Noname manuscript No.
(will be inserted by the editor)

SmallClient for big data: an indexing framework towards fast data retrieval

Aisha Siddiqa · Ahmad Karim · Victor Chang

Received: date / Accepted: date

Abstract Numerous applications are continuously generating massive amount of data and it has become critical to extract useful information while maintaining acceptable computing performance. The objective of this work is to design an indexing framework which minimizes indexing overhead and improves query execution and data search performance with optimum aggregation of computing performance. We propose SmallClient, an indexing framework to speed up query execution. SmallClient has three modules: block creation, index creation and query execution. Block creation module supports improving data retrieval performance with minimum data uploading overhead. Index creation module allows maximum indexes on a dataset to increase index hit ratio with minimized indexing overhead. Finally, query execution module offers incoming queries to utilize these indexes. The evaluation shows that SmallClient outperforms Hadoop full scan with more than 90% search performance. Meanwhile, indexing overhead of SmallClient is reduced to approximately 50% and 80% for index size and indexing time respectively.

Keywords Big data · Big data indexing · Big data retrieval · Big data analytics · Query execution · Data search performance

A. Siddiqa
Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur, 50603, Malaysia
Tel.: +60-11-14391908
E-mail: aasiddiqa@gmail.com

A. Karim
Department of Information Technology, Bahauddin Zakariya University, Multan, 60000, Pakistan

V. Chang
IBSS, Xi'an Jiaotong Liverpool University, Suzhou, 100044, China

1 Introduction

With the evolution of big data technologies, the research trends have been moved from finding massive storage to efficient big data analytics. In todays competitive world, being up-to-date has become a necessity of every business to survive in tremendously changing situations [1]. For this purpose, obtaining timely responses to queries plays a significant role in decision making [2] where researchers are talking about big data security [3][4], privacy on social network [5] and analytics for internet of things [6]. Furthermore, efficient computing resource utilization to perform analytics and search operations on big data is also a critical aspect. Therefore, the researchers are inclined to come up with efficient data analytics solutions for rapidly growing amount of data. Distributed parallel processing systems are widely adopted by big data analytics where data volumes have exceeded from exabytes and are still explosively growing [7][8]. For such large repositories, it has become challenging to practice data analysis, search and retrieval results with same performance as before and continuous improvement to meet efficiency requirements is needed [9][10]. As a result, data indexing has always been an efficient mechanism to increase query execution and data search performance.

Contemporary big data processing technologies are efficient to perform mining operation such as CouchDB for mining document data [11]. Similarly, text mining [12] and data mining [13] algorithms and procedures are also well-implemented for big data. Furthermore, these technologies are prone to adopt feasible indexing structures for better data analytical performance. For instance, Hadoop which is a de facto big data processing framework has gained 32 times improvement in task execution with implementation of indexing [14]. However, indexing techniques which are proven to be efficient for traditional datasets do not perform well when applied for big data. In addition, indexing time and size are also very crucial for voluminous datasets. Another challenge for big data analytics is that, it is impractical to face longer delays between data uploading and starting data search operations [15][16]. Meanwhile, even bigger indexes for big data do not make sense. These facts motivate us to explore more about indexing and make further advancements in indexing procedures for big data.

Consequently, in this paper we focus on minimizing indexing overhead and improving query execution and search performance for voluminous datasets along with velocity of processing needs while aggregating computing resource utilization. We consider both index creation time and index size to present minimized indexing overhead in our work. Furthermore, to evaluate query execution and search performance, we measure index traversing time and data retrieval time. In order to carry out research process, we first briefly describe the inefficiency of contemporary big data processing frameworks when full scan query operations on large datasets are performed. We show that the performance of query execution declines at Hadoop MapReduce framework when data size grows. We further demonstrate the inability of recent indexing structures to deal with large size datasets. Additionally, we conclude that index attribute set size is constrained to replication factor for clustered indexing mechanisms [14] and with less number of indexes the chances of full scan become high. While, non-clustered indexing mechanisms result in longer delays to start query execution [17]. Besides index creation time, size of index is also large. Moreover, non-clustered indexes for big data are not well structured to retrieve data attributes other than indexed attributes which is a serious performance bottleneck. Another aspect related to data retrieval is block creation policy. File systems composed with big data processing frameworks, HDFS as an example, creates blocks by taking subsequent bytes from dataset to fill a block. HDFS does not take care of contents or so called records of dataset while creating blocks and thus record split is common phenomenon. Split of last record in a block increases access time to that record as next block containing broken part of record may be located on distant site in a distributed data storage environment.

We introduce SmallClient, a first non-clustered block level indexing client for big data which offers multiple indexes to be created on datasets regardless of available replication factor. Unlike clustered indexing, our non-clustered indexing approach is independent of replication factor and thus, allows more indexes to be created than clustered indexing. Hence, larger index attribute space than clustered indexing increases index hit ratio for incoming search queries. Furthermore, the introduced block level indexing concept utilizes the properties of data blocks, due to which, indexes are more manageable than full data indexing. In comprehend, the benefits of proposed indexing client are threefold: first, our data block creation policy prevents record splitting in more than one blocks and minimizes record access time. Second, as many attributes as provided by a user are utilized to create indexes with least indexing overhead and improved index hit ratio. Third, our indexes are fast enough in creation and traversal with less space consumption which results in improved search performance.

