execution

The decisive module of our indexing framework is query execution. Query execution takes queries that are submitted by users as input, searches data required by users, and returns the data if these data are found in a data set. The overall time taken to traverse a data set and retrieve required data should be very low to achieve efficient query execution.

Query execution occurs when full scan or indexes are used. Existing full scan is performed by leveraging Apache Hive warehouse, which offers an SQL-like query language called HQL and utilizes MapReduce for efficient execution. However, as stated in Chapter 3, indexed search is more advantageous than full scan. Queries are executed by using indexed search when indexes are available. As discussed in Chapter 3, both clustered and non-clustered approaches were recently implemented for big data indexing to reduce time consumption in query execution. However, this search performance must be improved by using SmallClient, which implements a fast traversable indexing method to decrease query execution time.

A query execution module that utilizes indexes created by the index creation module is presented. Queries, for which indexes are not available, are executed using full scan. Submission of queries invokes the query execution module, and the module analyzes query strings before execution. During query string analysis, SmallClient separates the sel_data_list, file_name, and selection predicates. sel_data_list specifies the attribute names for the data that should be retrieved. Selecting the predicate of a query consists of two parts: the name of attributes and the value of attributes. The name of attributes should match any of the stored index names to perform indexed search, whereas the value of attributes is used as a key to search an index. When the query string is analyzed, SmallClient verifies all the collected parameters. The process of query execution can be conducted only if valid sel_data_list, file_name, and selection predicates are provided.

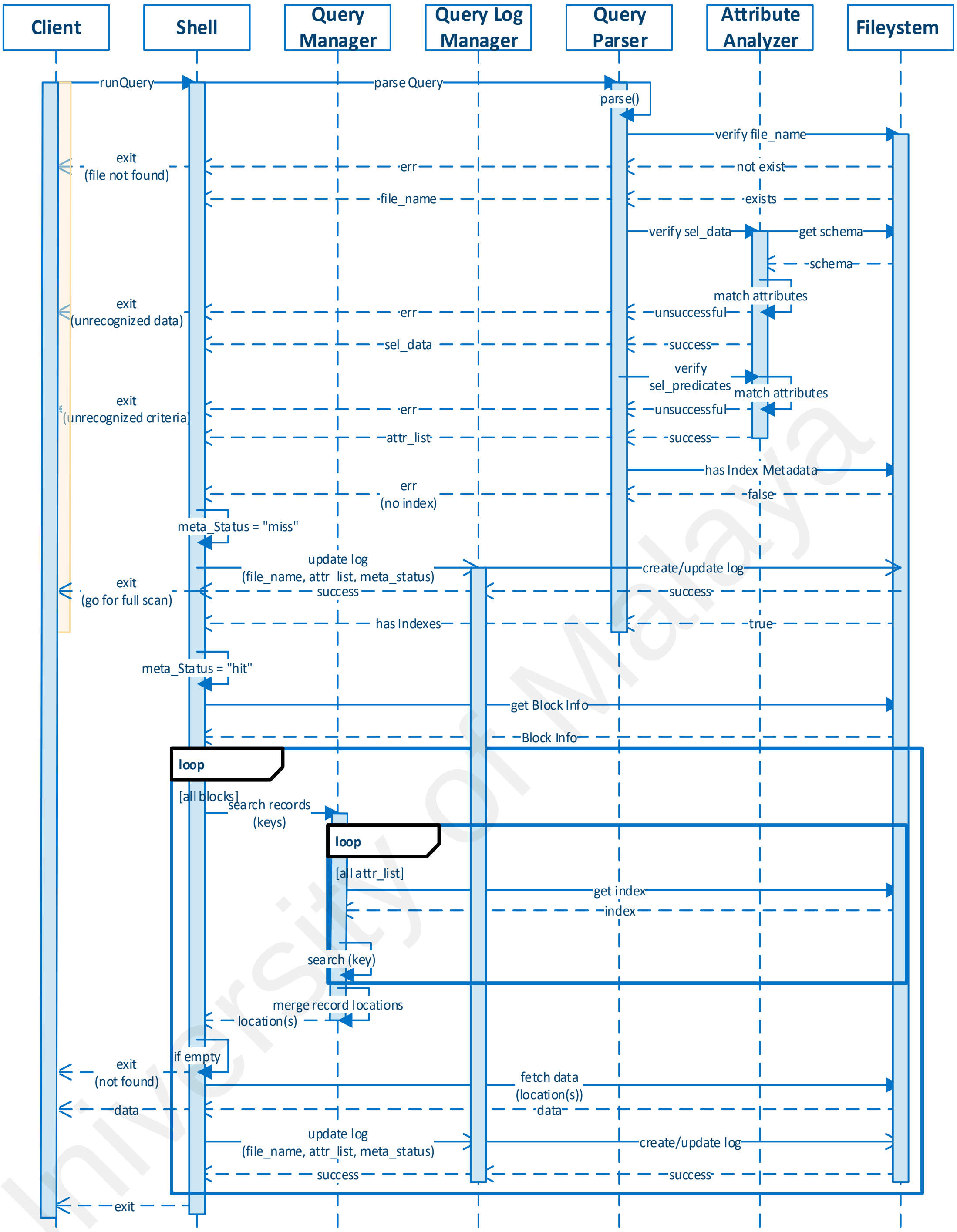


Figure 4-7: Sequence Diagram of Query Execution

The sequence flow of query execution is presented in Figure 4-7. Respective indexes are loaded to the memory, and the indexes are traversed to determine the location of records. The data from the file are obtained by directly accessing the location of expected records.

4.4 Mathematical Model for SmallClient

The mathematical model for SmallClient is described in this section. SmallClient modules are modeled by using Colored Petri Nets (CPN) tools, which leverage the mathematical modeling language of Petri nets. The mathematical modeling results are compared with the results obtained from experiments to verify the correctness of SmallClient results.

CPN tools (Jensen, Kristensen, & Wells, 2007) were utilized, which were broadly used for modeling and analyzing concurrent systems. Each module of SmallClient is implemented using CPN Tools. Basic graphical notation and primitives for modeling are implemented by adopting built-in discrete-event modeling language, whereas standard meta-language (ML) is utilized to define data types, describe data manipulation, and create models. The time for each activity of the modules is set in milliseconds (ms).

The number of records in a data set and the block size for the block creation module are specified. The data set, which is in the form of records, resides in the local disk from where each record is read and added in a block if the block size is appropriate. When a block reaches its maximum size, it is sent in HDFS for storage. The index creation module is modeled parallel to the block creation module. However, the index creation process can also be invoked when records are read from HDFS instead of the local disk. The indexes from HDFS are accessed for the query execution module, and these indexes are traversed to search the selection predicates provided in a query. Index traversal returns the locations to retrieve required records.

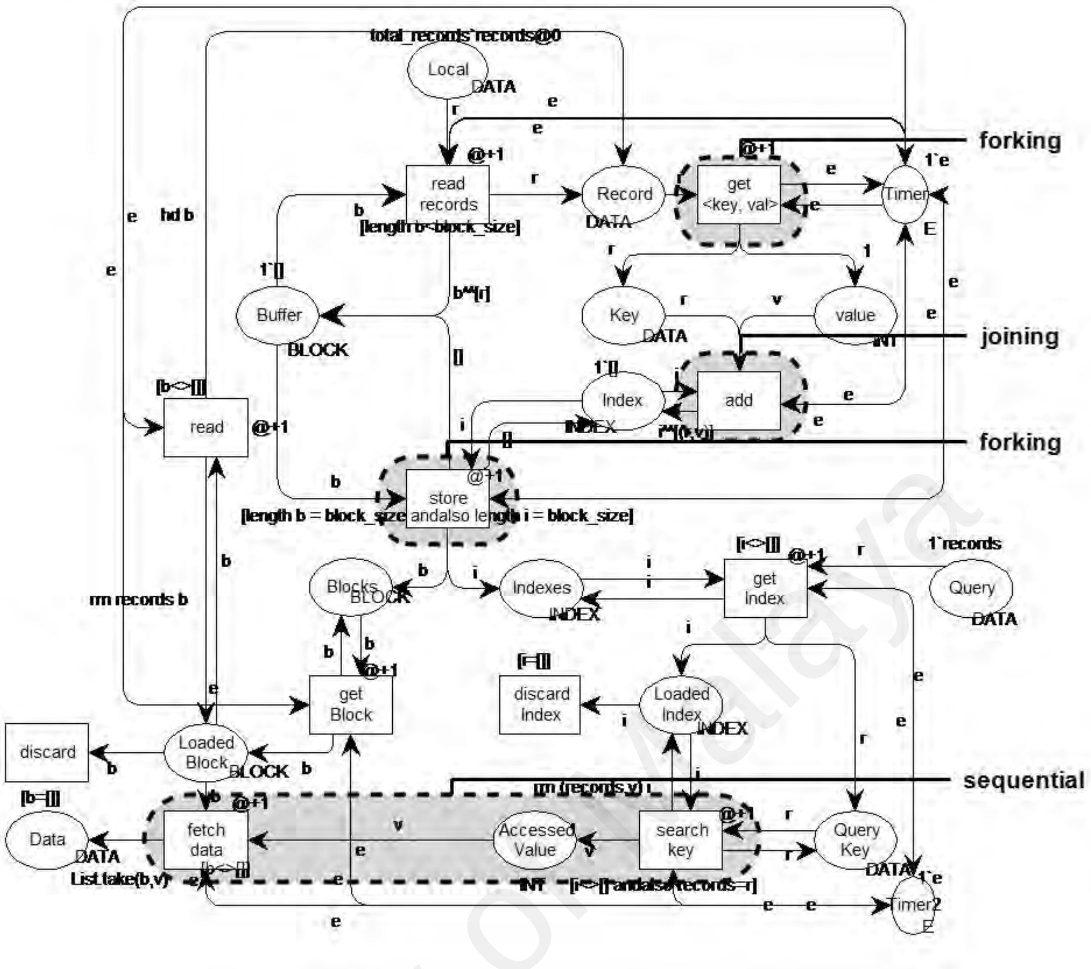


Figure 4-8: Mathematical Model for SmallClient

Figure 4-8 presents the mathematical model for SmallClient, which comprises places presented as ovals, transitions denoted as rectangles, input and output arcs shown as arrows and initial marking, respectively. The proposed model for SmallClient contains 16 places and 11 transitions. Each place and transition in the model is explained in Table 4-1 and Table 4-2, respectively. Two timers were added to this model to collect the time results. The first timer is used to calculate data uploading and index creation time, whereas the second timer (i.e., Timer 2) is utilized to calculate query execution time.

The mathematical model elaborates that sequential, forking, and joining operations are performed. For instance, search key and fetch data are executed sequentially, whereas both store and get <key, value> are forking operation that generate outputs for two places. Add is one of the joining operations. Read Records, Get <key, value>, Store Block, Store

Index, Get Block, Get Index, and Fetch Data transitions in the proposed model are timed transitions, and each transition entails a certain amount of time. Firing time is defined for these transitions in the next chapter. The timed transitions show that block creation and index creation times depend on the number of records of a data set and block_size (number of blocks also influences these times). Query execution time depends on the number of blocks and the number of instances in an index.

The block creation module and start execution from the place Local are modeled. The initial marking, i.e., total_records, denotes the number of records in a data set. These records are read one by one with an increment in timer value and placed in a buffer until the buffer reaches block_size. Store block transition is enabled when the buffer is full (see corresponding guard function in Figure 4-8). This transition also increments a timer. The process continues to upload the entire data set.

The index creation module is initialized from the place Record. This place accepts records either from read records transition, which is enabled during block creation, or from read transition, which creates indexes for data blocks residing in HDFS. The keys and values from each record are obtained and sent to Key and Value places while the timer is incremented. The transition adds two tuple <key, value> pairs in the index. Store index transition is enabled when the index has entries for all records of a data block. This transition also increments the timer.

The query execution module of SmallClient is modeled by taking the token as input from the Query place. Index transition is enabled when query and indexes are available in HDFS. Indexes are loaded in the memory with one time increment, and the selection predicates of queries are sent to Query Key place. The search key transition searches keys in the loaded index and returns the value. Search key transition also increments the timer.

The data from the location specified by the token in the Value place are retrieved from the loaded data block while incrementing the timer by one.

Table 4-1: Description of Places in SmallClient Mathematical Model

Place	Description	Initial Marking
Local	Local place contains data set residing on local disk	No. of records in a data set
Buffer	Buffer place contains the in-writing block	Empty
Blocks	Blocks place contains data blocks stored on HDFS, each block is up to specified block_size	Empty
Timer	Timer place contains time taken in block creation and/or index creation	One token initially having 0ms time
Record	Record place contains one record and is used for indexing	No record
Key	Key place contains the contents of record for indexing	Empty
Value	Value place contains the record location	Empty
Index	Index place contains array of key-value pairs	Empty
Indexes	Indexes place contains indexes stored on HDFS	Empty
Query	Query place contains query	At least one
Loaded Index	Loaded Index place contains index which is loaded from HDFS and is utilized by queries	Empty
Query Key	Query Key place contains selection predicate	Empty
Accessed Value	Accessed Value place contains value of the matched key in query execution	Empty
Loaded Block	Loaded Block contains records of a block loaded from HDFS to create indexes. Loaded Block is also used to retrieve data for a query.	Empty
Data	Data place contains records which are retrieved for a query	Empty
Timer 2	Timer 2 place contains time taken in query execution and data retrieval	One token initially having 0ms time

Table 4-2: Description of transitions in SmallClient Mathematical Model

Transition	Description
Read records	Reads records one by one from locally stored data set and sends each record to buffer if buffer has capacity. Records are also sent for index creation.
Get <key,val>	Returns keys and values from incoming records
Add	Adds <key,value> pairs in index
Store	Stores blocks and/or indexes to HDFS
Get Index	Returns index from HDFS
Discard Index	Discards index from memory when utilized by a query
Search Key	Searches the input key from query in an index and returns value if key is matched
Get Block	Returns records of a block stored on HDFS
Read	Reads records one by one from a loaded block and sends for index creation
Discard	Discards data block from memory when all records are read
Fetch Data	Returns data residing on an input location

4.5 Features of SmallClient

Aside from the performance gains of indexing, some quality aspects are considered to design and develop this framework. Therefore, a big data indexing client, which is deployable on contemporary distributed file systems, exhibits the following properties:

- Adjustable Block Size

Designing an indexing client with data split utility allows the size of data blocks to be adjusted according to manageability of the file system. However, the default block size of the file system can also be utilized. Block size plays a significant role in improving the utility of storage space. In the case of a fixed block size, a larger free space may be occupied by data blocks. For instance, if the size of records in a data set is large (i.e., approximately 10 MB) and the block size is 64 MB, every block will have up to 4 MB unused space. Although 4 MB does not seem to be an overhead for terabyte-scale data sets, which comprise thousands of blocks, 4 MB of unused space per block drastically affect the overall space consumed by the data.

When the block size (i.e., 60 MB) is adjusted during the block creation stage, this overhead can be minimized. Furthermore, this framework is deployable on this distributed file system, which allows data blocks with their own default block size. The block size in the data upload parameters can be adjusted according to the nature of data sets or based on the file system default block.

- Configurable Replication Factor

The replication factor for the data set and/or for indexes is also configurable in the indexing client. Depending on the availability of storage space and other factors related to imposing replication, the big data indexing client allows the replication factor to be adjusted. A fixed number of replicas when data are uploaded on a file system has its own implications: the capacity of a file system may prohibit more replicas of large data sets.

The physical condition of a cluster (i.e., distance between storage nodes and/or contingency of storage node failure) may require more replicas to ensure data integrity. The replication of indexes is subject to the access of the indexes. More replicas of indexes are suggested if more users are expected to access these indexes. However, the indexes are fairly small for big data sets. Thus, storage space limitations of a file system do not prevent the number of index replicas to increase.

The default replication factor of HDFS is used for data storage, which is three. The indexes were not replicated for evaluation purposes in this thesis because the data loss, which is attributed to the rare node failure observed in the cluster, is not predicted. Furthermore, the storage nodes in the cluster are physically co-located. Therefore, the proposed replication factor of HDFS is sufficient to ensure data availability and integrity.

- Adaptable to changing query workload

The predictor function of SmallClient automatically updates indexes for a data set with changing query workload. Static indexes, which are created based on user-specified index attributes, may not fulfill incoming query needs when users cannot predict future workload of incoming queries. In this case, SmallClient is sufficiently smart to decide new indexes. Query log is maintained, which stores the history of incoming queries. This query log is utilized by the predictor function, which determines whether each attribute is indexed or non-indexed to identify its access rate in recent past queries. New indexes are created. Non-indexed attributes are analyzed from missed queries. Existing indexes are deleted, whereas non-indexed attributes are analyzed. Predictor function keeps the indexes up-to-date and adaptive to query workload. Thus, the increased index hit ratio is achieved using SmallClient.

- Support to big data indexing requirements

The big data indexing requirements were explained in Chapter 2. SmallClient effectively supports the volume, velocity, and variety of big data. Varying size data sets were taken to show the capability of SmallClient to support big data. The experiments show that indexing overhead decreases and search performance increases with data size (see Chapter 6).

With the rapid growth of data volume, index update is straightforward in SmallClient. SmallClient suggests block level indexing where indexes for each data block are separately created without the intervention of indexes on other data blocks. Block level indexing takes specific data block as input, reads all records one by one, and creates indexes for the provided list of index attributes. When data are added in the existing data set, new blocks are easily indexed.

Data sets from various sources, such as spatial data, are utilized where shape files are converted into CSV files to ensure the variety of data. Any type of data set with metadata and schema can be indexed using SmallClient.

4.6 Conclusion

The proposed SmallClient indexing framework and its modules, namely, block creation, index creation, and query execution, are presented in this chapter. The layered architecture of query execution on a file system for big data is described. The components of SmallClient and their process flows are presented by using sequence diagrams. The mathematical model for SmallClient is also shown.

The block creation module of SmallClient overcomes the problem of record splitting and improves record retrieval performance by reducing the number of MapReduce jobs required to access jobs. However, block creation results in negligible data uploading time and data set size overhead for big data sets.

The index creation module of SmallClient achieves minimized indexing overhead. The index creation module implements non-clustered NAI indexing approach, which overcomes the problems of existing clustered indexing approaches for big data. The adopted approach is specialized to demonstrate its lower index time and index size performance than that of contemporary non-clustered implementations.

The query execution module of SmallClient ensures improved search performance. SmallClient indexes are fast traversable, thereby quickly returning the record offset(s) of required data. Data retrieval operation jumps the offset and displays required data.

Predictor logic, which is associated with the index creation module of SmallClient, confirms the maximized index hit ratio by suggesting automatic index updating. Adaptive index updating adds new indexes and removes existing indexes based on changing query workload. Thus, the maximum incoming queries are served by available indexes.

SmallClient supports the large volume, velocity, and variety from big data indexing requirements, and is therefore efficient for big data.

CHAPTER 5: EVALUATION

This chapter aims to present the evaluation of the proposed indexing framework for indexing overhead, query execution time, and index hit ratio. The test bed, which specifies hardware and software utilized in experiments to collect data, is presented. The evaluation measures are discussed, and the algorithms are presented as framework logic to execute the experiment. All the modules of SmallClient are implemented by using Eclipse IDE on a physical four-node cluster, and data are collected for data upload overhead, indexing overhead, and search performance. The tools to gather data for the experiment, benchmarking, and mathematical modeling are also described in this chapter.

This chapter has four sections: Section 5.1 presents the test bed by explaining the hardware and software specifications. Section 5.2 discusses the evaluation measures, whereas Section 5.3 explains the algorithms used to execute the framework. Section 5.2 elaborates the data collection tools, and this chapter is concluded in Section 5.5.

5.1 Test bed

The experimental setup, including hardware and software specifications, is explained in this section. The hardware consists of four physical machines. Each machine has 250 GB disk storage, 4 GB RAM, and 2 GHz processor. The operating system runs on 64-bit Ubuntu Desktop latest stable release. The available cluster size is 1 TB.

A four-node Hadoop cluster is configured on these machines. MapReduce and HDFS daemons are configured. One master and four slave nodes are created. MapReduce and HDFS daemons are configured on the master–slave cluster for data processing and storage purposes.

The default configuration of Hadoop is used for most data sets, i.e., default replication factor is three and the default block size is 64 MB. However, SmallClient allows custom configuration according to user needs and data set requirements.

The framework modules, i.e., block creation, index creation, and query execution, are developed for evaluation. Each module of SmallClient is implemented, and several Java, Lucence, and Hadoop packages are used. Eclipse IDE is utilized for code implementation. These modules can be executed from any client in a cluster.

5.2 Evaluation Measures

The metrics, which are used to evaluate the performance of the proposed indexing framework, are discussed. These metrics are as follows: data upload overhead, indexing overhead, search performance, and index hit ratio. The definition of each measure is provided by using equations. The framework is validated and the results are verified using these metrics.

The evaluation measures are used to obtain the results from each module of SmallClient. The results show the extent to which the research aim and objectives are achieved. An explanation of each established evaluation measure is presented in this section.

5.2.1 Data Upload Overhead

Data upload overhead is the percentage of increased activity to upload data using the block creation module of SmallClient for a big data processing file system in comparison with their own data upload policy. The data upload overhead is measured in terms of both data upload time overhead and data size overhead. Both overheads are defined as follows:

Data upload time is composed of the time (in seconds) to create data blocks and store these data blocks on the file system. Chapter 4 indicates that the block creation process reads records from the data set and maintains the records in a block until the block does not have any capacity to store more records ($\sim block\_limit$). Data upload time is calculated in the following equation for a data set with k blocks:

$$T_{blockCreation} = \sum_{c=1}^{k} (T_{create(\mathcal{B}`_c)} + T_{upload(\mathcal{B}`_c)}) \qquad \textbf{5-1}$$

The time taken by the block creation module of SmallClient is utilized as data upload time. Therefore, $T_{blockCreation}$ is used to define data upload time where $T_{create(B'_c)}$ denotes the time to create a block c and $T_{upload(B'_c)}$ indicates the time to upload a block. The percentage increase in data uploading time defines the upload time overhead. This overhead is defined as follows:

$$O_{dataUpload} = \frac{T_{blockCreation} - T_{uploadData}}{T_{uploadData}} \times 100 \qquad \textbf{5-2}$$

where $O_{dataUpload}$ denotes data upload overhead, $T_{blockCreation}$ is the data uploading time of SmallClient, and $T_{uploadData}$ is the data upload time of the existing file system.

The size of uploaded data (in MB) using SmallClient also differs from the actual size of data sets. While records of data are added to a block where block size is same for all blocks for a data set, some space of few bytes are left empty in a block container. However, the configurable block size offered by SmallClient allows the block size to be adjusted according to the size of records. Therefore, the overall size overhead becomes negligible. The size of each block (i.e., $\mathrm{S}_{\mathrm{B}`}$) and the overall data set size (i.e., $\mathrm{S}_{\mathbb{D}`}$) are defined in the following equations, where $\mathrm{S}_{\mathbb{D}`} - \mathrm{S}_{\mathbb{D}} < l$:

$$S_{B^`} = l \qquad \textbf{5-3}$$

$$S_{\mathbb{D}^`} = S_{B^`} \times k \qquad \textbf{5-4}$$

The size of SmallClient blocks $\mathrm{S}_{\mathrm{B}^`}$ is the configured block size $l$, and the size of an entire data set is the product of block size $\mathrm{S}_{\mathrm{B}^`}$ and the number of blocks in data set which is denoted as k.

Data size overhead $O_{dataSize}$ is calculated using Equation 5-5.

$$O_{dataSize} = \frac{\mathrm{S}_{\mathbb{D}^`} - \mathrm{S}_{\mathbb{D}}}{\mathrm{S}_{\mathbb{D}}} \times 100 \qquad \textbf{5-5}$$

where $\mathrm{S}_{\mathbb{D}^`}$ denotes the size of data sets uploaded using SmallClient, and $\mathrm{S}_{\mathbb{D}}$ indicates the data set size uploaded using HDFS.

5.2.2 Indexing Overhead

Indexing overhead is the increased cost of performing the index creation process on data sets. Although indexing can improve data search performance, the cost to create indexes should not be high to execute queries using full sequential scan. Therefore, indexing overhead is a significant evaluation measure for big data where users do not tolerate long delays to start query execution after uploading their data (Idreos, Alagiannis, Johnson, & Ailamaki, 2011). This overhead is measured in terms of both index creation time and the size of indexes.

Index creation time is the time taken to extract $< \text{key}, \text{value} >$ pairs according to an index attribute from each block. Each pair is at the right place in the B-Tree index, and the index is uploaded to the file system. According to Algorithms 5-2 and 5-3, index creation time varies with the number of blocks i.e. k, index attributes, and records in a block i.e. m_i. The indexing time and indexing overhead for a data set with k blocks are defined in the following equations:

$$T_{indexing} = \sum_{i=1}^{k}\left(\sum_{record=1}^{m_i} T_{get\ \&\ put(<key,value>pairs)}\right) + T_{storeIndexes} \qquad \textbf{5-6}$$

$$O_{indexing} = \frac{T_{indexing}}{T_{uploadData}} \times 100 \qquad \textbf{5-7}$$

Index size S_I refers to the aggregated size of B-Tree objects i.e. $S_{B-Tree_{i,c}}$ for all index attributes and all k blocks. The number of index attributes j, size of keys, and number of records in a data set affect index size S_I. S_I is defined in the following equation:

$$\mathrm{S}_I = \sum_{attribue=1}^{j}\sum_{c=1}^{k} S_{B-Tree_{i,c}} \qquad \textbf{5-8}$$

Index size overhead $O_{index\ size}$ is defined as follows:

$$O_{index\ size} = \frac{\mathrm{S}_I}{\mathrm{S}_{\mathbb{D}}} \times 100 \qquad \textbf{5-9}$$

5.2.3 Search Performance

Search performance refers to the percentage of improvement in the data search process. Indexing primarily aims to reduce query execution time to retrieve specific data from big data. Therefore, the percentage of reduced query execution time because of indexing indicates the overall data search performance. Query execution, which uses full scan, traverses all records in each data block despite the millions of records in a data set. By contrast, smaller indexes are traversed for query execution in the case of indexed data. Thus, search performance improves. However, indexing is a costly process. Thus, the search performance is expected to be significantly higher than the cost of index creation. An efficient index structure should be faster in traversal to improve search performance.

Search performance is used to evaluate the query execution time with indexes, which are created using the index creation module. The query execution module takes a query as input, loads respective indexes in the memory, which were previously created, obtains record positions as values by traversing the indexes using keys provided in queries, and fetches the required data by directly accessing the records of a data set. Index traversing time and data fetch time for each block c are mathematically represented in the following equations. The traversing time for B-Tree structures is proven (Comer, 1979) as $\mathrm{O}(\log \mathrm{n})$. Therefore, the time to traverse indexes $TT_c(keys)$ at block c for j indexes given in the selection predicate is:

$$TT_c(keys) = \sum_{attr=1}^{j} \mathrm{O}(\log n) \qquad \textbf{5-10}$$

The indexes return the offset of data records as $value$, which are requested by queries. Time to retrieve data $T_{fetch(sel\_data)}$ is the product of the number of required records S_{value} and the time required to access each record T_{access}. $T_{fetch(sel\_data)}$ is defined as follows:

$$T_{fetch(sel\_data)} = S_{value} \times T_{access} \qquad \textbf{5-11}$$

The overall query execution time for all blocks of a data set is defined as:

$$TQ = \sum_{c=1}^{k} \left(TT_{attr,c}\right) + TF_{sel\_data,c} \qquad \textbf{5-12}$$

The query execution time depends on the length of selection predicates, number of blocks in a data set, and number of records to be retrieved against that query. Thus, the search performance is the percentage difference of query execution times of SmallClient TQ and full scan operation TFS.

$$Search\_Performance = \frac{TFS - TQ}{TFS} \times 100 \tag{5-13}$$

5.2.4 Index Hit Ratio

The index hit ratio (IHR) is the ratio of queries that are executed using indexes. IHR is introduced as a measure to evaluate the probable rate of incoming queries to be executed using indexes. If more indexes are created for a data set, the probability to execute queries using indexes increases and full scan is avoided. Incoming queries are unpredictable. The only way to increase the probability of executing these queries using indexes is to raise the index attribute space (aI). IHR is calculated using the following equation:

$$IHR = \frac{S_{\mathrm{aI}}}{S_{a\mathbb{D}}} \tag{5-14}$$

However, creating more indexes may adversely affect the indexing overhead. For instance, some index structures (Dittrich et al., 2012; Halim, Idreos, Karras, & Yap, 2012) do not allow more than one index to be created on each replica of a data set. Thus, more replicas of data sets are needed to create more indexes. In this situation, the index size for one attribute corresponds to the size of a data set and incrementally increases for each new index. Thus, IHR can help present the capability of an indexing structure to the maximum number of indexes, which can be created with a manageable indexing overhead.

IHR can also be improved when users know the incoming query workload and invoke index creation or updating according to the predicted query workload. However, users cannot always predict the query workload. In this case, SmallClient leverages historical information on queries from the query log to update indexes. This feature positively affects IHR. IHR is defined in Equation 5-23 for adaptive indexes, which are updated with predicting query workload.

$$IHR_w = \frac{No.of\ Hit\ Queries}{Total\ Queries} \qquad \textbf{5-15}$$

5.3 Framework Design

This section presents the logic design of the proposed indexing framework by defining the different processes involved for block creation, index creation, and query execution modules, which were explained in the previous section. Algorithms and equations are utilized to define these processes.

Algorithm 5-1 createBlocks(*file*)

```
block_limit = DefaultBlockSize
has_capacity = true
block_number = 0
while reading records not reached end of file do
  if has_capacity then
    add record in block
  else
    uploadBlock(block, block_number)
    block_number = block_number + 1
    has_capacity = true
  end if
end while
uploadBlock(block, block_number)
Return
```

The process of block creation is presented in Algorithm 5-1. block_limit defines the block size specified for a data set. The default block size offered by HDFS is used. has_capacity determines whether the block container can contain an incoming record or not, and block_number is used to manage the sequence of blocks.

In Equations 5-16 and 5-17, based on the assumption that a data set $\mathbb{D}$ is composed of x records, block creation initiates by reading records one by one. Block $\mathcal{B}$` , which is created using the block creation module, has m records out of x, and the remaining bytes to reach block limit l are denoted as α. Overall, k blocks are created for $\mathbb{D}$.

$$\mathbb{D} = \sum_{c=1}^{x} record_c \tag{5-16}$$

$$\mathcal{B}`_i \ = \sum_{c=1}^{m_i} record_c \ + \ \alpha \ : 0 \leq i < k \tag{5-17}$$

When the block reaches block_limit, the block is uploaded, block_number is incremented, and block_limit is set to true. After uploading the last block, the process ends and returns the time taken in the block creation process.

Algorithm 5-2 runIndex(*file_name*, *file_schema*, *index_attr_list*)

```
if index_attr_list is empty then
  write err_message
  Exit
end if
compare index_attr_list with file_schema & remove unmatched attributes from index_attr_list
calculate index_attr_offset_list from updated index_attr_list
get block_info
for all blocks do
   createIndexes(file, block_locations, index_attr_offsets_list)
   for all indexes do
      storeIndex(index, file_name, index_attr )
      get & update index_metadata
   end for
end for
```

The next stage is creating indexes for uploaded blocks of a data set. The process of index creation is elaborated in Algorithms 5-2 and 5-3. Algorithm 5-2 shows the pre-index creation verification steps involved to obtain the exact index attribute set according to the provided schema of a data set. Users may mistakenly provide some attributes in index_attr_list, which are not present in the data set. Comparing index_attr_list with the schema helps remove these attributes from index_attr_list. The offset addresses of index attributes are also obtained from the schema, which are helpful to transform to contents as keys in a record. The index creation phase is invoked.

Based on the assumption that a$\mathbb{D}$ denotes the list of attributes of a data set that is available in its schema, index_attr_list, aI is defined in the following equation:

$$aI \subseteq a\mathbb{D} \qquad \textbf{5-18}$$

Empty B-Tree index objects are initialized for each element of index_attr_list after successful verification. SmallClient starts reading blocks line by line and obtains keys corresponding to the offset addresses and value from each record. As explained in the previous section, the content from a record for each index attribute is collected as key and the location of that record is collected as value. More than one occurrence of a key in a block are stored as list of values in B-Tree. As a result, we can define our index I for a block i as follows:

$$I_{attr,i} = \sum_{record=1}^{m_i} < key_{attr_{record}}, value_{record} > \qquad \textbf{5-19}$$

Algorithm 5-3 createIndex(*file*, *block_location*, *index_attr_offset_list*)

```
for all index_attr do
  create empty BTree
end for
value = block_offset
while reading records not reached end of block do
  for all index_attr do
    key = contents at index_attr_offset
    add <key, value> in its BTree
  end for
end while
store each BTree
store index_metadata of each BTree
```

Algorithm 5-3 presents the steps involved in creating indexes. The process begins with creating n (i.e., $n = \mathrm{S}_{\mathrm{index\_attr\_list}}$) empty B-Trees, which shows that the indexes do not contain any $< \mathrm{key}, \mathrm{value} >$ at this stage. The offset of block is assigned as the offset of the first record to value. The value of the next record is updated by adding the byte size of the record. The process continues until all records in a block are indexed. The indexes and their metadata are stored in the file system as small objects, which maintain the information of indexes.

SmallClient also offers adaptive indexes based on the proposed predictor logic. This function calculates the access rate for each attribute of a data set by using Equation 5-20, where $\mathrm{Occurence}_{\mathrm{attr,i}}$ denotes the number of queries in a time slot i with attribute attr as the selection predicate and $\mathrm{Total\ queries}_{\mathrm{i}}$ indicates the total number of queries in time slot i. Ten values of the access rate for each attribute are calculated because 10 time slots are used for prediction.

$$Access\ Rate_{attr,i} = \frac{Occurence_{attr,i}}{Total\ queries_i}\ , i = 1,2,3, \dots , n \qquad \textbf{5-20}$$

Algorithm 5-4 runQuery(*query*)

```
if analyze(query) is not successful then
    write $error\_message$
    Exit
end if
get & verify provided $file\_name$ from query
get & verify $sel\_data\_list$ from query
get $sel\_data\_offset\_list$ from $file\_schema$
get $attr\_list$ from selection predicate(s) of query
if indexes are not available for attr_list then
    go for full_scan
Else
    get values of attribute_list as $keys$ from query
    get $block\_locations$
    for all blocks do
        load respective index(es)
    search keys & fetch sel_data_list if keys are found
  end for
end if
```

The query execution process of SmallClient is explained. Algorithm 5-4 describes the process of executing queries using indexes. An incoming query is first analyzed to validate its syntax, and the parameters specified in the query are verified. Queries with typographical and syntax errors or queries that do not match any file in the file system are discarded after an error message is received. The successful analysis of query string provides attribute(s) as selection predicates to search data. Full scan operation is recommended only when indexes are not available for selection predicates.

5.4 Data Collection Tools

Data collection tools are presented in this section. Experiments are executed on the test bed to collect data for all modules of SmallClient. These data are further testified where benchmarking and mathematical modeling are employed. Data collection methods for experimental data verification are also presented.

5.4.1 Data Collection for Experiment

Java code is executed in Eclipse, and Apache StopWatch API and Hadoop FileStatus package were used to obtain most of the results. HDFS user interface and console were utilized for data collection. StopWatch was used to obtain the time results of different processes. Data uploading time, index creation time, and query execution time results are generated with Apache StopWatch API, which starts and ends with the process and displays the time taken by a process.

User interface is utilized to browse HDFS and Hadoop FileStatus package to obtain the size results. The HDFS user interface is used to collect the results uploaded file size, whereas Hadoop FileStatus package is used to generate the size of indexes. These Java packages generate reliable and accurate results without human intervention. Therefore, these packages were leveraged to achieve accurate results for further evaluation. Full-scan operation was performed using HQL queries in the console, and the query execution time generated by the console is determined as the full-scan search time.

5.4.2 Data Collection for Benchmarking

This section elaborates the method used to collect data for benchmarking for each performance measure. The process of executing different modules of the proposed framework was elaborated to collect data for data upload overhead, indexing overhead, and search performance.

Data were obtained for evaluation measures by executing SmallClient modules. Block creation and index creation modules can be invoked together or separately depending on user requirements. Users can also request to perform index deletion based on query workload knowledge. However, SmallClient offers adaptive index updating for unpredictable incoming query workload. SmallClient utilizes configuration information and block placement policy of Hadoop during the execution of these modules. Users can also invoke query execution module from any node that uses Hadoop block selection policy.