The rest of the paper is organized as follows: Section 2 discusses the work related to clustered and non-clustered indexing and their unaddressed problems which motivate our work. Section 3 presents the proposed SmallClient indexing framework for big data. A B-Tree based block level indexing mechanism for big data analytics and a detailed description of SmallClient components is provided. Section 4 explains the experimental setup and discusses the results. Finally, conclusion and future work are described in Section 5.

2 Related Work

In this section, we provide a brief review and background information of state-of-the-art contribution in the field of big data indexing. Indexing has a widespread implication for information retrieval in an efficient manner. However in the field of big data indexing, the contribution of database cracking [18] is an inaugural effort. Database cracking is derived to propose clustered indexing solutions which are implemented on Hadoop and have shown enhanced data retrieval performance. MapReduce programming model of Hadoop which has become de facto for big data analytics does not perform well in some aspects when compared with traditional data management systems (i.e. shared nothing databases) [19]. Therefore, a trend is started to integrate these traditional systems with MapReduce to utilize the benefits of both technologies. HadoopDB [20] is one example of this integration. Due to architectural complexities found in HadoopDB and other such systems, the idea of integration is not well-acknowledged. Alternatively, Hadoop++ [21] has been introduced which is simpler and an easily deployable solution. Moreover,

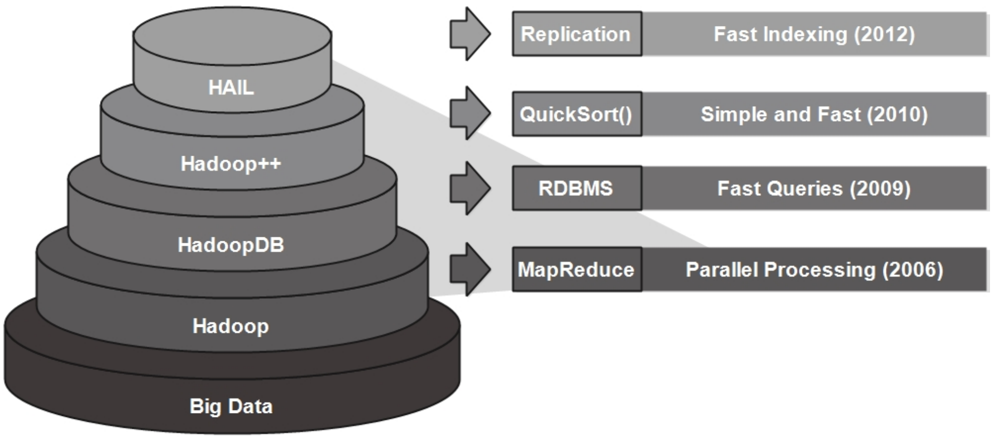


Fig. 1 Evolution of Big Data Indexing

Table 1 Clustered and non-clustered Indexing Approaches

Features	Clustered Indexing	Non-Clustered Indexing
Process	Physically re-orders data rows	Separate structure of key-value pairs
No. of Indexes	Depends on No. of replicas	As much as No. of Attributes in schema
Index Size	Small Index Metadata	Separate index structure needs significant space
Index Updating	Requires re-ordering whole data	Requires traversal of data
Data write	Slow(requires re-ordering)	A key-value pair is inserted in each index
Data Read	Fast (searches in sorted list)	First traverses index then jumps to record

Hadoop++ has same or sometimes improved query execution performance over HadoopDB. Hadoop++ proposes Trojan indexing which implements the concept of data sorting (same as database cracking) at the time of uploading data and thus facilitates query execution times.

As a consequence, task execution and data search performance of Hadoop framework is undoubtedly improved with Hadoop++ (i.e. 20 times). However, Trojan index has two drawbacks: first, the index creation time of Trojan index is much longer than executing a full sequential scan operation on Hadoop. Second, Trojan index offers only one index for a dataset and thus the performance improvement in query execution is subject to execute queries having only that index attribute as selection predicate. In this case, the selection of an attribute to create indexes is very crucial. To deal with these performance bottlenecks, HAIL [14] utilizes data replication rather than just providing reliability and load balancing to increasing the number of indexes. HAIL offers as many indexes as replication factor is set for a dataset and achieves up to 64 times improved task execution performance. Despite from task execution performance, the number of indexes in HAIL is constrained to replication factor and it does not make sense to create more replicas of large datasets to increases number of indexes. Although some improvements are proposed in HAIL yet there is no solution independent from data replication.