Block creation module of SmallClient offers custom data uploading, thereby resulting in data uploading time and data set size, which differ from HDFS. The results for data uploading time and data set size are collected and compared with the results for Hadoop default data uploading. The percentage overhead of data uploading of SmallClient was calculated to compare SmallClient data uploading results with HDFS data uploading. Data upload time and size results are presented in Chapter 6.

The indexing overhead results of SmallClient in terms of index creation time and index size are collected and compared with the results of Lucene indexing overhead. SmallClient also offers index creation with data uploading (called i-SmallClient). In this case, the indexing overhead of i-SmallClient is different from the overhead when indexes are created separately. Therefore, two evaluation methods exist for index creation. First, SmallClient data uploading without indexing results are compared with i-SmallClient results. The result shows the overhead caused by indexing. Second, the i-SmallClient results are compared with separate index creation in SmallClient.

The indexing results obtained from SmallClient are compared with Lucene indexing results. The comparison indicates that SmallClient performs better. Index updating includes the creation of new indexes and deletion of unused indexes. The current index deletion results are presented. The new index size overhead is calculated and compared with the previous index size overhead before being deleted. All index overhead results are presented in Chapter 6.

The query execution time and search performance results of SmallClient are obtained by executing multiple queries. The query execution time results of SmallClient are compared with the full-scan results of Hadoop using HQL and with Lucene indexed search results. The same queries are executed on Hive for full scan and on Lucene and SmallClient for indexed search.

The search performance of SmallClient is calculated over full-scan query execution and the search performance of Lucene indexes. The same process is employed to calculate search performance for both SmallClient and Lucene indexes. The search performance of SmallClient is compared with that of Lucene for evaluation.

The IHR results for both unpredictable query workload and after predicting query workload are obtained. The IHR results of SmallClient results for unpredictable and predictive query workload are compared with the IHR results of Lucene. IHR is improved because the indexes in SmallClient are adaptively updated. The results are presented in Chapter 6.

5.4.3 Data Collection for Mathematical Model

The time results are collected to mathematically verify and validate the correctness and reachability of the specified properties of SmallClient mathematical model. The state space report for each module provides fair information on some state space statistics and standard behavioral properties, e.g., integer bound of places and multi-set bound of places. A state space report is generated by specifying the number of records and block_size for a data set to be uploaded without index creation. A state space report is created for index creation with block creation and for separate index creation. Furthermore, a state space report for query execution is also generated.

Block creation in the mathematical model offers data uploading with a specified number of records in a data set and block_size. Data upload time results are obtained by observing the value of tokens at Timer place. The supply of records is halted to obtain <key, value> transition and the accurate results of data upload time, which also increments Timer. The data upload time is presented in following Equation 5-21.

$$Data\ Upload\ Time\ = (c \times S_{\mathbb{D}}) + (\mu_{read\ records} \times Local^{\#}) + (\mu_{store\ block} \times Blocks^{\#}) \qquad \textbf{5-21}$$

where c denotes the factor of data set size $S_{\mathbb{D}}$, which shows the effect of data set size on data uploading time. $\mu_{read\ records}$ indicates the time to read a record and $\mu_{store\ block}$ is the time to store a block. $Local^{\#}$ presents the number of records of a data set and

$\text{Blocks}^{\#}$denotes the number of tokens in Blocks place. The following values are set: 0.065 for c, 0.00006 for $\mu_{read\ records}$ and 3.5 for $\mu_{store\ block}$.

The indexing overhead results are obtained from the mathematical model of SmallClient in terms of index creation time. The model offers index creation during data uploading or for data blocks residing in HDFS. The value of tokens in a timer place shows the time taken to create indexes in both cases. The index creation time is presented in Equation 5-22, where $\mu_{get<key,val>}$ denotes time to obtain <key, value> pair from each record out of $Local^{\#}$. $\mu_{store\ index}$ indicates the time to store an index for a block where the total number of blocks in a data set are $Blocks^{\#}$. Equation 5-6 has also explained this indexing time calculation and shows that this time is accumulative to creating <key,value> pairs for data set and storing index into file system.

$$Indexing\ Time = \left(\mu_{get<key,val>} \times Local^{\#}\right) + \left(\mu_{store\ index} \times Blocks^{\#}\right) \qquad \textbf{5-22}$$

The query execution time and search performance results of SmallClient are collected by adding tokens in Query place. Query execution time results are the value of tokens at Timer2 place. The value of token at Timer2 place is updated by loading index in memory, traversing index to match keys, and retrieving data from Block place. Therefore, query execution time is calculated as follows:

$$Query\ Execution\ Time = \left(\mu_{get\ index} \times Blocks^{\#}\right) + \left(\mu_{search\ key} \times Local^{\#}\right) + \mu_{fetch\ data} \qquad \textbf{5-23}$$

where $\mu_{get\ index}$ denotes the time to load the index for a block to memory, $\mu_{search\ key}$ is the time to compare the value of the selection predicate with each key in an index and to return the relevant value when they are matched, and $\mu_{fetch\ data}$ is the time to load data from the file system. The explanation of $\mu_{fetch\ data}$ is provided in Equation 5-11.

$Blocks^{\#}$ are used to present the number of index files for a data set. Block level indexing is performed where indexes are managed at the block level. Thus, the number of index files is equal to the number of blocks. Each record in a data set is assumed to have a <key, value> pair in index. Thus, $Local^{\#}$ represents the number of comparisons performed to retrieve the location from index. As we have defined in Equation 5-12, query execution time includes time to traverse indexes and time to fetch data. $\mu_{get\ index}$ and $\mu_{search\ key}$ in Equation 5-23 define time to traverse indexes whereas $\mu_{fetch\ data}$ defines time to fetch data.

5.5 Conclusion

Performance evaluation of the proposed indexing framework is presented in this chapter. The test bed where all the modules of SmallClient are executed is explained to evaluate the effectiveness of the modules. The algorithms showed the implementation processes of the modules, such as block creation, index creation, and query execution, to collect data. Data collection tools presented a method to obtain data for evaluation measures. Data collections tools also explained the procedure to collect data for benchmarking and mathematical modeling. These data are utilized to verify the experimental data for each evaluation measure, such as data uploading overhead, indexing overhead, and search performance.

CHAPTER 6: RESULTS AND DISCUSSION

This chapter presents and discusses the results obtained by executing the proposed SmallClient. The objectives of proposing the SmallClient indexing framework for big data are achieved, as described in this chapter. Chapter 4 stated that SmallClient minimizes indexing overhead, reduces query execution and data search time, and maximizes index hit ratio. The experimental results of data upload overhead, indexing overhead, and search performance are verified by using benchmarking and mathematical modeling in this chapter.

The rest of the chapter is organized as follows: Section 6.1 presents the performance validation, and Section 6.2 presents the verification of the results by using benchmarking and mathematical modeling. Section 6.4 concludes the chapter.

6.1 Validation

The results of SmallClient for data upload overhead, indexing overhead, search performance, and IHR are presented and discussed. The effectiveness of SmallClient is ensured to fulfill research objectives, i.e., minimize indexing overhead, reduce data retrieval time with faster query execution, and achieve the maximum index hit ratio. Statistical tables and charts are used to present the results. Same data sets (Eldawy & Mokbel, 2015) are used to execute the experiment that were utilized for performance analysis in Chapter 3.

The data collection process was executed in 10 iterations for reliability. The data, which are collected for time results, exhibit a slight difference (i.e., few milliseconds). However, the size results are consistent in all 10 executions. Therefore, mode is considered from the observations for time results in evaluation, whereas the data for size results are used from any iteration.

6.1.1 Experimental Results

The experimental results are collected by executing experiments on a test bed, as described in the previous chapter (see Section 5.4.1). The data upload time and data upload size results are presented for block creation module validation. The indexing overhead, search performance, and IHR results from the experiments are described.

- Data Upload Results

Data uploading affects the time taken to upload data and the size of uploaded data. The results are presented from both perspectives. Table 6-1 shows the results of data upload time, whereas Table 6-2 indicates the results of data upload size.

Table 6-1: Data Upload Time Results

Data Sets	Data Set Size (MB)	No. of Records	Data Upload Time (sec)	
			SmallClient	I-SmallClient
Primary Roads	77.1	13373	15.46	17.78
Area Landmark	406	121960	57.03	62.54
Tabulation Area	1560	33144	172.89	212.15
Area Hydrography	6460	2298808	820.74	981.55
All Edges Combined (I)	16220	19291957	2535.13	3128.76
Linear Hydrography	18270	5857442	2252.09	2504.94
All Edges Combined (II)	23180	70000000	4962.00	7270.70
All Edges Combined (III)	61900	72700000	72304.00	11452.94
All Nodes	96400	2700000000	124010.00	135252.64
Road Network	137500	717000000	103788.00	114864.52

Table 6-1 presents the results of data uploading time taken by the experiments on the block creation module. Data uploading offers parallel index creation, which decreases overall delay to start query execution. The time results of data uploading are presented, as well as index creation. This module is called i-SmallClient. The index creation time for five indexes is included to present the i-SmallClient results. The results of SmallClient and i-SmallClient indicate that data uploading time slightly increased when five indexes are created in parallel.

Table 6-1 shows that data sets All Edges Combined (II) and All Edges Combined (III) consist of almost the same number of records, whereas their sizes vary significantly. Therefore, the difference between overheads of data upload time for these data sets is also very high. This result indicates that the size of the data set and the number of records in a data set both affect the data uploading time taken by SmallClient.

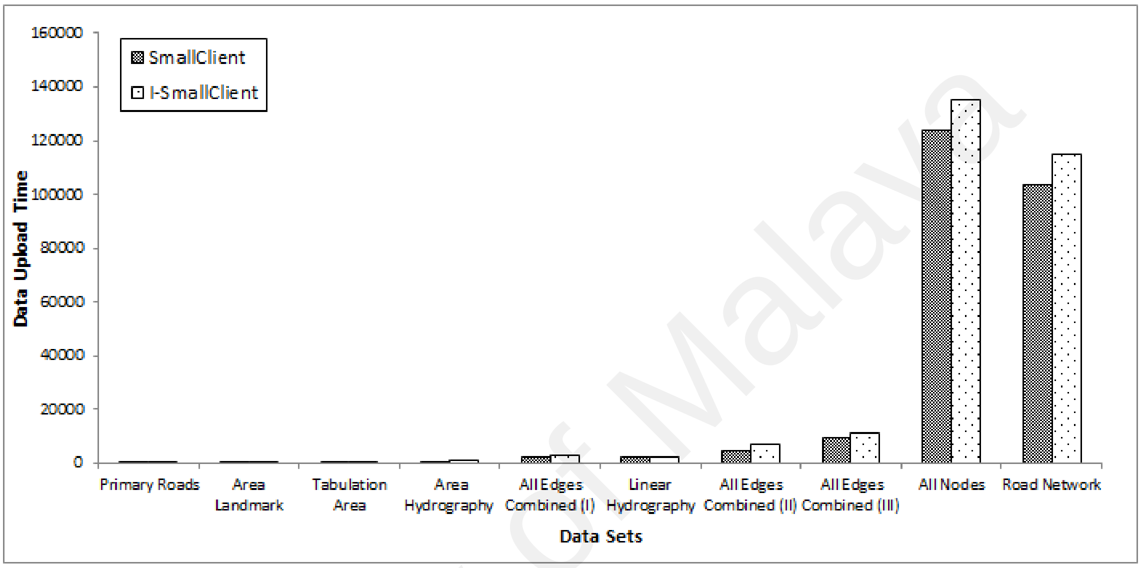


Figure 6-1: Data Upload Time Experiment Results

The data upload time experiment results of SmallClient and i-SmallClient are presented in Figure 6-1. The closed dotted bars indicate the data uploading time results for SmallClient, and dotted bars show the time for i-SmallClient. Figure 6-1 indicates that data uploading time increases with data set size. The experiment results also show that index creation in parallel to data uploading saves time as a minor increase in data uploading time occurs when five indexes are created during data uploading.

Table 6-2 presents the results of the size of uploaded data using SmallClient. Table 6-2 shows that the uploaded data using SmallClient are larger than the actual data size. However, this difference becomes negligible for large data sets.

Table 6-2: Data Upload Size Results

Data Sets	No. of Blocks	Data Size (MB)	Data Upload Size (MB)
Primary Roads	2	77.1	128
Area Landmark	7	406	448

Tabulation Area	25	1560	1560
Area Hydrography	104	6460	6500
All Edges Combined (I)	260	16220	16250
Linear Hydrography	293	18270	18310
All Edges Combined (II)	363	23180	23190
All Edges Combined (III)	969	61900	61940
All Nodes	1500	96400	96460
Road Network	2141	137500	137570

The size of uploaded data for small size data sets significantly increases when these data are uploaded using SmallClient (i.e., for primary road data set, the size of uploaded data using HDFS is 77.1 MB, whereas this size is 128 MB when the data are uploaded using SmallClient). This observation occurs because the default block size of Hadoop for data uploading, which is 64 MB, and the size of primary road data set are slightly larger than the data in one block. SmallClient created two blocks for the fixed-sized data set of primary road of while null values are added in the end. Therefore, the size of uploaded data for the data set of primary roads increased. The data upload size results are presented in Figure 6-2.

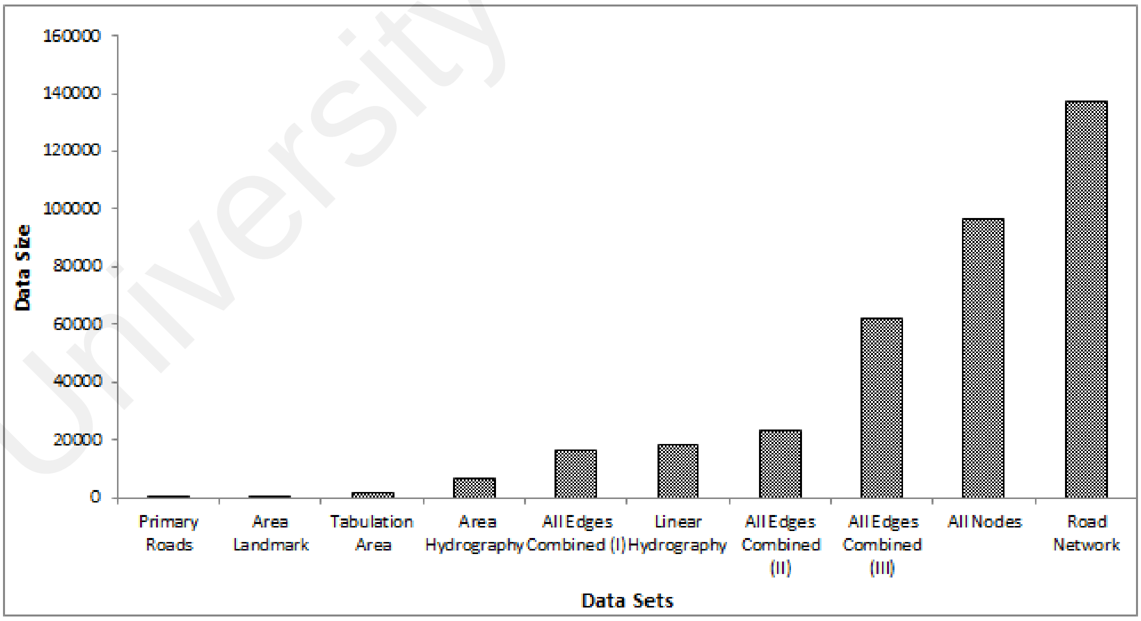


Figure 6-2: Data Size Results from Experiments

The data uploading results for both data uploading time and data upload size indicate that data uploading time and data upload size also increased with data set size. The time results for i-SmallClient are presented, which show that index creation parallel to data upload is more efficient. Five indexes are created with a slight increase in time.

- Indexing Results

The indexing experiment results are presented in terms of indexing time and index size in this section. Up to five indexes are created for evaluation. Table 6-3 shows the results for indexing time and index sizes.

Table 6-3 presents the results of indexing time of SmallClient to create up to five indexes. The results indicate that indexing time increases for large data sets. Indexing time also depends on the number of records in a data set. When a data set has more records, indexing time also increases. For instance, all edges combined (I) data set has more records than its adjacent data sets. Therefore, this data set takes more time to create one to five indexes using SmallClient.

The relationship between data set size and the number of records, which affects indexing time, is depicted in Table 6-3 for all nodes and road network data sets. All nodes data set is smaller. These data sets have more records. Thus, the indexing time for all node data sets is higher than the indexing time taken by road networks.

Indexing time is also affected when the number of indexes increases. Table 6-3 shows that indexing time is low to create one index, whereas more than one index takes more time in index creation. The index time experiment results are presented in Figure 6-3.