This evolution of Hadoop in terms of improved data retrieval is elaborated in Fig. 1. Fig. 1 shows that Hadoop was introduced with MapReduce as programming model that performs specialized parallel processing operations on distributed data to fasten job execution procedure. However, the simplicity of design offered by Hadoop has bypassed the features associated with traditional database management systems. Therefore, HadoopDB [20] was introduced. As we have discussed earlier, the architectural complexities of HadoopDB have impeded its acceptance. However, Hadoop++ [21] and HAIL [14] have proved to be efficient clustered indexing implementations with improved indexing structures to achieve improved search performance on big data.

As far as non-clustered data indexing is concerned, it is available in four categories: tree-based, hashing, inverted and bitmap indexes. Unlike clustered approach of indexing, physical reordering of data is not required to create non-clustered indexes. Non-clustered indexing has been an efficient query execution and data retrieval mechanism for medical images [22], event stream data [23] and for face databases [24]. In our previous work [25], we have elaborated that non-clustered indexes are fast in creation, robust, small in size and their computational cost is less. However, B-Tree is more feasible as these indexes are adaptable to growing size and suitable for various types of data. Furthermore, the tree based structure makes it fast in traversing as compared to other non-clustered indexes. Recent implementations of B-Tree indexing are on flash memory optimization [26] and on main memory [27].Apache Lucene [34] implements inverted indexes that are non-clustered indexes on big data to retrieve required data by searching indexes instead of whole data sets. However, the index design implemented by Apache Lucene has few limitations: indexing on whole data set requires a significant size of main memory, searching data for non-indexed attributes and searching whole row when selective attributes are indexed is not available with Apache Lucene indexes.

We present a brief comparison of clustered and non-clustered indexing approaches in Table 1. Table 1 shows that both indexing approaches have their own benefits and limitations. However, replica dependency of clustered indexing is the main constraint in implementing this approach. We have also discussed the design limitations of Apache Lucene that implements non-clustered indexing. Although Apache Lucene indexing library is advantageous on clustered approach, these limitations deteriorate the performance of Apache Lucene indexing. Consequently, we propose B-Tree based indexing framework, a non-clustered indexing approach, that creates block level indexes and overcomes the challenges of existing clustered and non-clustered indexing approaches.

3 SmallClient

In this section, we present our indexing framework named as, SmallClient in detail and elaborate how it improves search performance and increases index hit ratio by enabling larger index attribute space than state-of-the-art indexing solutions. We further describe the achievement of SmallClient to reduce index creation time and index size for larger size data. Fig. 2 presents our proposed indexing client. In a systematic way, we decompose our client into three modules: first module is designed to create data blocks which splits 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 of our framework which utilizes B-Tree structure to store $< key, value >$ pairs extracted from data blocks. Finally, expected data can be retrieved using query execution module which shows improved search performance for larger datasets.

SmallClient is a generalized framework for big data indexing that is implementable on any distributed file system. SmallClient works as an intermediate layer between user and distributed file system and offers data uploading and query execution mechanism. SmallClient offers indexes to facilitate search operations on data. We present the architecture for SmallClient in Fig. 3 that comprises three layers: (1) User Interface (UI) layer, (2) SmallClient layer and (3) File System layer. User initiates data uploading and index creating operations via UI layer. User also sumbits queries on UI layer and the results of indexed search are returned by SmallClient on UI. SmallClient layer accepts data uploading and indexing instructions from UI and invokes block creation to store data on file system and index creation to create indexes on stored/storing data. SmallClient layer also takes queries as input from UI layer, invokes indexed search on data that is stored in file system and returns the required data to user via UI. File System layer is responsible to accommodate data blocks and specified replicas on available storage. File System layer also stores block metadata, schema of data set, indexes, index metadata and query log. SmallClient utilizes different procedures associated with file system in order to store, load and retrieve files on file systems. For instance, file path, live nodes and capacity are HDFS information involved to carry out data storage and retrieval operations.

3.1 Block Creation

Contemporary big data processing systems offer distributed storage for big data where data reliability is enforced with the help of data replication. Meantime, a lot of debate is available in literature to signify data chunk storage instead of storing data as a whole. Therefore, each big data storage system has its own data splitting mechanism where block size and placement in file system is decided. Generally, block size is fixed for a file and last record faces breakage when splitting data into fixed size blocks. These blocks are then placed on any allocated site in distributed file system regardless of taking care of accessing broken records. As a result, accessing more than one site to retrieve that broken record increases overall data load time. However, in order to decrease time required to access resulting records, each record should be accessed as a whole from a single site. That is, we introduce block creation in a way that last record in each block is never split, presented in Fig. 4.