Table 6-3: Indexing Time Results for up to five Indexes

Data Sets	Data Set Size (MB)	No. of Records	Indexing Time (sec)				
			1	2	3	4	5
Primary Roads	77.1	13373	3.35	3.48	3.95	4.40	4.50
Area Landmark	406	121960	8.19	9.30	10.80	11.94	13.19
Tabulation Area	1560	33144	24.72	27.54	30.01	31.29	32.25
Area Hydrography	6460	2298808	141.65	155.66	164.74	154.34	176.59
All Edges Combined (I)	16220	19291957	380.53	420.30	504.42	617.57	795.93
Linear Hydrography	18270	5857442	288.26	418.14	444.34	464.16	492.07
All Edges Combined (II)	23180	70000000	785.52	1121.05	1488.22	1988.89	2335.09
All Edges Combined (III)	61900	72700000	1694.89	1772.45	1829.82	1890.37	1991.86
All Nodes	96400	2700000000	20572.89	20534.76	20725.30	20899.25	20922.36
Road Network	137500	717000000	17350.42	18210.49	18991.21	19390.75	19948.28

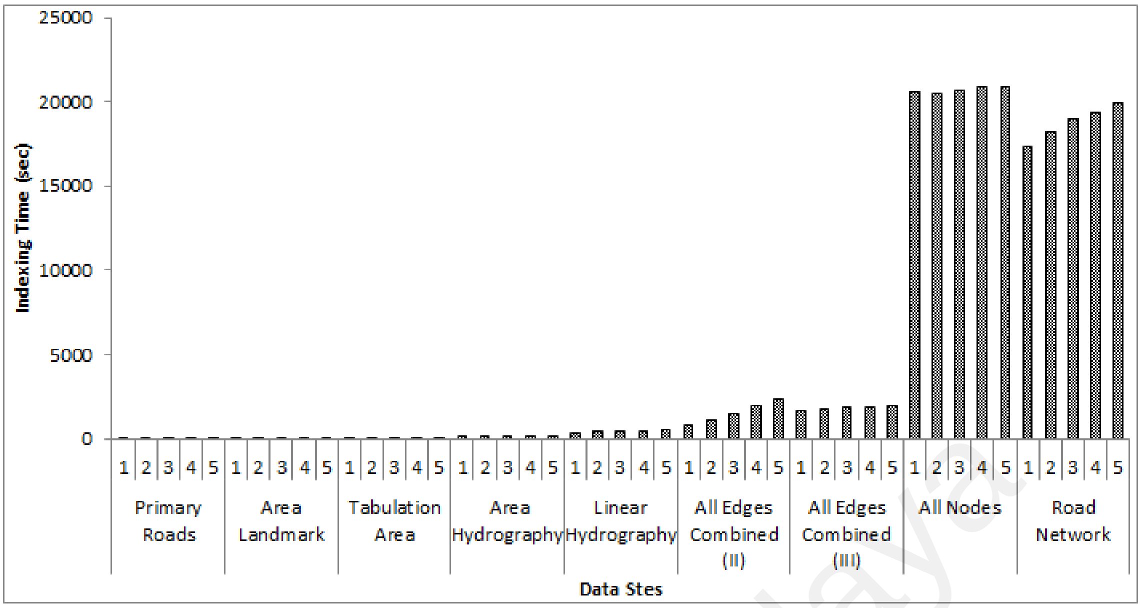


Figure 6-3: Indexing Time Results from Experiments for up to five indexes

Figure 6-3 shows the indexing time experiment results and indicates that indexing time increases with data set size. The slight elevation in indexing time with the increase in the number of index attributes shows that SmallClient facilitates the creation of more indexes for a data set, which is advantageous to improve search performance. Therefore, SmallClient offers fast index creation despite the increased number of indexes for index creation.

The data for index size evaluation are obtained. The index size results for up to five indexes are presented in Table 6-4. The table shows the index size results when up to five indexes are created using SmallClient. Index size depends on data set size and the number of records in a data set. Therefore, index size increases with data set size for SmallClient. Index size increases with the number of indexes. This increase in index size is higher for small data sets after three indexes.

Table 6-4: Index Size Results for up to five Indexes

Data Sets	Data Set Size (MB)	No. of Records	Index Size (MB)				
			1	2	3	4	5
Primary Roads	77.1	13373	0.13	0.34	0.85	1.37	1.90
Area Landmark	406	121960	0.16	1.35	3.91	8.41	14.94
Tabulation Area	1560	33144	0.34	1.80	3.44	5.11	6.89
Area Hydrography	6460	2298808	0.07	5.12	24.24	43.37	62.60
All Edges Combined (I)	16220	19291957	173.22	315.98	754.62	1460.54	2397.80
Linear Hydrography	18270	5857442	0.21	16.16	156.24	296.33	340.87
All Edges Combined (II)	23180	70000000	2197.11	5078.66	8242.46	11529.61	15659.79
All Edges Combined (III)	61900	72700000	554.79	1150.99	2692.31	4986.74	8532.78
All Nodes	96400	2700000000	1062.34	2130.24	3521.57	4827.45	7321.65
Road Network	137500	717000000	2645.25	4837.94	7346.86	9826.15	20134.67

More indexes for a data set occupy more space. Thus, index size increases. The index size results for up to five indexes are shown in Figure 6-4. The size of bars indicates the size of the index. Figure 6-4 shows that the index size is low for small data sets. The index size also increases with the number of indexes for a data set.

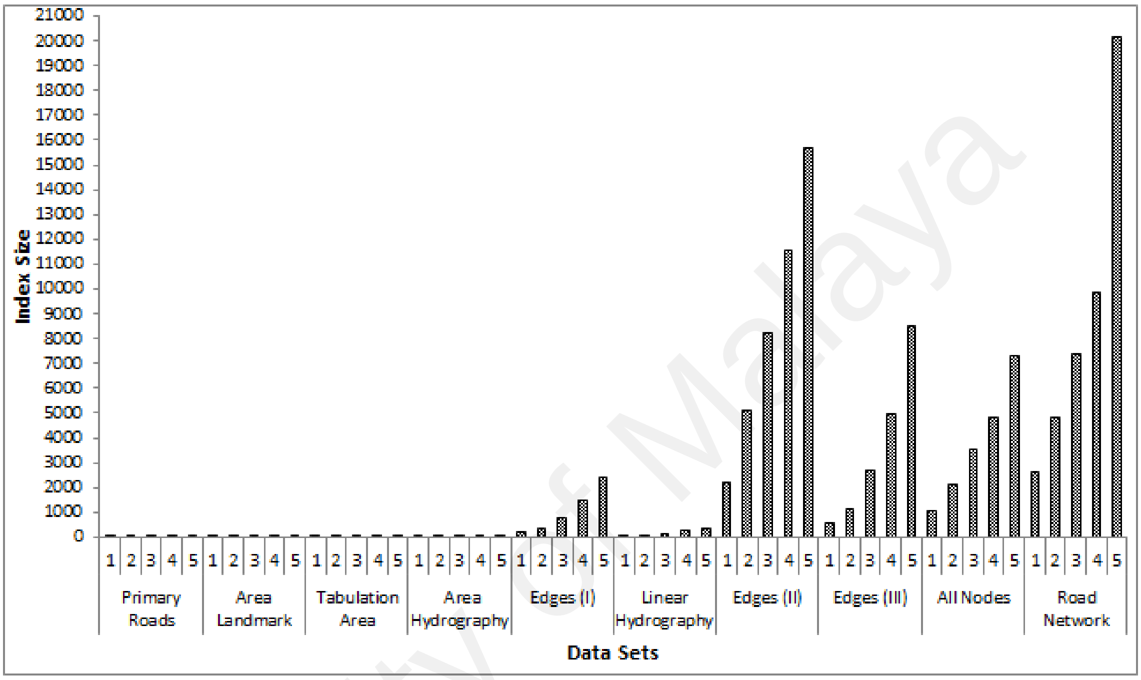


Figure 6-4: Index Size Resuts for five indexes

- Search Performance

The query execution time and search performance results are presented in this section. Table 6-5 indicates the results of query execution and search performance. The query execution time for a specific attribute (I) and for the entire record (*) are included using full scan and indexed search. The results of query execution time indicate that time increases with data set size.

Another factor that affects query execution time is the index size of a data set. When the index size is large, i.e., the index size for all edges combined (I) and all edges combined (II) data sets, the query execution time for these data sets is also high.

Table 6-5: Query Execution Time and Search Performance Results

Data Sets	Full Scan		Query Execution Time (sec)		Search Performance
	I	*	*I*	*	
Primary Roads	15.55	15.45	0.66	0.08	95.76
Area Landmark	21.21	20.04	1.50	1.46	92.93
Tabulation Area	21.84	38.97	1.16	1.25	94.68
Area Hydrography	56.31	45.54	2.90	3.68	94.85
All Edges Combined (I)	102.25	102.25	40.42	41.12	60.47
Linear Hydrography	183.00	152.61	3.11	3.04	97.96
All Edges Combined (II)	175.77	174.52	61.51	65.45	65.00
All Edges Combined (III)	280.26	275.45	3.92	3.88	98.60
All Nodes	410.34	410.21	7.21	7.24	98.24
Road Network	546.80	545.54	10.56	10.43	98.07

The query execution time results for specific attribute retrieval are presented in Figure 6-5. Figure 6-5 also indicates that the query execution time increases with the size of data sets in all cases. For instance, query execution time for smallest size data set i.e. Primary Roads data set is least in Figure 6-5. Query execution time for larger size data sets is higher than Primary Roads data set. However, the query execution time for all edges combined (I) and all edges combined (II) data sets is high because of the large indexes.

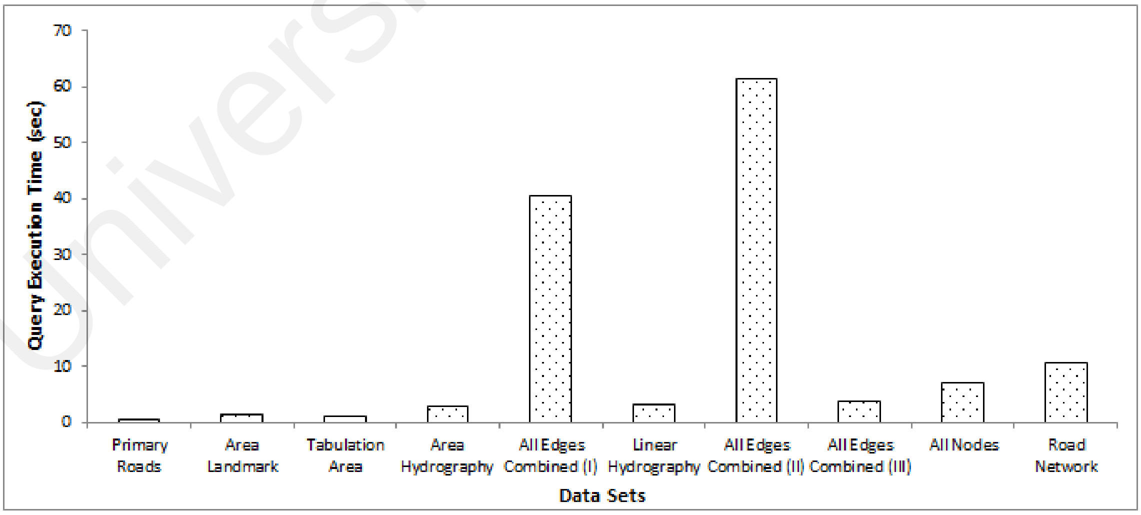


Figure 6-5: Query Execution Time Results from Experiments

- Index Hit Ratio

The index ratio results are presented in Table 6-6 and Figure 6-6. Predictor logic was adopted to decide whether new indexes are created or existing indexes are deleted, as described in Chapter 4. Predictor logic observes 10 time slots of the query log to make decisions. Thus, IHR increases. The results of 20 time slots, which were obtained from the query log, are presented in Table 6-6. Total queries indicate the number of queries, which are executed in a time slot, whereas hit queries represent the number of queries that utilized indexes.

The first 10 time slots (i.e., T1 to T10) consist of queries that are executed when static indexes were created. Each attribute from a data set schema is observed in each of the 10 time slots by predictor logic, and indexes are adaptively updated. Ten time slots (i.e., T11 to T20) are taken for adaptive indexes, which are updated using predictor logic. The IHR in Table 6.8 fluctuates with changing query workload in each time slot. However, after updating the indexes, the overall IHR increases to more than 0.54 for the T11 to T16 time slots.

Table 6-6: Query Log observations and Index Hit Ratio Results before and after Predictor Logic

	Time Slots	Total Queries	Hit Queries	Index Hit Ratio
Static Indexes	T1	28	23	0.82
	T2	22	15	0.68
	T3	21	6	0.29
	T4	24	4	0.17
	T5	30	5	0.17
	T6	26	18	0.69
	T7	24	12	0.50
	T8	22	9	0.41
	T9	30	8	0.27
	T10	27	5	0.19
Adaptive Indexes	T11	35	23	0.66
	T12	32	19	0.59
	T13	28	18	0.64
	T14	33	25	0.76
	T15	31	20	0.65
	T16	24	13	0.54
	T17	21	4	0.19
	T18	24	7	0.29
	T19	30	7	0.23

	T20	26	6	0.23

Figure 6-6 depicts the IHR for static and adaptive indexes when predictor logic is applied. The bars from T11 to T16 in the adaptive indexes show that the IHR improves when the indexes are updated according to the changing query workload. The bars from T17 to T20 are very short, which indicates that the index must be updated again.

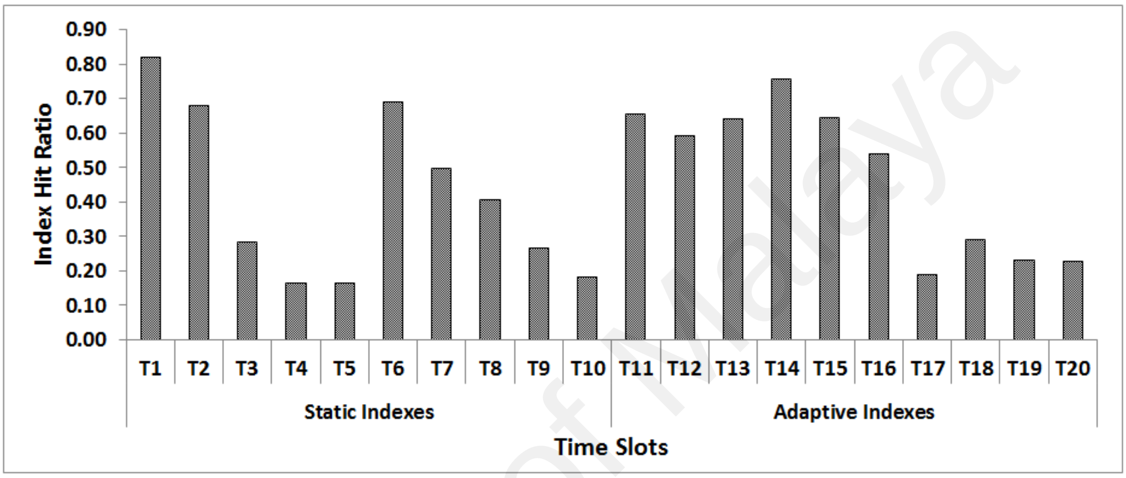


Figure 6-6: Index Hit Ratio Results for Static and Adaptive Indexes

6.1.2 Mathematical Modeling Results

The validation of the results from the mathematical modeling of data uploading, indexing, and query execution modules of SmallClient is presented in this section. The transitions in the proposed mathematical model are tuned with the timed values and collected data for uploading time, indexing time, and query execution time.

- Data Upload Time

The data upload time results are collected by using the mathematical model designed for SmallClient. Data uploading time integrates the time taken by reading each record and storing these blocks in a distributed file system, as described in Chapter 5 (see Equation 5-21). Data set size also affects block creation and data uploading time. Therefore, 0.065

was set as the data set size factor, which was denoted by c, 0.00006 for $\mu_{read\ records}$ and 3.5 for $\mu_{store\ block}$. The data upload time results are presented in Figure 6-7.

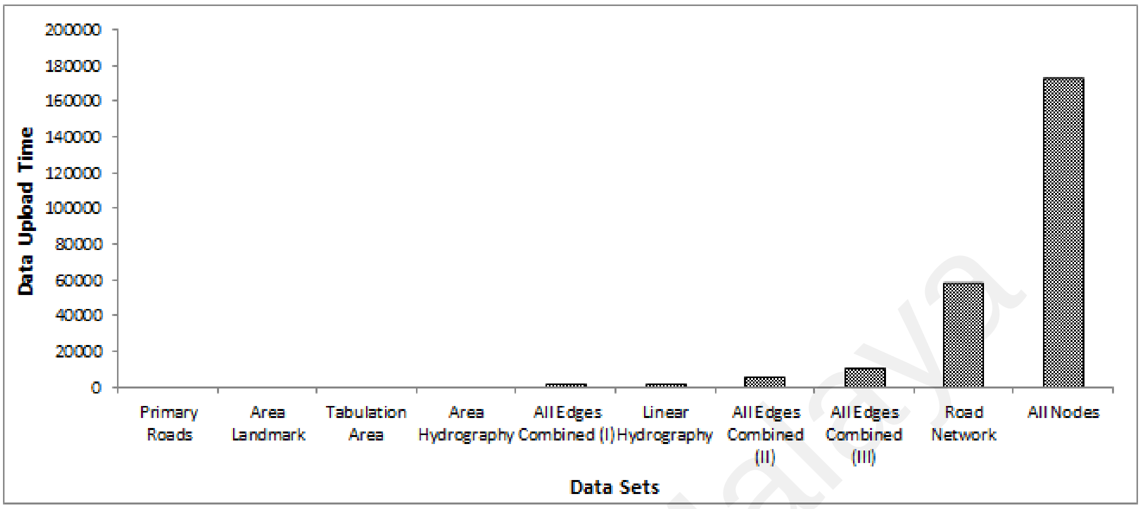


Figure 6-7: Data Upload Time Resuts using Mathematical Model

- Indexing Time

The indexing time results are obtained using the mathematical model when the data are already uploaded to the file system. Indexing time is composed of time periods to obtain <key, value> pairs from each record and store an index for a block to the file system (see Equation 5-14). The values for $\mu_{get<key,val>}$ and $\mu_{store\ index}$ are set to 0.000006 and 1, respectively. The indexing time results obtained from the mathematical model are presented in Figure 6-8.

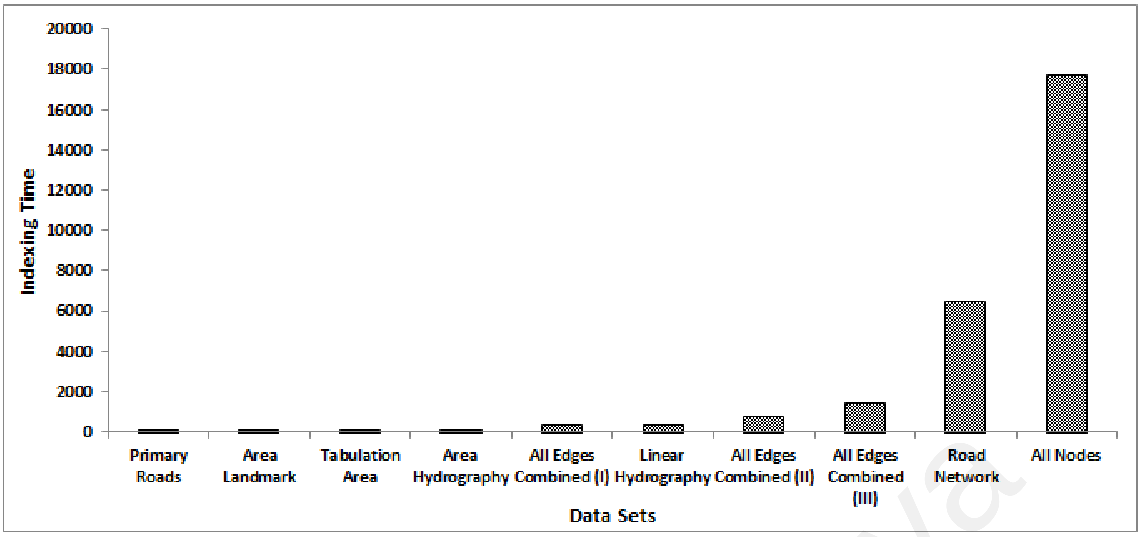


Figure 6-8: Indexing Time Resuts using Mathematical Model

- Search Performance

The query execution time results, which are obtained from the mathematical model, are used to validate the search performance. The values for $\mu_{get\ index}$, $\mu_{search\ key}$, and $\mu_{fetch\ data}$ are set to 0.004, 0.000000002, and 0.0015, respectively, to collect the query execution time results from the CPN model using Equation 5-15. The collected results are presented in Figure 6-9.