During block creation phase, records are read and stored in a block until the block reaches its storage limit. We offer block size which is adjustable according to normal

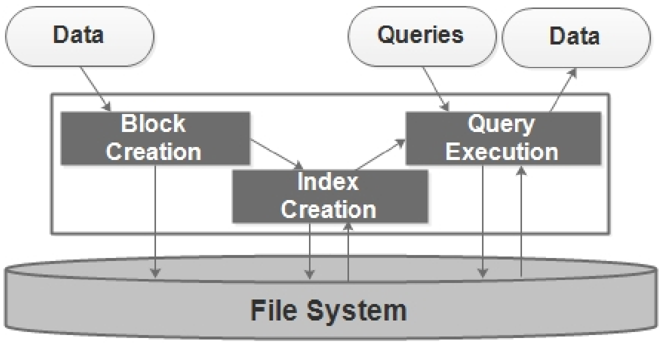


Fig. 2 Illustration of SmallClient components over a File System

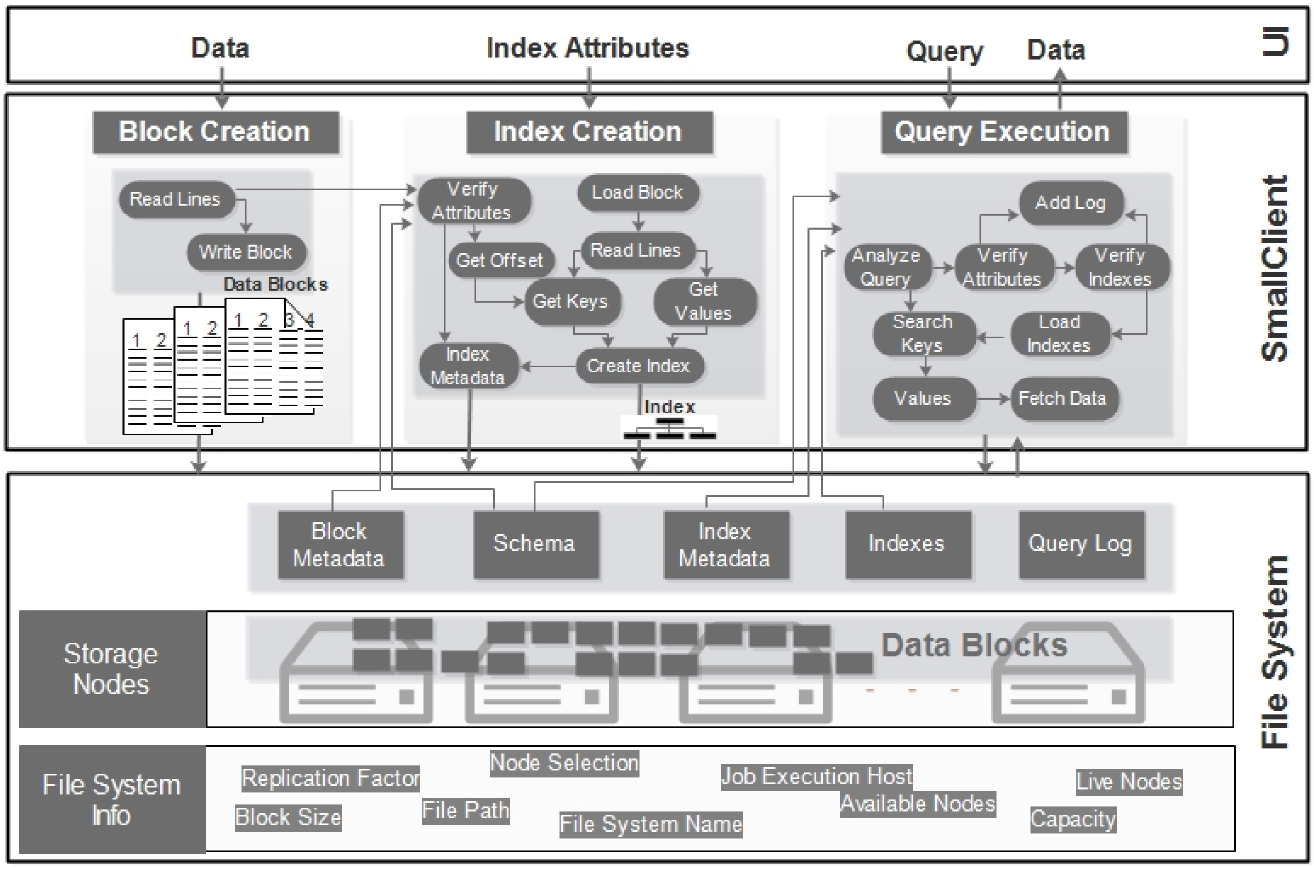


Fig. 3 The Architecture for SmallClient

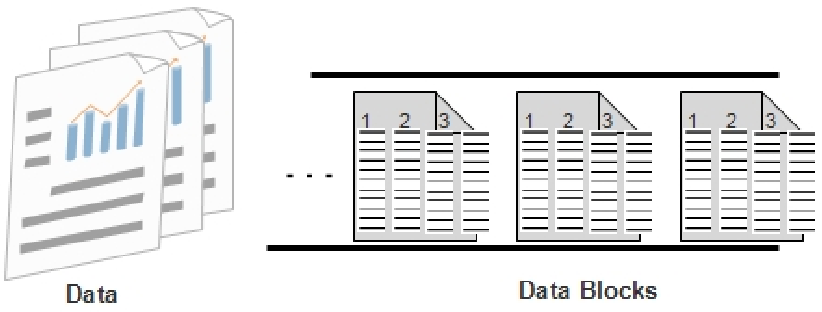


Fig. 4 Block Creation Process

Algorithm 1 createBlocks($file$)

```
1: $block\_limit = DefaultBloockSize$
2: $has\_capacity = true$
3: $block\_number = 0$
4: while reading records not reached end of file do
5:     if $block$ $has$ $capacity(block\_limit)$ then
6:         add record in $block$
7:     else
8:         uploadBlock($block$, $block\_number$)
9:         $block\_number = block\_number + 1$
10:        $has\_capacity = true$
11:    end if
12: end while
13: uploadBlock($block$, $block\_number$)
```

record size in data or the default block size of a file system. Suppose, the dataset is represented as D and comprises x records (as shown in Eq. 1), k blocks will be created from the dataset (as shown in Eq. 2). Each block will contain mi records and some free space. The performance analysis as presented in Fig. 7 data overhead motivates our design of block creation and shows that this overhead decreases as data size grows. The results show that block creation process is negligible for large datasets and it takes few bytes more than the original dataset size. The process of block creation is presented in Algorithm 1. Block size is adjustable in SmallClient. However, we utilize HDFS default block size (i.e. 64MB) for implementation and evaluation. Block creation takes place before data uploading in a distributed file system and divides data into smaller blocks. Each block is then uploaded with adjustable replication factor into file system. The next step is to create indexes.

$$D = \sum_{c=1}^{x} record_c \tag{1}$$

$$B'_i = \sum_{c=1}^{mi} record_c + \alpha \tag{2}$$

where B' denotes the created blocks and size of each block is $S_{B'} = l$

3.2 Index Creation

Index creation process takes place after data is uploaded to file system and indexes will be created to improve data retrieval time. We need to create fast traversable indexes for big data so that data search time against queries is minimized. Moreover, there is also a need to reduce the overhead caused by index creation so that not only the delay between data uploading and starting query execution is minimized but the extra space consumed by indexes is also reduced. Index creation process using SmallClient reduces this size