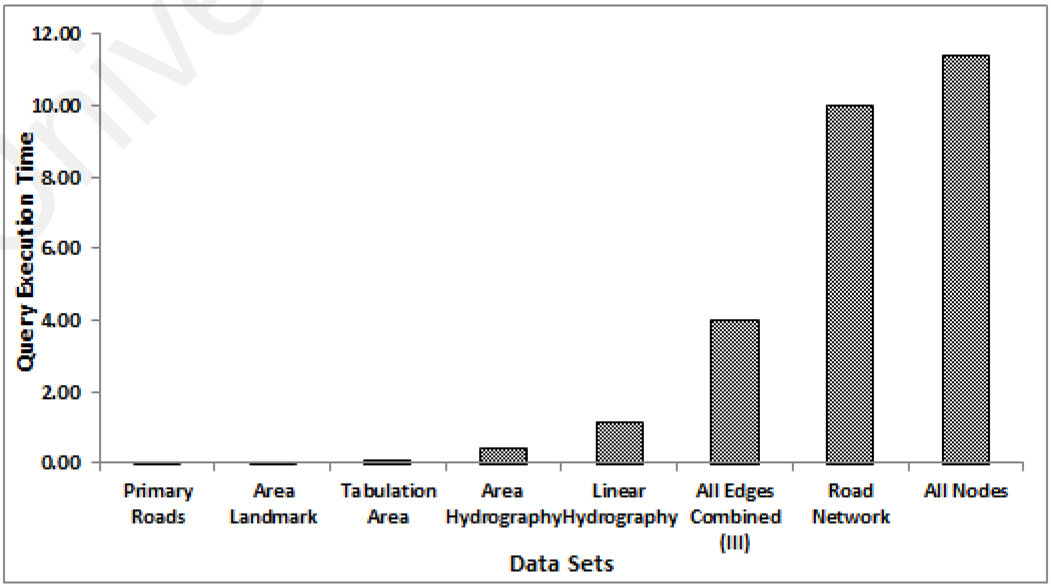


Figure 6-9: Query Execution Time Resuts using Mathematical Model

6.2 Verification

The verification of the performance results of SmallClient for all modules, such as data upload overhead, indexing overhead, and search performance, are presented in this section. Benchmarking and mathematical modeling are employed to verify the correctness of the results obtained by executing the experiments.

6.2.1 Benchmarking

The SmallClient results are verified by using benchmarking. Hive and Lucene library are used for benchmarking, which return the results for full scan and indexed search environments, respectively.

As shown in Chapter 5 (see Section 5.1), the test bed is designed and Apache Hadoop four-node cluster is created. Hadoop offers highly efficient distributed task execution and data management by using MapReduce and HDFS. Apache Hive warehouse is configured to execute SQL-Like queries, which leverage MapReduce instead of indexes, to efficiently execute full-scan data search operation in a distributed parallel manner. The proposed indexing framework outperforms Apache Hive in query execution with a minimum overhead caused by indexing.

Apache Lucene is used as a benchmark for verification, which offers an indexing library to achieve high search and data retrieval performance on big data. SmallClient performs better than Apache Lucene in query execution and reduced indexing overhead.

- Data Upload Overhead

The results obtained from the SmallClient experiments for data uploading time and data upload size are compared with data uploading time and data size of HDFS. As defined in Chapter 5, data upload overhead is the percentage of increased activity to upload data using the block creation module of SmallClient. Therefore, the level of size and time increase by SmallClient is presented as overhead.

The data upload time overhead and data upload size overhead verification are first presented. Figure 6-10 shows the data upload time taken by HDFS and SmallClient. The figure also indicates the data upload time results for i-SmallClient. The bars that present the data upload time for i-SmallClient are the largest, whereas the bars that indicate the data upload time for HDFS are the smallest, which indicates that the time taken in data uploading by i-SmallClient is the highest. However, the data uploading time for SmallClient is also high.

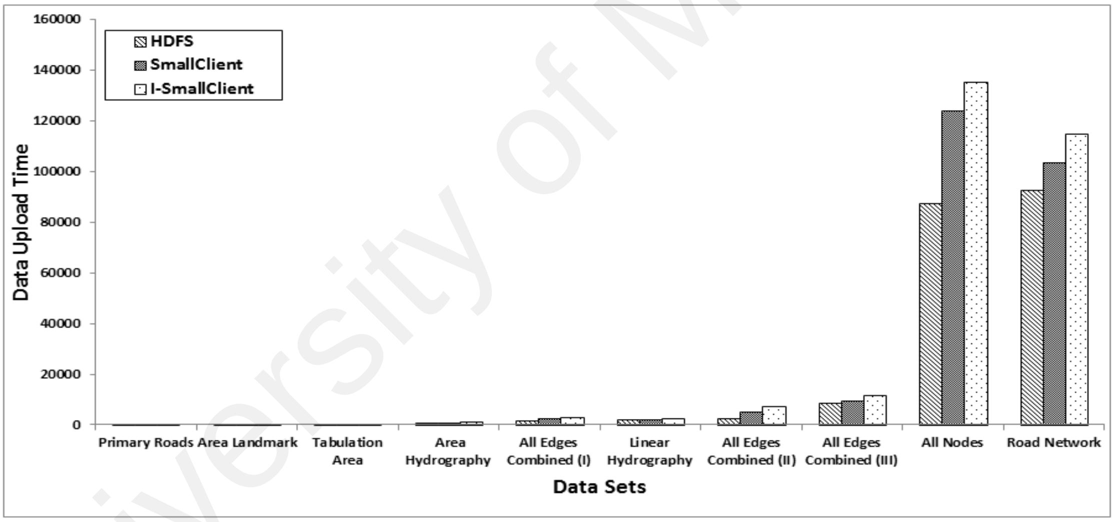


Figure 6-10: Benchmarking on Data Uploading Time

Figure 6-10 indicates that SmallClient takes more time in data uploading than HDFS. This time consumption increases with data set size. SmallClient reads the entire data set line by line, pushes them into blocks, and uploads each block individually. The proposed data uploading mechanism is explained in Chapter 4. Data uploading is considerably time-consuming with SmallClient because of the process involved in block creation activities. The data uploading time by i-SmallClient includes additional time to create indexes. Therefore, this time consumption is the highest in all platforms.

The data uploading overhead caused by SmallClient depends on two factors: data set size and number of records. When a data set is larger and the number of records is less, the data uploading time overhead becomes very low. For instance, the size of the tabulation area data set is larger than the area landmark data set, whereas the number of records is less. Therefore, the observed data uploading time overhead of SmallClient for the tabulation area data set is lower than that of the area landmark data set.

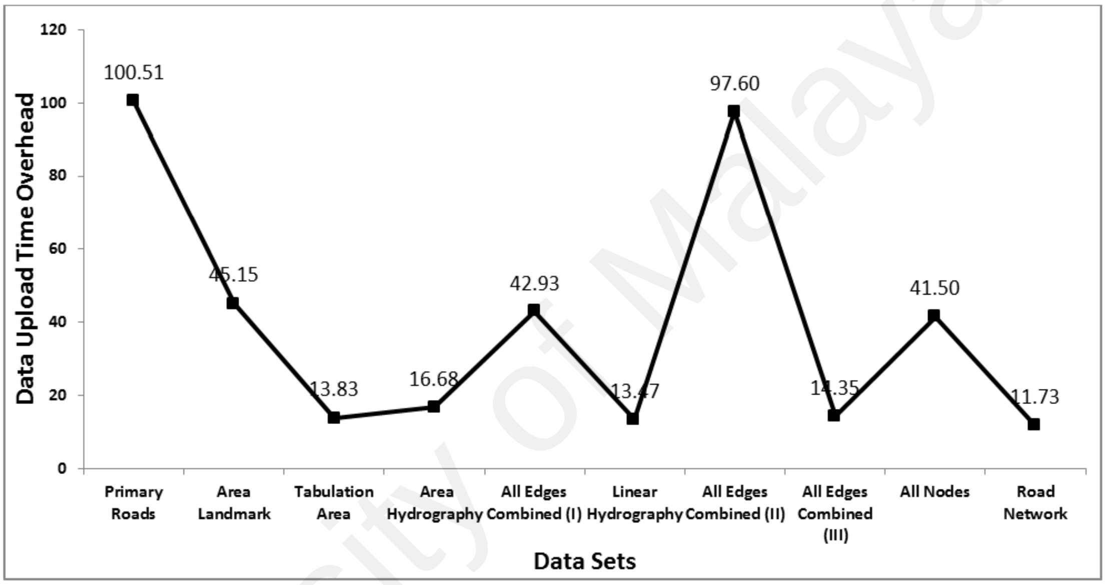


Figure 6-11: Data Upload Time Overhead

The data uploading time overhead results by SmallClient are presented as a line graph in Figure 6-11. The Primary Roads data set, which is the smallest data set, has the maximum data upload time overhead. The line in Figure 6-11 gradually decreases for larger data sets even when the number of records, i.e., area landmark and tabulation area data sets, gradually increase. However, data uploading overhead is high for data sets with larger data sets, which are composed of an extensive number of records, i.e., All Edges Combined (I), All Edges Combined (II), and All Nodes data sets.

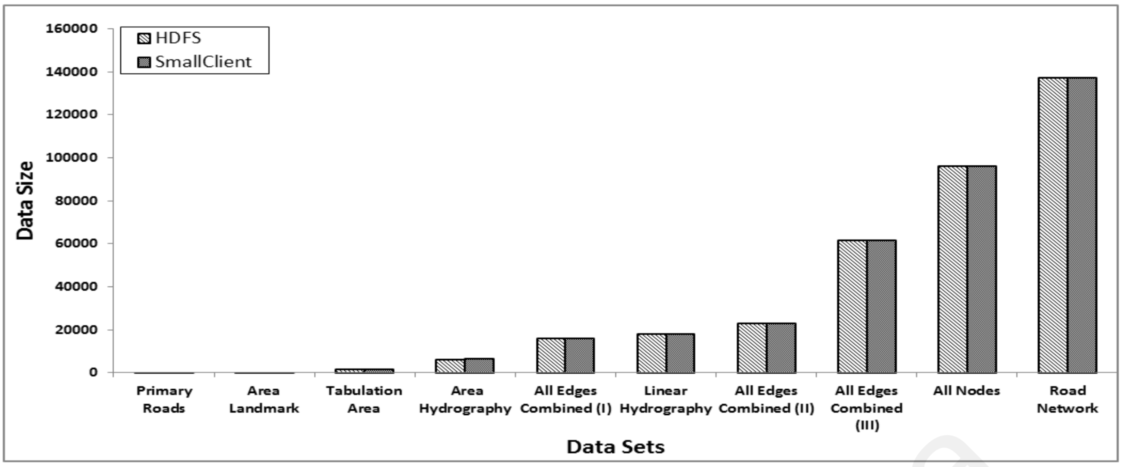


Figure 6-12: Benchmarking on Data Size

The data upload size results are also verified. Figure 6-12 presents the data upload size results obtained from HDFS and SmallClient. The data upload sizes using HDFS and SmallClient are almost the same. The downward diagonal bars indicate the size of uploaded data using HDFS, whereas the closed dotted bars show the data size results for SmallClient. Figure 6-12 depicts that the difference between the sizes of uploaded data using HDFS and SmallClient is negligible.

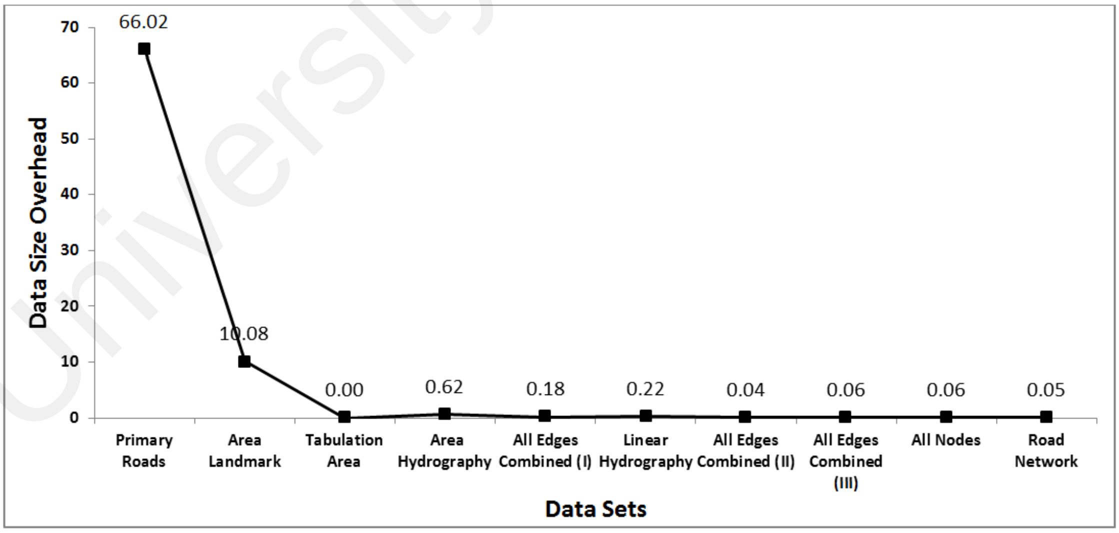


Figure 6-13: Data Upload Size Overhead

The data upload size overhead is presented in Figure 6-13. The points on the line that start in the X-axis for large data sets show that the data upload size overhead decreases and approaches zero for large data sets. The results indicate that data size overhead for primary roads is high at 66.02%. However, for large-volume data sets, the size of uploaded data using SmallClient is almost similar to the size of uploaded data using HDFS. This result verifies that the SmallClient data uploading module is efficient for large data sets and the size overhead of uploaded data is less than 1%.

The data uploading overhead for both data uploading time and data upload size presented in this section show that SmallClient affects the size and uploading time of data. The overhead of the data upload size using SmallClient is very low for large data sets. However, the data uploading time depends on the size of a data set and the number of records. Therefore, the data uploading time varies with the size and number of records.

- Indexing Overhead

The indexing results of the proposed SmallClient are compared with the Lucene indexes, which are created using well-known Apache Lucene indexing library for big data. Apache Lucene, which is a widely adopted library for big data indexing and other search operations, creates indexes while loading data sets in the main memory. As explained in Chapter 3, the Lucene results are out-of-memory error when the data set is larger than the available main memory (see Table 3-2). Therefore, Lucene fails to create indexes for All Edges Combined (I), All Edges Combined (II), All Edges Combined (III), and All Nodes and Road Network data sets. However, SmallClient outperforms Lucene and results in low index creation overhead.

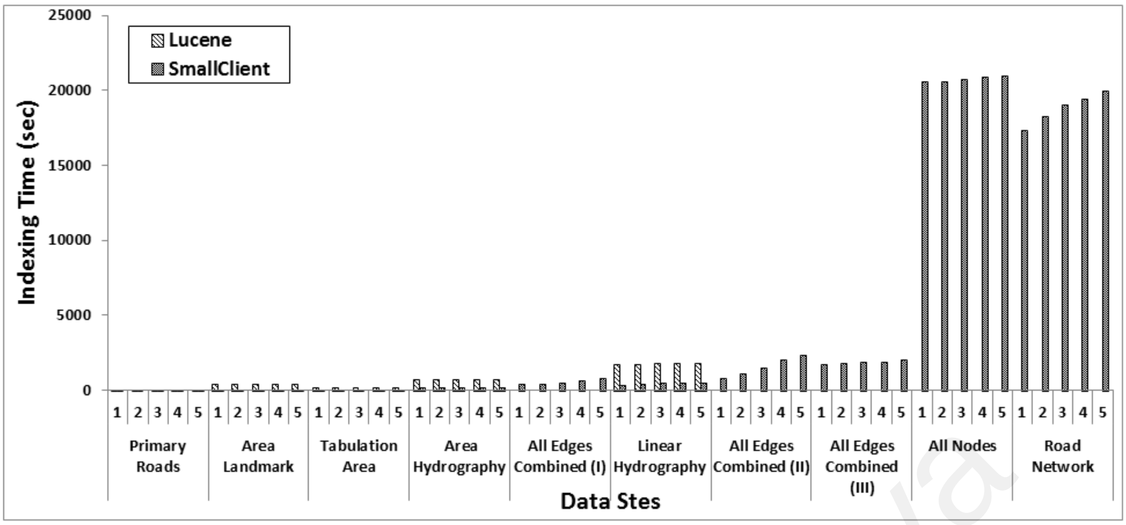


Figure 6-14: Indexing Time Comparison for up to five indexes when created using Lucene vs SmallClient

The index creation time and index size overhead of SmallClient are presented and compared with the results of the Apache Lucene indexes. Figure 6-14 shows the index creation time results for Lucene and SmallClient. The figure indicates that the indexing time of SmallClient is lower than that of Lucene. The indexing time results are verified for up to five indexes, which indicate that SmallClient performed better.

Figure 6-14 shows that Lucene could not create indexes for large data sets and returned out of main memory error. However, SmallClient solves this problem by considering small manageable data blocks instead of the entire data set in index creation. The results prove that Lucence takes more time to create indexes regardless of data set size or other features of a data set.

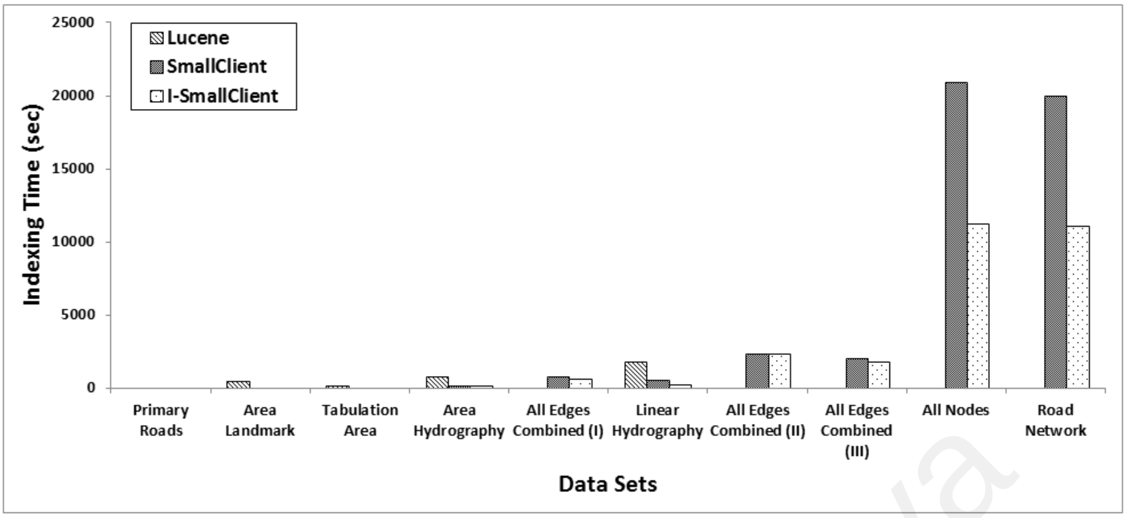


Figure 6-15: Indexing Time Results using Lucene, SmallCLient and I-SmallClient

The index time results when five indexes are created using Lucene, SmallClient, and i-SmallClient are presented in Figure 6-15. The downward diagonal bars show the indexing time for Lucene, the near dotted bars denote the indexing time taken by SmallClient, and the dotted bars indicate the indexing time results for i-SmallClient. The time taken by Lucene to create five indexes is the highest for each data set. The indexing time by i-SmallClient is the least among all the indexing methods for all data sets. This result indicates that creating indexes in parallel to data uploading is more beneficial to users. Moreover, creating indexes any time with changing query workload using SmallClient is still better than utilizing Lucene.

The indexing time overhead shows additional delay because of indexing to start query execution. The indexing time overhead SmallClient is lower than that of Lucene. The indexing time overhead results are presented in Figure 6-16.

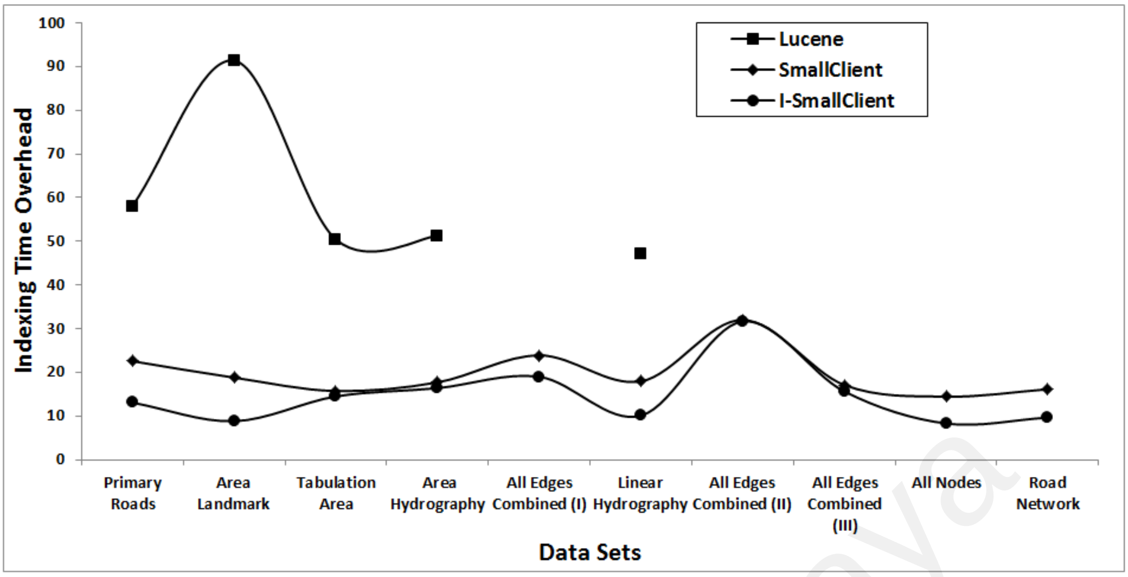


Figure 6-16: Indexing Time Overhead Results

Figure 6-16 shows that SmallClient has lower indexing time overhead than that of Lucene, which proves that SmallClient minimizes indexing time overhead. The lowest indexing time overhead of i-SmallClient indicates that indexes should be created parallel to data uploading.

Queries experience more than 40% delay when these queries are executed after Lucene index creation, whereas this delay is reduced to 14%–32% when SmallClient indexing is applied. However, i-SmallClient further reduced indexing overhead to 6%–32%. Figure 6-16 shows the improved results of SmallClient in indexing time overhead.

The index size performance of SmallClient improves for large data sets. Its index size in creating up to five indexes is lower than Lucene indexes. The index size results of SmallClient and Lucene are presented in Figure 6-17.

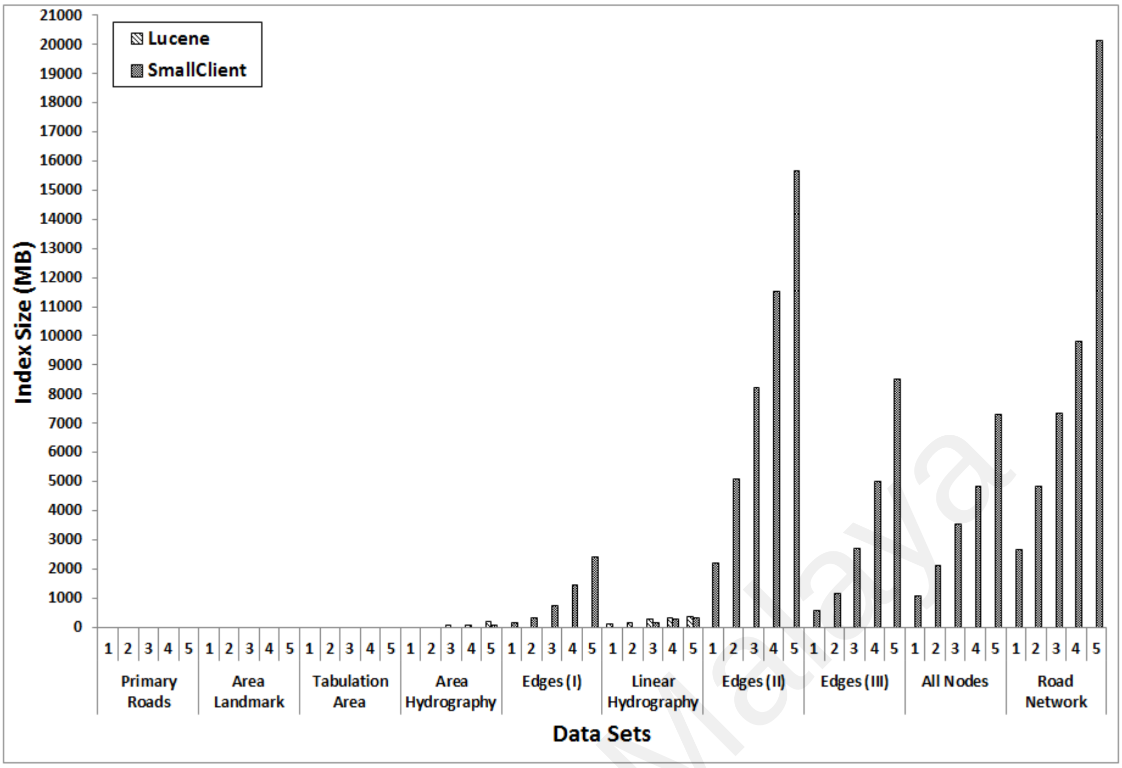


Figure 6-17: Index Size Results using Lucene and SmallCLient

Figure 6-17 presents a comparison of the index size results of SmallClient and Lucene. The downward diagonal bars show the index size results for Lucene, and the near dotted bars present the index size results for SmallClient. Index size is proven to be lower with SmallClient than with Lucene for large data sets, i.e., area hydrography and linear hydrography data sets. SmallClient performs well, and its indexes require lesser space for large data sets than that of Lucene indexes.

The index size overhead for five indexes over data size using Lucene and SmallClient is presented in Figure 6-18. The results show that the index size overhead using SmallClient decreases for large data sets. SmallClient exhibits better index size overhead performance when the indexes are created for large data sets and the index size overhead is up to 41%.

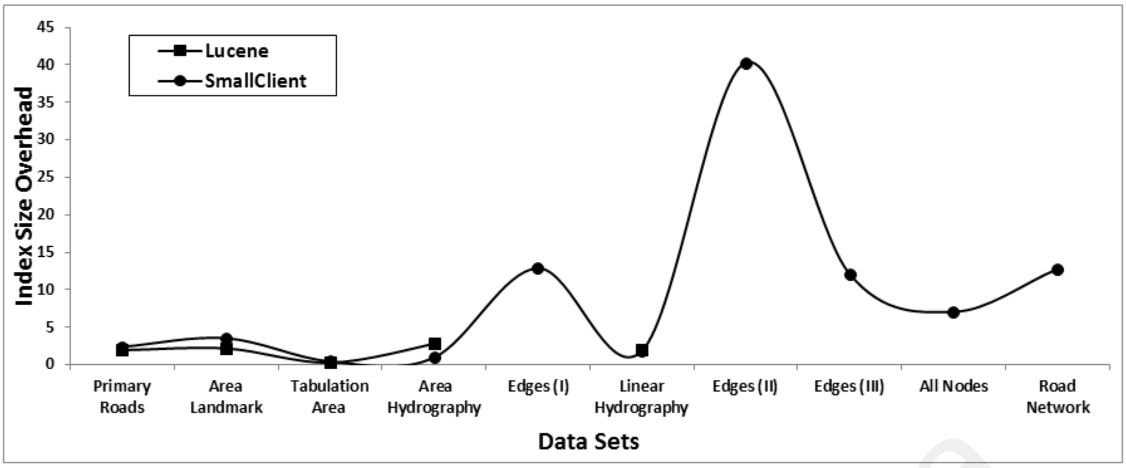


Figure 6-18: Index Size Overhead Results

The results in Figure 6-18 show that the index size overhead using SmallClient is higher than that of Lucene for small data sets, such as primary roads and area landmarks. However, the index size overhead using SmallClient for other large data sets decreases, which indicates that SmallClient exhibits better index size performance for large data sets.

- Search Performance

Apache Lucene indexes are used for indexed search comparison, and Hive is used for full-scan comparison. Up to five indexes are generated instead of creating indexes on all attributes of a data set.

Lucene does not allow access to the entire record through query. However, only these attributes can be retrieved using Lucene indexes for which indexes are available. Therefore, the query execution time results for the entire record retrieval are not available using Lucene. Moreover, query execution time results are not available for data sets where Lucene failed to create indexes.

SmallClient outperforms Lucene and overcomes this limitation. Therefore, the entire records can be accessed and retrieved from a data set composed of indexed and/or non-indexed attributes by using SmallClient.

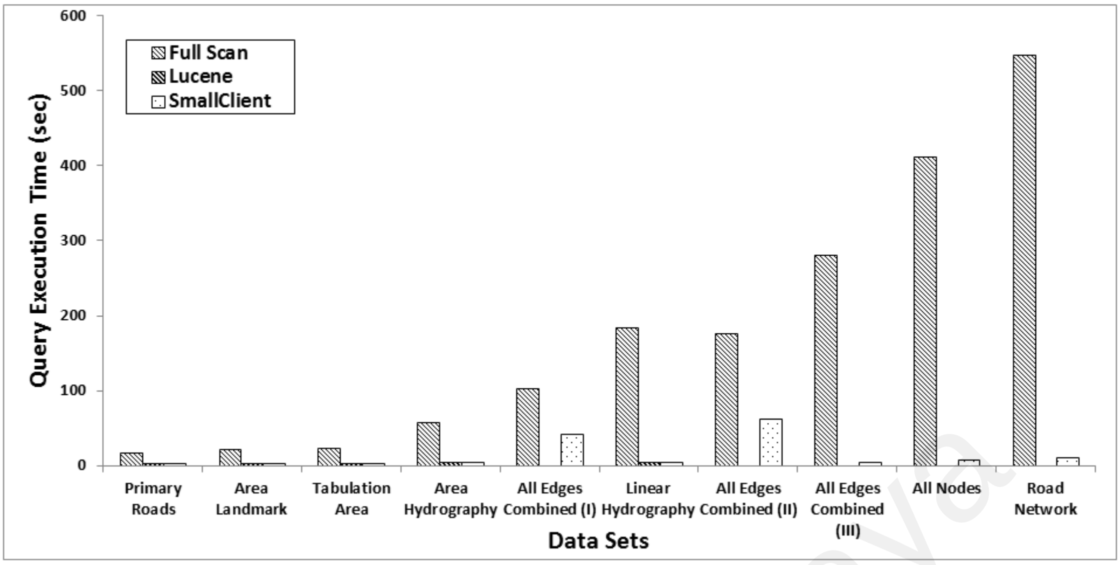


Figure 6-19: Query Execution Time Comparison between full scan, Lucene and SmallClient

Figure 6-19 presents the query execution time results by using full scan, Lucene indexes, and SmallClient indexes. The results show that the speed query execution improves with indexes. Full scan requires a longer time than indexes. The query execution performance of SmallClient is better than that of Lucene indexes. Figure 6-19 also indicates that query execution time increases at a higher rate with data set size than that of Lucene and SmallClient Indexes.

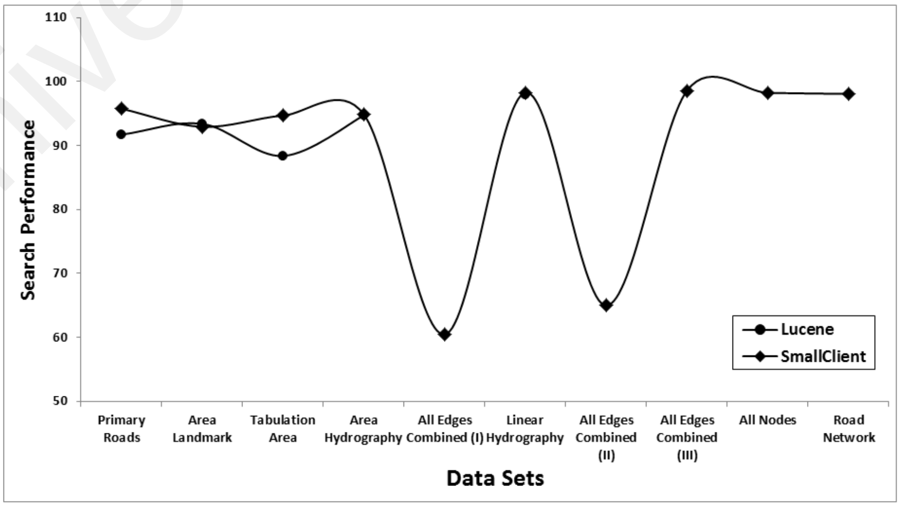


Figure 6-20: Search Performance results using SmallClient and Lucene

The search performance results using Lucene and SmallClient are presented in Figure 6-20. The search performance results indicate that search performance improves using

SmallClient. The search performance of SmallClient is more than 92% for all data sets except for All Edges Combined (I) and All Edges Combined (II) data sets because of the large indexes of these data sets.

6.2.2 Mathematical Modeling

The experimental results of SmallClient are verified by using mathematical modeling. The experimental data for data upload time, indexing time, and query execution time are compared with the data collected using the mathematical model to verify the correctness of these results.

Petri nets mathematical modeling language was used to develop the mathematical model for the proposed SmallClient framework by using CPN tools. CPN tools leverage their built-in discrete-event modeling language and Standard ML. The data are obtained from the mathematical model, and the comparison of the results from the experiment and mathematical model verifies the performance of the proposed framework.

- Data Upload Overhead

The data upload time results obtained for the data upload overhead from the experiments are compared with results for the same parameter from the mathematical model. The comparison is described in Figure 6-21. The dotted bars show the data upload time results from the experiment, whereas the line indicates the validation results, which are obtained using the mathematical model.

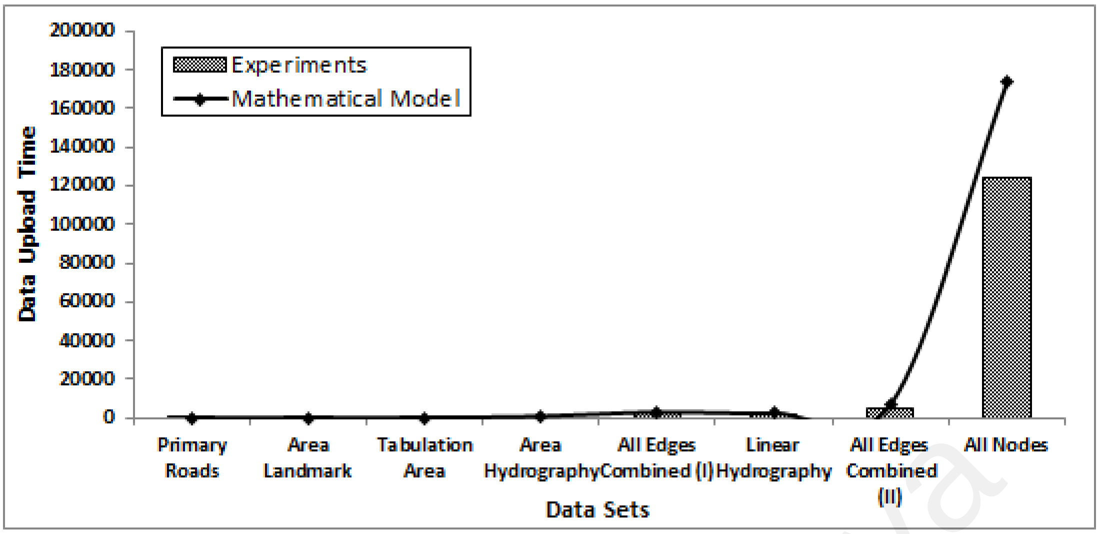


Figure 6-21: Data Upload Time Mathematical Verification

Figure 6-21 shows that both the experiments and mathematical model produced almost the same results for all data sets except the All Nodes data set, which verifies the accuracy of data upload time using SmallClient in the performance evaluation. The points of lines touching the top of the bar verify the results.