Algorithm 2 runIndex($file\_name$, $file\_schema$, $index\_attr\_list$)

```
1: if $index\_attr\_list$ is empty then
2:     write $err\_message$
3:     exit
4: end if
5: compare $index\_attr\_list$ with $file\_schema$ & remove unmatched attributes from $index\_attr\_list$
6: calculate $index\_attr\_offset\_list$ from updated $index\_attr\_list$
7: get $block\_locations$
8: for all $blocks$ do
9:     createIndexes($file$, $block\_locations$, $index\_attr\_offset\_list$)
10:    for all $indexes$ do
11:        storeIndex$index$, $file\_name$, $index\_attr$
12:        get & update $index\_metadata$
13:    end for
14: end for
```

Algorithm 3 createIndex($file$, $block\_locations$, $index\_attr\_offset\_list$)

```
1: for all $index\_attr$ do
2:     create empty $BTree$
3: end for
4: $value = block\_offset$
5: while reading records not reached end of $block$ do
6:     for all $index\_attr$ do
7:         $key = contentsatindex\_attr\_offset$
8:         add $< key, value >$ in its $BTree$
9:     end for
10: end while
11: store each $BTree$
12: store $index\_metadata$ of each $BTree$
```

and time overhead by utilizing B-Tree structure for indexing. During index creation phase, separate indexes are created for each block and for each index attribute. SmallClient takes the contents from a record for each index attribute as key and the location of that record as $value$. Fig 5 depicts the process of obtaining two tuple record as $< key, value >$ pairs from data blocks and storing them in B-Tree. More than one occurrences of a key in a block are stored as list of values in B-Tree. The results presented in later sections will show the performance of index creation with less index size and index creation time.

The process of index creation is elaborated in Algorithm 2 and 3. Algorithm 2 shows the pre-index creation steps involved in getting verified index attribute set (aI) according to provided schema (aD) of that dataset. Offset addresses of index attributes are also obtained from schema which are helpful to jump to contents as keys in a record. Later, index creation phase is invoked. Eq. 3 shows that index attributes set a is a subset of data attributes set aD as given in schema