- Indexing Overhead

The indexing time taken by SmallClient is considered in verifying the indexing overhead results when SmallClient is executed using the experiments and the mathematical model. The indexing time comparison results are presented in Figure 6-22.

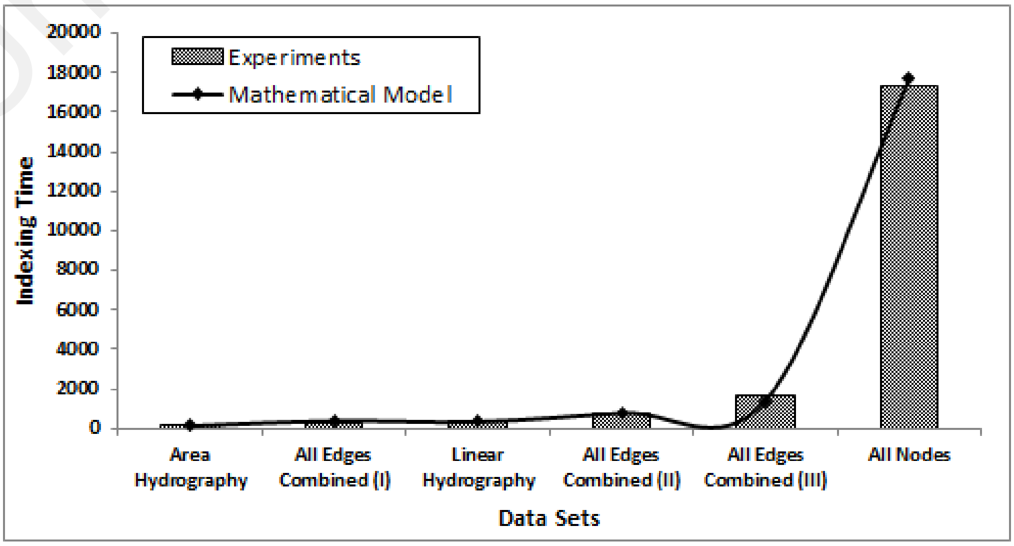


Figure 6-22: Indexing Time Mathematical Verification

The bars in Figure 6-22 indicate the indexing time for each data set obtained from the experimental model, whereas the line shows the results from the mathematical model. The points in the line slightly touch the top of the relevant bars for each data set, which shows that the results exhibit minor differences and verifies the results for indexing time.

- Search Performance

The query execution time results are selected to verify the search performance results from the experiments and the mathematical model. Figure 6-23 presents the verification results for query execution time.

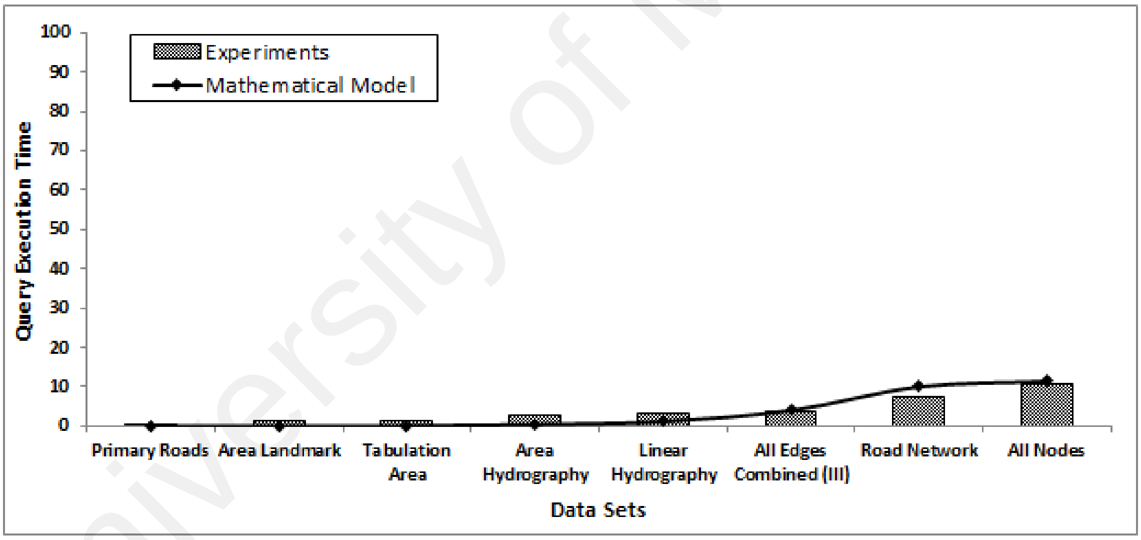


Figure 6-23:s Query Execution Time Mathematical Verification

The bars in Figure 6-23 indicate the experiment results for query execution, whereas the line validates these results using the mathematical model. The slight distance between each line point and the relevant bar top verifies the results for query execution time.

6.3 Conclusion

This chapter presents the evaluation results of the performance of the proposed SmallClient. The data for data upload overhead, indexing overhead, search performance, and index hit ratio are obtained and validated. Varying size data sets are used in the experiment, which shows that SmallClient can create indexes for any data set size. However, the performance of SmallClient is better for large data sets. The data upload overhead from SmallClient reduces for large data sets. Indexing overhead in terms of index creation time and index size is also reduced. Indexing overhead using i-SmallClient is lower than that using SmallClient.

The verification of search performance results using benchmarking indicates that a remarkable difference exists between query execution time by full scan (using Hadoop) and SmallClient when similar queries are submitted. SmallClient performs better than Apache Lucene indexing, which is considered as a high-performance information retrieval and search software.

The SmallClient framework is effective in index creation, whereas Apache Lucene cannot handle large data sets. The Apache Lucene program returns out-of-memory error for up to 20 GB data sets on the test bed, whereas the proposed indexing mechanism, SmallClient, efficiently handles the workload and successfully accomplishes the index creation task.

The verification of SmallClient results by using mathematical modeling also proves the reliability of SmallClient results. The results for data uploading, indexing, and query execution time are obtained using the mathematical model of SmallClient. The comparison of the experimental and mathematical modeling results is presented. The comparison verified the results and proved the better performance of the proposed indexing framework.

CHAPTER 7: CONCLUSION AND FUTURE WORK

This chapter concludes the research work and highlights future research directions. The objectives of this thesis are achieved, as described in this chapter. The contributions are highlighted.

The rest of the chapter is organized as follows: Section 7.1 presents the examination of the achieved objectives. Section 7.2 highlights the contributions of this study, and Section 7.3 presents the limitations and future work.

7.1 Fulfillment of Aim and Objectives

The achieved objectives of this thesis are examined in this section. As described in Chapter 1, the aim of this study was to expedite the data retrieval process against search queries over big data by proposing a novel indexing framework that introduces both static and adaptive indexing with minimized indexing overhead and improves data search performance and index hit ratio. This section describes how the objectives are fulfilled in this study.

7.1.1 Investigating the capability of existing indexing techniques to address the challenges of big data to establish potential research problem

The indexing techniques for traditional data, which are presented in credible publications, are reviewed to fulfill this objective. The indexing requirements for big data are identified as the “six Vs” (volume, velocity, variety, veracity, variability, and value) and complexity. The performance of contemporary indexing implementation on big data under the clustered and non-clustered categories is reviewed (see Chapter 2). The problem of this thesis is established by analyzing the performance of the Apache Lucene indexing library, which implements the non-clustered indexing approach on big data (see Chapter 3).

The clustered indexing approach allows a number of indexes up to available data replicas. To create more indexes, clustered indexing requires more replicas of an entire data set or several blocks, which increase the storage overhead for big data. Lucene indexes are non-clustered. However, indexing overhead in terms of index size and indexing time is moderately high.

7.1.2 Designing and implementing an indexing framework using non-clustered indexing structure incorporated with predictor function for adaptive index updating, which ensures the following:

- Minimized indexing overhead in terms of index creation or updating time and the space consumed by indexes (index size) for large-volume data;
- Reduced data retrieval time with faster query execution and data search performance; and
- Maximum index hit ratio by predicting the future workload of incoming search queries

This objective is achieved by proposing a novel indexing framework for big data called SmallClient, which implements the non-clustered indexing approach and allows indexes to be created either statically at the time of data uploading or any time when users realize changing query workload and invoke adaptive index updating. SmallClient also introduces predictor function, which automatically predicts incoming query workload and updates available indexes (see Chapter 4).

SmallClient is an indexing solution for big data with minimized indexing overhead and improved query execution and data search performance. The adaptability of SmallClient to changing query workload functionality keeps available indexes up-to-date. Thus, the maximum index hit ratio is achieved.

7.1.3 Evaluating the effectiveness of proposed indexing framework with respect to overhead resulted by static and adaptive indexing, query execution and data retrieval time, and index hit ratio

The test bed is designed to validate the performance of SmallClient. Varying data set sizes are used to execute the experiment. Data for data upload overhead, indexing overhead, search performance, and index hit ratio are collected. The obtained data from the experiment on varying size data sets proves the adaptability of SmallClient on big data.

The results proved that SmallClient can fulfil the indexing and data search requirements of data sets of any size. However, the performance of SmallClient is better for large data sets. The data upload overhead results from SmallClient reduces for large data sets. Indexing overhead in terms of index creation time and index size is also reduced. The indexing overhead using i-SmallClient is lower than that using SmallClient.

7.1.4 Verifying the results of proposed indexing framework using benchmarking and mathematical modeling

The results of experiments are verified by using benchmarking and mathematical modeling. The full scan of Apache Hive and indexed search of Apache Lucene are used for benchmarking. The results for data upload overhead, indexing overhead, and search performance, which are obtained from the experiments on SmallClient, are compared with the results of Apache Hive and Apache Lucene for the same parameters. The comparison showed that SmallClient outperformed existing methods and exhibited improved search performance with reduced indexing overhead.

Mathematical modeling was also utilized to verify the experiment results of SmallClient. Petri nets are leveraged to design the mathematical model for SmallClient. Data uploading time, indexing time, and query execution time are obtained. The experiment results are compared with the mathematical model results, which prove that SmallClient perform similarly in both environments.

7.2 Research Contributions

The contributions of this study to the body of knowledge are as follows:

7.2.1 Taxonomy of State-of-the-Art Indexing Techniques

The taxonomy of indexing techniques was devised, and recent indexing techniques are categorized into NAI, AI, and CAI indexing techniques. Recent indexing contributions from highly cited recent articles of credible journals are reviewed. The taxonomy was created by analyzing each indexing technique. This taxonomy is presented in Chapter 2, which is published (Gani et al., 2015) as a survey on indexing techniques for big data.

7.2.2 SmallClient: a novel indexing framework for big data

A novel indexing framework for big data is proposed, which is called SmallClient. SmallClient implements non-clustered indexing and offers both static and adaptive indexing mechanisms (see Chapter 4). Static indexes are created based on a user-provided list of index attributes regardless of the number of indexes needed at the time of data uploading. Indexing overhead for SmallClient is lower than that of the existing indexing library for big data. Users can invoke index updating, i.e., create new indexes and/or delete available indexes whenever the query workload changes.

The predictor logic is also introduced by SmallClient to adaptively update indexes with changing query workload. Thus, the maximized index hit ratio which was an objective of this study, is obtained. Index maintenance implements a non-clustered approach. Query execution time is improved unlike that of existing full scan and indexed search techniques.

7.2.3 Algorithmic Design for SmallClient

Algorithms are designed for block creation, index creation, and query execution modules for the proposed indexing framework. The step-by-step procedures involved in each module are presented. The algorithm for block creation takes the local disk location of data and its schema as input and shows the process of creating and uploading blocks.

Index creation has two algorithms to accomplish the process: the first algorithm presents the pre-index creation steps, such as verifying the provided list of index attributes and obtaining the location of each index attribute from data set schema. The second algorithm for index creation presents the activities of index creation. The algorithm for query execution takes query as input, verifies the elements of queries, and defines data search and retrieval operations.

7.2.4 Java Class Library for SmallClient[7]

The Java code for SmallClient indexing framework is developed by using Eclipse IDE. Java IO, util and text packages, and Apache Hadoop FileSystem, conf and hdfs packages are imported to implement procedures. The executable client offers block creation (see Appendix A), index creation (see Appendix B), query execution (see Appendix C), and predictor execution (see Appendix D) services. Users invoke various methods with input parameters to obtain the required outputs.

Users invoke block creation by specifying data and their schema source and destination location to upload data to HDFS. Users can also specify index attributes at this stage to invoke index creation in parallel to block creation. Users invoke query execution with query parameters to obtain data. The predictor method does not need any information

[7] The library is publically available on following link:

https://github.com/aasiddiqa/smallclient except the data set location form of users and automatically invokes index creation and index deletion methods. All the methods are well-written and easy to use. Users do not need Java proficiency to apply the solution for big-data analysis.

7.2.5 Mathematical Model of SmallClient

Another contribution of this study is the mathematical model for the indexing framework. Petri net mathematical modeling language was used to develop the SmallClient model. The model provides a clear visualization of the process flow for SmallClient. The required time to perform block creation, index creation, and query execution is calculated by specifying the required parameters. Block creation and index creation time estimation requires data set size, the number of records in a data set, and block size. Query execution time estimation requires index size.

7.3 Limitations and Future Work

Automatic index updating is limited to time setting in the current study. Real-time index updating is disregarded. However, the periodic execution of predictor logic is considered to automatically update indexes. The time setting for a frequently queried system is 10 minutes. After the elapsed time setting, the predictor logic is invoked to update indexes.

Triggered index updating by each incoming query is required in future research to improve index hit performance. A novel predictor logic is proposed to automatically update indexes while periodically predicting future query workload based on incoming query trends.

User-provided metadata is also a limitation of the current study. Users provide schema and metadata for data sets. Input metadata are considered to obtain knowledge on the attributes of a data set. The current work is limited to user-provided metadata. However, a data set may exhibit features other than those defined in the metadata. Users may have data retrieval requirements, which differ from metadata that are available with data sets.

Automatic generation of metadata and provision of multi-schema for various kinds of data sets, such as images, videos, and audio files, are required to induce intelligence in the framework, which may increase the search options and improve the performance of the data retrieval system. Metadata generation is facilitated when the set of properties for a data set increases and more indexes exist for data retrieval.

The predictor logic of SmallClient depends on the derivation of the average access rate for an attribute to make index updating decisions. The average access rate for an attribute is calculated from queries that are executed in n time slots. The average access rate of indexed attributes is used to delete or keep the indexes. In the case of non-indexed attributes, the average access rate is utilized to create new indexes. The exploitation of machine learning methods is required to automatically learn the workload for indexes and to improve the performance of the predictor logic.

SmallClient traverses the entire index for repetitive queries as many times as they are submitted. SmallClient offers query log service to retain information of submitted queries. However, the results of pre-executed queries, i.e., locations of records requested by queries, are unavailable. Therefore, SmallClient repeats index traversal with the submission of queries. Query result caching is required to avoid index traversal for repetitive queries. The cache can return requested data locations with less time than index traversal time.

REFERENCES

Ali, S. T., Sivaraman, V., & Ostry, D. (2013). Authentication of lossy data in body-sensor networks for cloud-based healthcare monitoring. *Future Generation Computer Systems, 35*(0), 80-90. doi:http://dx.doi.org/10.1016/j.future.2013.09.007

Barbierato, E., Gribaudo, M., & Iacono, M. (2014). Performance evaluation of NoSQL big-data applications using multi-formalism models. *Future Generation Computer Systems, 37*(0), 345-353. doi:http://dx.doi.org/10.1016/j.future.2013.12.036

Białecki, A., Muir, R., & Ingersoll, G. (2012). *Apache lucene 4.* Paper presented at the SIGIR 2012 workshop on open source information retrieval: 17-24.

Bordogna, G., Pagani, M., & Pasi, G. (2006). A dynamic hierarchical fuzzy clustering algorithm for information filtering *Soft Computing in Web Information Retrieval* (Vol. 197, pp. 3-23): Springer.

Bouadjenek, M. R., Hacid, H., & Bouzeghoub, M. (2013). *LAICOS: an open source platform for personalized social web search.* Paper presented at the Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining.

Cambazoglu, B. B., Kayaaslan, E., Jonassen, S., & Aykanat, C. (2013). A term-based inverted index partitioning model for efficient distributed query processing. *ACM Trans. Web, 7*(3), 1-23. doi:10.1145/2516633.2516637

Chakrabarti, S., Pathak, A., & Gupta, M. (2011). Index design and query processing for graph conductance search. *The VLDB Journal, 20*(3), 445-470. doi:10.1007/s00778-010-0204-8

Chaudhuri, S., Datar, M., & Narasayya, V. (2004). Index selection for databases: A hardness study and a principled heuristic solution. *Knowledge and Data Engineering, IEEE Transactions on, 16*(11), 1313-1323.

Chen-Yu, C., Ta-Cheng, W., Jhing-Fa, W., & Li Pang, S. (2009, 19-24 April 2009). *SVM-based state transition framework for dynamical human behavior identification.* Paper presented at the Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on:1933-1936.

Chen, J., Chen, Y., Du, X., Li, C., Lu, J., Zhao, S., & Zhou, X. (2013). Big data challenge: a data management perspective. *Frontiers of Computer Science, 7*(2), 157-164. doi:10.1007/s11704-013-3903-7

Cheng, J., Ke, Y., Fu, A. W.-C., & Yu, J. X. (2011). Fast graph query processing with a low-cost index. *The VLDB Journal, 20*(4), 521-539.

Chu, W. W., Liu, Z., Mao, W., & Zou, Q. (2005). A knowledge-based approach for retrieving scenario-specific medical text documents. *Control Engineering Practice, 13*(9), 1105-1121. doi:http://dx.doi.org/10.1016/j.conengprac.2004.12.011

Comer, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys (CSUR), 11*(2), 121-137.