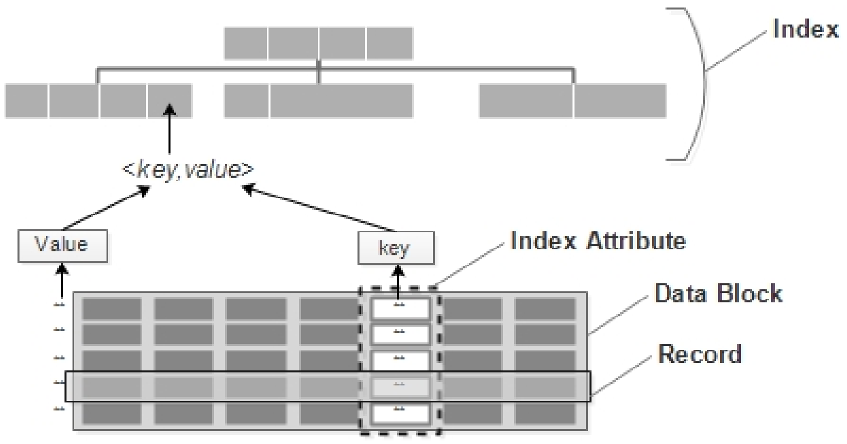


Fig. 5 Index Creation Process

of dataset. We define an index I for a block i having m records in Eq. 4.

$$aI \subseteq aD \tag{3}$$

$$I_{attr,i} = \sum_{c=1}^{m_i} < key_{attr_c}, value_c > \tag{4}$$

where $attr \in aI$

We also introduce i-SmallClient to initialize index creation process in parallel to block creation which reduces cost of loading data blocks into memory for index creation. User provides list of attributes for index creation along with disk location of data files. In this situation, block creation module of SmallClient takes data as input, reads records line by line to create data blocks and sends each records to index creation module which extracts two tuple $< key, value >$ pair from that record. i-SmallClient saves the time to load data blocks and perform record reading operation and therefore it is better than SmallClient when index attributes are known at the time of data uploading.

3.3 Query Execution

The final and decisive module of our indexing framework is query execution. Based on results taken from the execution of queries, we will be able to show the performance of data search operation using SmallClient and how much the ultimate goal of indexing is achieved. As we have described earlier that indexing, which plays a significant role in big data processing, results in some overhead. However, the search performance gains from indexing must be more than the overhead caused by index creation process. Here, we present our query execution module that utilizes indexes created during index creation stage. This module retrieves data (i.e. sel data) for those queries having the same attributes as selection predicates (i.e. index attr list) for which indexes were previously created.Query execution module performs indexed search and retrieves both indexed and non-indexed attributes successfully. The fault-tolerance and availability of data blocks depends upon underlying file system. Therefore, query execution module of SmallClient exhibits successful execution as long as stored data blocks and indexes are accessible from the underlying file system. Furthermore, the accomplishment of query execution and data retrieval process using this module is subject to selection predicates in queries. Queries having non-indexed attributes as selection predicates are not served by query execution module.

Algorithm 4 runQuery(*query*)

```
 1: if analyze($query$) is not successful then
 2:     write $err\_message$
 3:     exit
 4: end if
 5: get provided $file\_name$ from $query$
 6: get $sel\_data\_list$ from $query$
 7: get $sel\_data\_offset\_list$ from $file\_schema$
 8: get $selection\_predicates$ as $attr\_list$ from $query$ to load
    respective $indexes$
 9: if $indexes$ are not available for $attr\_list$ then
10:     go for $full\_scan()$
11: else
12:     get $attr\_value\_list$ as $keys$ from $query$
13:     get $block\_locations$
14:     for all $blocks$ do
15:         load respective $index(es)$
16:         search $keys$ & fetch $sel\_data\_list$ if $keys$ are found
17:     end for
18: end if
```

Algorithm 4 describes the process of executing queries using indexes. Initially, we analyze incoming query to validate its syntax and verify the parameters specified in query. The system discards queries having typos, syntax error or not matching any file residing in file system after displaying a respective error message. With successful analysis of query string, we ensure the availability of indexes for a query (full scan operation is recommended only when indexes are not available for selection predicates of a query). We load and traverse the respective indexes to find location of records. Finally, the data is fetched from file by directly accessing the location of expected records. The results to show the performance of search operation are presented in next section.

4 Evaluation

In this section, we demonstrate a detailed evaluation of experiments for our work as discussed in previous section. We prove that SmallClient which is an indexing client for big data reduces indexing overhead and the delay between data uploading and starting query execution. We show that index hit ratio of our indexing framework is higher than state-of-the-art clustered index mechanisms. Furthermore, indexing overhead is less than available non-clustered indexing approaches. We evaluate the performance of SmallClient presented in this paper and compare it with indexes designed using Apache Lucene library [28] to measure indexing overhead and search performance. Moreover, we compare the performance with full scan approach of MapReduce programming model [29] adopted by Hive [30]. In addition, we measure the performance of block creation module by comparing it with data uploading process of Hadoop Distributed File System (HDFS) [31]. As far as index hit ratio is concerned, we compare our non-clustered SmallClient indexing with HAIL, a clustered approach presented to create one index on each replica. Precisely, along with our proposed indexing framework we measure the performance of following systems: (1) HDFS and Hive for block creation and full scan, (2) Apache Lucene for indexing overhead and search performance and (3) HAIL for index hit ratio.