Cuggia, M., Mougin, F., & Beux, P. L. (2005). Indexing method of digital audiovisual medical resources with semantic Web integration. *International Journal of Medical Informatics, 74*(2–4), 169-177. doi:http://dx.doi.org/10.1016/j.ijmedinf.2004.04.027

Dieng-Kuntz, R., Minier, D., Růžička, M., Corby, F., Corby, O., & Alamarguy, L. (2006). Building and using a medical ontology for knowledge management and cooperative work in a health care network. *Computers in Biology and Medicine, 36*(7–8), 871-892. doi:http://dx.doi.org/10.1016/j.compbiomed.2005.04.015

Dittrich, J., Blunschi, L., & Vaz Salles, M. (2011). MOVIES: indexing moving objects by shooting index images. *GeoInformatica, 15*(4), 727-767. doi:10.1007/s10707-011-0122-y

Dittrich, J., Quian, J.-A., Quiané-Ruiz, Richter, S., Schuh, S., Jindal, A., & Schad, J. (2012). Only aggressive elephants are fast elephants. *Proc. VLDB Endow., 5*(11), 1591-1602. doi:10.14778/2350229.2350272

Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., & Schad, J. (2010). Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment, 3*(1-2), 515-529.

Done, B., Khatri, P., Done, A., & Draghici, S. (2010). Predicting novel human gene ontology annotations using semantic analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), 7*(1), 91-99.

Eldawy, A., & Mokbel, M. F. (2015). *Spatial Hadoop: A MapReduce Framework for Spatial Data*. Paper presented at the 2015 IEEE 31st International Conference on Data Engineering.

Elleuch, N., Zarka, M., Ammar, A. B., & Alimi, A. M. (2011). *A fuzzy ontology: based framework for reasoning in visual video content analysis and indexing*. Paper presented at the Proceedings of the Eleventh International Workshop on Multimedia Data Mining, San Diego, California.

Ferragina, P., & Venturini, R. (2010). The compressed permuterm index. *ACM Trans. Algorithms, 7*(1), 1-21. doi:10.1145/1868237.1868248

Gacto, M. J., Alcala, R., & Herrera, F. (2010). Integration of an Index to Preserve the Semantic Interpretability in the Multiobjective Evolutionary Rule Selection and Tuning of Linguistic Fuzzy Systems. *Fuzzy Systems, IEEE Transactions on, 18*(3), 515-531. doi:10.1109/TFUZZ.2010.2041008

Gani, A., Siddiqa, A., Shamshirband, S., & Hanum, F. (2015). A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowledge and Information Systems, 46*(2), 1-44. doi:10.1007/s10115-015-0830-y

Gospodnetic, O., & Hatcher, E. (2005). *Lucene*: Manning:1-421.

Halim, F., Idreos, S., Karras, P., & Yap, R. H. (2012). Stochastic Database Cracking: towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment, 5*(6), 502-513

Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., & Khan, S. U. (2015). The rise of "big data" on cloud computing: Review and open research issues. *Information Systems, 47*, 98-115.

Hsu, W., Lee, M. L., Ooi, B. C., Mohanty, P. K., Teo, K. L., & Xia, C. (2002). *Advanced database technologies in a diabetic healthcare system*. Paper presented at the Proceedings of the 28th international conference on Very Large Data Bases, Hong Kong, China.

Huang, Z., Lu, X., Duan, H., & Zhao, C. (2012). Collaboration-based medical knowledge recommendation. *Artificial Intelligence in Medicine, 55*(1), 13-24.

Idreos, S., Alagiannis, I., Johnson, R., & Ailamaki, A. (2011). *Here are my Data Files. Here are my Queries. Where are my Results?* Paper presented at the Proceedings of 5th Biennial Conference on Innovative Data Systems Research, No. EPFL-CONF-161489.

Jayaraman, U., Prakash, S., & Gupta, P. (2013). Use of geometric features of principal components for indexing a biometric database. *Mathematical and Computer Modelling, 58*(1–2), 147-164. doi:http://dx.doi.org/10.1016/j.mcm.2012.06.005

Jensen, K., Kristensen, L. M., & Wells, L. (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf., 9*(3), 213-254. doi:10.1007/s10009-007-0038-x

Jindal, A., Quiané-Ruiz, J.-A., & Dittrich, J. (2011). *Trojan data layouts: right shoes for a running elephant.* Paper presented at the Proceedings of the 2nd ACM Symposium on Cloud Computing.

Kadiyala, S., & Shiri, N. (2008). A compact multi-resolution index for variable length queries in time series databases. *Knowledge and Information Systems, 15*(2), 131-147.

Kaisler, S., Armour, F., Espinosa, J. A., & Money, W. (2013, 7-10 Jan. 2013). *Big Data: Issues and Challenges Moving Forward.* Paper presented at the System Sciences (HICSS), 2013 46th Hawaii International Conference on.

Katal, A., Wazid, M., & Goudar, R. H. (2013, 8-10 Aug. 2013). *Big data: Issues, challenges, tools and Good practices.* Paper presented at the Contemporary Computing (IC3), 2013 Sixth International Conference on: 404-409.

Kaushik, V. D., Umarani, J., Gupta, A. K., Gupta, A. K., & Gupta, P. (2013). An efficient indexing scheme for face database using modified geometric hashing. *Neurocomputing, 116*(0), 208-221. doi:http://dx.doi.org/10.1016/j.neucom.2011.12.056

Kelley, J., Stewart, C., Morris, N., Tiwari, D., He, Y., & Elnikety, S. (2015). *Measuring and managing answer quality for online data-intensive services.* Paper presented at the Autonomic Computing (ICAC), 2015 IEEE International Conference on.

Komkhao, M., Lu, J., Li, Z., & Halang, W. A. (2013). Incremental collaborative filtering based on Mahalanobis distance and fuzzy membership for recommender systems. *International Journal of General Systems, 42*(1), 41-66.

Kwon, O., Lee, N., & Shin, B. (2014). Data quality management, data usage experience and acquisition intention of big data analytics. *International Journal of Information Management, 34*(3), 387-394.

LaValle, S., Lesser, E., Shockley, R., Hopkins, M. S., & Kruschwitz, N. (2013). Big data, analytics and the path from insights to value. *MIT Sloan Management Review, 52*(2), 21-32.

Lazaridis, M., Axenopoulos, A., Rafailidis, D., & Daras, P. (2013). Multimedia search and retrieval using multimodal annotation propagation and indexing techniques. *Signal Processing: Image Communication, 28*(4), 351-367. doi:http://dx.doi.org/10.1016/j.image.2012.04.001

Leung, C. H. C., & Chan, W. S. (2010). Semantic Music Information Retrieval Using Collaborative Indexing and Filtering. In E. Gelenbe, R. Lent, G. Sakellari, A. Sacan, H. Toroslu, & A. Yazici (Eds.), *Computer and Information Sciences* (Vol. 62, pp. 345-350): Springer Netherlands.

Li, F., Hadjieleftheriou, M., Kollios, G., & Reyzin, L. (2010). Authenticated Index Structures for Aggregation Queries. *ACM Trans. Inf. Syst. Secur., 13*(4), 1-35. doi:10.1145/1880022.1880026

Li, F., Yi, K., & Le, W. (2010). Top-k queries on temporal data. *The VLDB Journal—The International Journal on Very Large Data Bases, 19*(5), 715-733.

Li, G., Feng, J., Zhou, X., & Wang, J. (2011). Providing built-in keyword search capabilities in RDBMS. *The VLDB Journal, 20*(1), 1-19.

MacNicol, R., & French, B. (2004). *Sybase IQ multiplex - designed for analytics*. Paper presented at the Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, Toronto, Canada.

Maier, M., Rattigan, M., & Jensen, D. (2011). Indexing network structure with shortest-path trees. *ACM Transactions on Knowledge Discovery from Data (TKDD), 5*(3), 15.

Mehrotra, H., Majhi, B., & Gupta, P. (2010). Robust iris indexing scheme using geometric hashing of SIFT keypoints. *Journal of Network and Computer Applications, 33*(3), 300-313. doi:http://dx.doi.org/10.1016/j.jnca.2009.12.005

Ongenae, F., Claeys, M., Dupont, T., Kerckhove, W., Verhoeve, P., Dhaene, T., & De Turck, F. (2013). A probabilistic ontology-based platform for self-learning context-aware healthcare applications. *Expert Systems with Applications, 40*(18), 7629-7646. doi:http://dx.doi.org/10.1016/j.eswa.2013.07.038

Paul, A., Chen, B.-W., Bharanitharan, K., & Wang, J.-F. (2013). Video search and indexing with reinforcement agent for interactive multimedia services. *ACM Trans. Embed. Comput. Syst., 12*(2), 1-16. doi:10.1145/2423636.2423643

Philip Chen, C., & Zhang, C.-Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences, 275*, 314-347.

Phuvipadawat, S., & Murata, T. (2010, Aug. 31 2010-Sept. 3 2010). *Breaking News Detection and Tracking in Twitter.* Paper presented at the Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on: 120-123.

Qian, X., Tagare, H. D., Fulbright, R. K., Long, R., & Antani, S. (2010). Optimal embedding for shape indexing in medical image databases. *Medical Image Analysis, 14*(3), 243-254. doi:http://dx.doi.org/10.1016/j.media.2010.01.001

Raghavendra, S., Mara, G., Buyya, R., Rajuk, V. K., Iyengar, S., & Patnaik, L. (2016). *DRSIG: Domain and Range Specific Index Generation for Encrypted Cloud Data.* Paper presented at the Computational Techniques in Information and Communication Technologies (ICCTICT), 2016 International Conference on.

Richter, S., Quiané-Ruiz, J.-A., Schuh, S., & Dittrich, J. (2012). Towards zero-overhead adaptive indexing in Hadoop. *arXiv preprint arXiv:1212.3480.*

Richter, S., Quiané-Ruiz, J.-A., Schuh, S., & Dittrich, J. (2014). Towards zero-overhead static and adaptive indexing in Hadoop. *The VLDB Journal, 23*(3), 469-494. doi:10.1007/s00778-013-0332-z

Rodríguez-García, M. Á., Valencia-García, R., García-Sánchez, F., & Samper-Zapater, J. J. (2013). Creating a semantically-enhanced cloud services environment through ontology evolution. *Future Generation Computer Systems, 32*(0), 295-306. doi:http://dx.doi.org/10.1016/j.future.2013.08.003

Russo, L. M., Navarro, G., & Oliveira, A. L. (2008). Fully-compressed suffix trees *LATIN 2008: Theoretical Informatics* (pp. 362-373): Springer.

Sandu Popa, I., Zeitouni, K., Oria, V., Barth, D., & Vial, S. (2011). Indexing in-network trajectory flows. *The VLDB Journal—The International Journal on Very Large Data Bases, 20*(5), 643-669.

Schuh, S., & Dittrich, J. (2015, 13-17 April 2015). *AIR: Adaptive Index Replacement in Hadoop.* Paper presented at the Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on: 22-29.

Shang, L., Yang, L., Wang, F., Chan, K.-P., & Hua, X.-S. (2010). *Real-time large scale near-duplicate web video retrieval.* Paper presented at the Proceedings of the international conference on Multimedia.

Siddiqa, A., Karim, A., & Gani, A. (2016). Big data storage technologies: a survey. *Frontiers of Information Technology & Electronic Engineering, 1.*

Siddiqa, A., TargioHashem, I. A., Yaqoob, I., Marjani, M., Shamshirband, S., Gani, A., & Nasaruddin, F. (2016). A Survey of Big Data Management: Taxonomy and State-of-the-Art. *Journal of Network and Computer Applications.*

Sinha, R. R., & Winslett, M. (2007). Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst., 32*(3), 16. doi:10.1145/1272743.1272746

Thilakanathan, D., Chen, S., Nepal, S., Calvo, R., & Alem, L. (2013). A platform for secure monitoring and sharing of generic health data in the Cloud. *Future Generation Computer Systems, 35*(0), 102-113. doi:http://dx.doi.org/10.1016/j.future.2013.09.011

van der Spek, P., & Klusener, S. (2011). Applying a dynamic threshold to improve cluster detection of LSI. *Science of Computer Programming, 76*(12), 1261-1274. doi:http://dx.doi.org/10.1016/j.scico.2010.12.004

Wai-Tat, F. (2012). Collaborative Indexing and Knowledge Exploration: A Social Learning Model. *IEEE Intelligent Systems, 27,* 39-46.

Wang, C.-H., Jiau, H. C., Chung, P.-C., Ssu, K.-F., Yang, T.-L., & Tsai, F.-J. (2010). A novel indexing architecture for the provision of smart playback functions in collaborative telemedicine applications. *Computers in Biology and Medicine, 40*(2), 138-148.

Wang, J., Wu, S., Gao, H., Li, J., & Ooi, B. C. (2010). *Indexing multi-dimensional data in a cloud system.* Paper presented at the Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.

Wang, M., Holub, V., Murphy, J., & O'Sullivan, P. (2013). High volumes of event stream indexing and efficient multi-keyword searching for cloud monitoring. *Future Generation Computer Systems, 29*(8), 1943-1962.

Wang, X., Luo, X., & Liu, H. (2014). Measuring the veracity of web event via uncertainty. *Journal of Systems and Software*(0), 1-11. doi:http://dx.doi.org/10.1016/j.jss.2014.07.023

Wang, Y. (2008). On contemporary denotational mathematics for computational intelligence *Transactions on computational science II* (pp. 6-29): Springer, **5150**.

Wei, L.-Y., Hsu, Y.-T., Peng, W.-C., & Lee, W.-C. (2013). Indexing spatial data in cloud data managements. *Pervasive and Mobile Computing, 15*(0), 48-61. doi:http://dx.doi.org/10.1016/j.pmcj.2013.07.001

Weng, M.-F., & Chuang, Y.-Y. (2012). Collaborative video reindexing via matrix factorization. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), 8*(2), 23.

Wu, D., Cong, G., & Jensen, C. S. (2012). A framework for efficient spatial web object retrieval. *The VLDB Journal—The International Journal on Very Large Data Bases, 21*(6), 797-822.

Wu, K., Shoshani, A., & Stockinger, K. (2010). Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst., 35*(1), 1-52. doi:10.1145/1670243.1670245

Wu, S., Wang, Z., & Xia, S. (2009). *Indexing and retrieval of human motion data by a hierarchical tree*. Paper presented at the Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology, Kyoto, Japan.

Yang, C., Zhang, X., Zhong, C., Liu, C., Pei, J., Ramamohanarao, K., & Chen, J. (2014). A spatiotemporal compression based approach for efficient big data processing on Cloud.

Journal of Computer and System Sciences, 80(8), 1563-1583. doi:http://dx.doi.org/10.1016/j.jcss.2014.04.022

Yeh, S.-C., Su, M.-Y., Chen, H.-H., & Lin, C.-Y. (2013). An efficient and secure approach for a cloud collaborative editing. *Journal of Network and Computer Applications, 36*(6), 1632-1641. doi:http://dx.doi.org/10.1016/j.jnca.2013.05.012

Yıldırım, H., Chaoji, V., & Zaki, M. (2012). GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal, 21*(4), 509-534. doi:10.1007/s00778-011-0256-4

Yuan, D., & Mitra, P. (2013). Lindex: a lattice-based index for graph databases. *The VLDB Journal, 22*(2), 229-252. doi:10.1007/s00778-012-0284-8

Zhang, P., Zhou, C., Wang, P., Gao, B. J., Zhu, X., & Guo, L. (2015). E-tree: An efficient indexing structure for ensemble models on data streams. *IEEE Transactions on Knowledge and Data Engineering, 27*(2), 461-474.

Zhu, X., Huang, Z., Cheng, H., Cui, J., & Shen, H. T. (2013). Sparse hashing for fast multimedia search. *ACM Trans. Inf. Syst., 31*(2), 1-24. doi:http://dx.doi.org/10.1145/2457465.2457469

Zhuang, Y., Jiang, N., Li, Q., Chen, L., & Ju, C. (2015). Progressive Batch Medical Image Retrieval Processing in Mobile Wireless Networks. *ACM Trans. Internet Technol., 15*(3), 1-27. doi:10.1145/2783437

Zou, Z., Wang, Y., Cao, K., Qu, T., & Wang, Z. (2013). Semantic overlay network for large-scale spatial information indexing. *Computers & Geosciences, 57*(0), 208-217. doi:http://dx.doi.org/10.1016/j.cageo.2013.04.019

LIST OF PUBLICATIONS AND PAPERS PRESENTED

All the contributions (see Section objectives) of this thesis were sent for publication to premier peer-reviewed journals in the field of Computer Science. The following publications are used in this thesis:

1. Gani, Abdullah, Siddiqa, Aisha, Shamshirband, Shahaboddin, & Hanum, Fariza. (2015). **A survey on indexing techniques for big data: taxonomy and performance evaluation**. *Knowledge and Information Systems*, 46(2), 1-44. doi: 10.1007/s10115-015-0830-y (Q1 Publication)
2. Siddiqa, Aisha, I.Targio, I.Yaqoob, M.Marjani, S.Shamshirband, A.Gani & F.Hanum (2016). **A Survey of Big Data Management: Taxonomy and State-of-the-Art**. *Journal of Network and Computer Applications* doi: http://dx.doi.org/10.1016/j.jnca.2016.04.008 (Q1 Publication)
3. Siddiqa, Aisha, Karim, Ahmad, Gani, Abdullah. (2016). **Big data storage technologies: a survey**. *Frontiers of Information Technology & Electronic Engineering* (Q3 Publication)
4. Siddiqa, Aisha, Karim, Ahmad, Saba, Tanzila and Chang, Victor (2016). **On the analysis of big data indexing execution strategies**. *Journal of Intelligent and Fuzzy Systems* (Accepted)
5. Siddiqa, Aisha, Karim, Ahmad and Chang, Victor, **SmallClient for Big Data: An Indexing Framework towards fast Data Retrieval** (2016), *Cluster Computing* (Q1 Publication)
6. **Modelling SmallClient indexing framework for big data analytics** *Supercomputing,* (Under Review)
7. **Big IoT Data Analytics State-of-the-Art and Open Challenges** *IEEE Transactions,* (Under Review)
8. **Greening the Emerging IT Technologies: Techniques and Practices** *Journal of Internet Services and Applications,* (Under Review)