All the modules and relative algorithms proposed in this paper are developed in java using Eclipse IDE under latest Ubuntu stable release. We use a physical four-node cluster where each node has 8GB RAM, 4x250GB Hard Drives and Processor. The size of records in chosen datasets for experimental evaluation is much smaller than default block size of HDFS. Therefore we used default block size of HDFS in block creation phase (replication factor is also set to default). However, block size is still customizable according to nature of datasets and replication factor is also adjustable in our system. We use default node selection policy of Hadoop to access data where Hive will execute full scan queries using MapReduce processing and SmallClient will use indexes for query execution. In order to evaluate indexing overhead and search performance we used up to five attributes from each dataset to create indexes on Lucene and the same attributes are used by SmallClient so that exact differences can be found. Finally, our objective is to increase index hit ratio so that maximum incoming queries are served by using indexes. Therefore, we calculate and compare index hit ratio of SmallClient with HAIL.

In order to see the effect of increasing size of data, we use varying size real datasets downloaded from SpatialHadoop repository [32], which are originally extracted from US Census Bureau TIGER files. The schema provided along with each dataset is used by algorithms to verify input lists of attributes for indexing and for queries. We downloaded 10 datasets from repository with varying size data and varying number of records. These features of a dataset effect data upload overhead, indexing overhead and ultimately search performance. However, number of attributes is used to present index hit ratio when these attributes have equal probability to be queried. We chose datasets with varying values for these features in our experiment to show the effect of dataset size i.e. volume of data. Moreover, we show No. of blocks for each dataset when uploaded via HDFS in Table 2. SmallClient also creates same No. of blocks when data is uploaded by using block creation module. We present the details of used datasets in Table 2. Table 2 presents that we have included small datasets as well as big datasets in evaluation of SmallClient. SmallClient has performed well for small datasets. Furthermore, the objective of designing an indexing framework with less indexing overhead and improved search performance is efficiently achieved for big datasets. The results show that SmallClient outperforms better existing methods for larger volume big text datasets.

These datasets are downloaded and extracted at local disk. There is no preprocessing required to upload these datasets and thus we input dataset location at local disk to our block creation module to create blocks for these datasets and upload blocks on underlying file system. We execute index creation to create indexes which verifies provided index attributes with available schema of a dataset. We finally execute selection queries to gather query execution time results. SmallClient supports following selection queries:

1. $SELECT\ attr\ FROM\ data\ WHERE\ attr = value$

Table 2 Datasets

Datasets	Data Size (MB)	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 (I)	16,220	19291957	37	260
Linear Hydrography	18,270	5857442	11	293
All Edges Combined (II)	23,180	70000000	24	363
All Edges Combined (III)	62,000	72700000	37	969
All Nodes	96,000	2700000000	4	1500
Road Network	137,000	717000000	9	2141

2. $SELECT\ attr\ FROM\ data\ WHERE\ attr1 = value1$ $OR\ attr2 = value2$
3. $SELECT\ attr\ FROM\ data\ WHERE\ attr1 = value1$ $AND\ attr2 = value2$
4. $SELECT\ *\ FROM\ data\ WHERE\ attr = value$

Our SmallClient query execution module efficiently works for each of these queries and we are able to gather query execution time results when any of these queries is executed. However, indexes created using existing Apache Lucene library have certain limitations in selecting attributes for queries. Unlike SmallClient, these indexes store offset of an attribute value instead of storing offset of a record. Therefore, indexes only contain offset addresses of indexed attributes and only indexed attributes can be accessed using Lucene indexes. Due to this reason, accessing whole record (i.e. $Query : 4$) or accessing non-indexed attribute is not possible with Lucene indexes. In our setup for SmallClient, we are creating indexes of up to five attributes whereas Lucene needs all attributes to be indexed in order to retrieve whole record. We use *sel_data_list* for $data\ attr(s)$ to be retrieved from file and the $attr(s)$ provided as selection predicate(s) will be used to decide query execution with full scan or using indexes if they are available. The parameter values provided in selection predicate will be used as keys which are to be searched in indexes.

4.1 Data Upload Overhead

As we have already discussed that state-of-the-art big data processing systems offer their own data storage plan. We are using Hadoop framework which has HDFS to store data. Configurable block size and replication factor can be used for uploading files on HDFS. However, in the process of creating blocks of provided block size for a file, HDFS simply splits the data bytes to fill up the container named as block. In this way, the last record of each block is split which results in increased time to retrieve one record when it resides on more than one physical location. For this reason, we introduce our block creation module with never splitting records policy. At the same time, block creation and data uploading should have minimum overhead while using data-intensive systems. This overhead increases the delay to start query execution [15]. Fig. 6 shows the results of data uploading size overhead whereas Fig. 7 shows the results of data uploading time. We observe that SmallClient has negligible size overhead over standard HDFS which is decreasing (~1%) for larger size datasets (Eq. 5 presents the size of data when it is uploaded with our block creation module and Eq. 6 shows the size overhead). Data upload time is also decreasing when we have larger size files and the overhead is also minimized (~11%). Data upload time and overhead are shown in Eq. 7 and 8

$$S_{D'} = k \times l \tag{5}$$

where $S_{D'} - S_D < l$

$$O_{datasize} = \frac{S_{D'} - S_D}{S_D} \times 100 \tag{6}$$

$$T_{blockCreation} = \sum_{c=1}^{k} (T_{create}(B'_c) + T_{upload}(B'_c)) \tag{7}$$

$$O_{dataupload} = \frac{T_{blockCreation} - T_{uploadData}}{T_{uploadData}} \times 100 \tag{8}$$

The results of dataset size as presented in Fig. 6 show that dataset size remains almost same when we upload data by using SmallClient. Size overhead results show that the overhead becomes negligible for large size datasets which proves better data upload performance of SmallClient for large size datasets. Similarly, data upload time overhead is also reduced for large size datasets. However, there are several factors effecting data upload time. For instance, number of records in a dataset effects data upload time of SmallClient. HDFS adopts conventional policy to create blocks and fills block container with required number of bits from

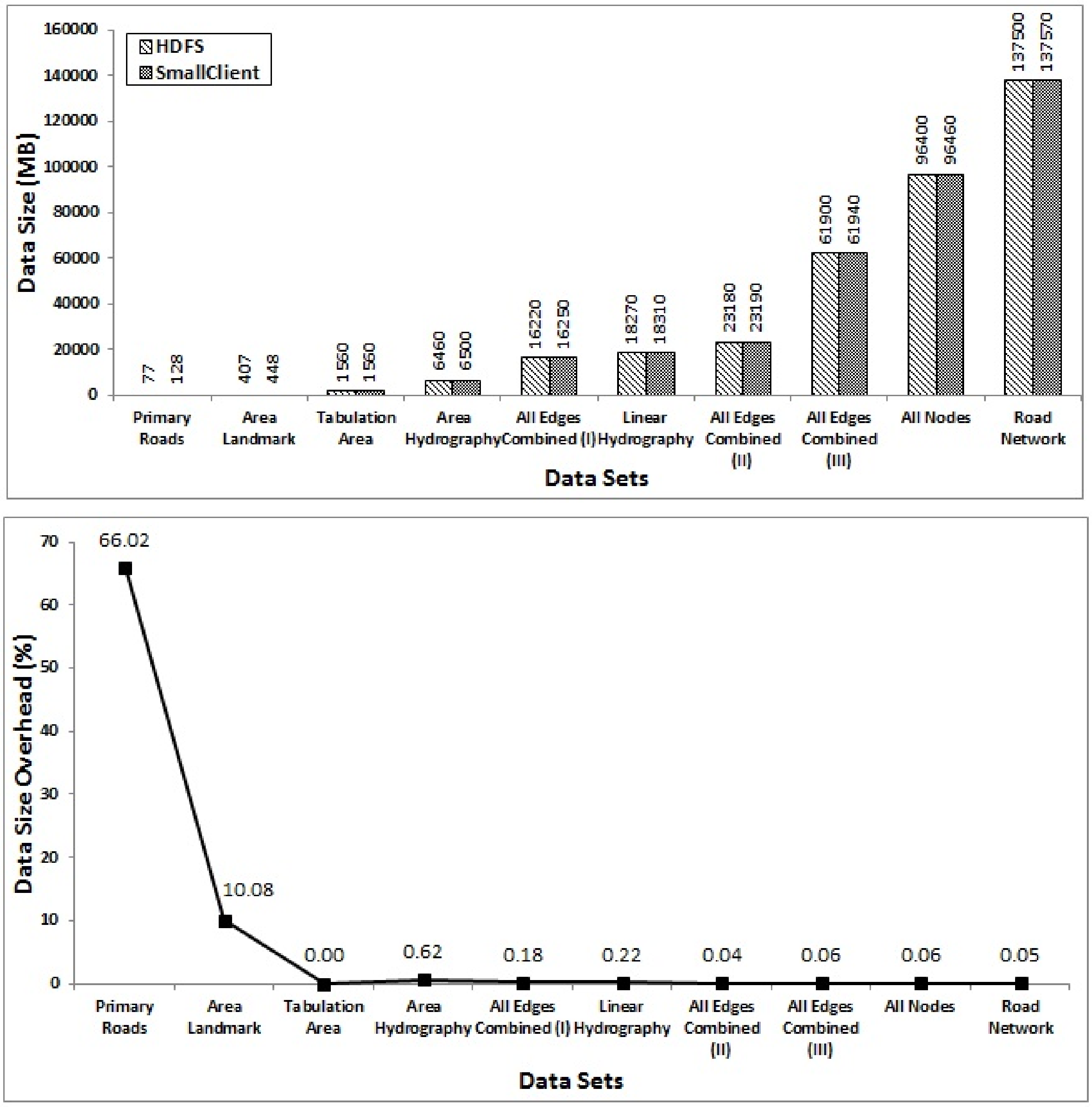
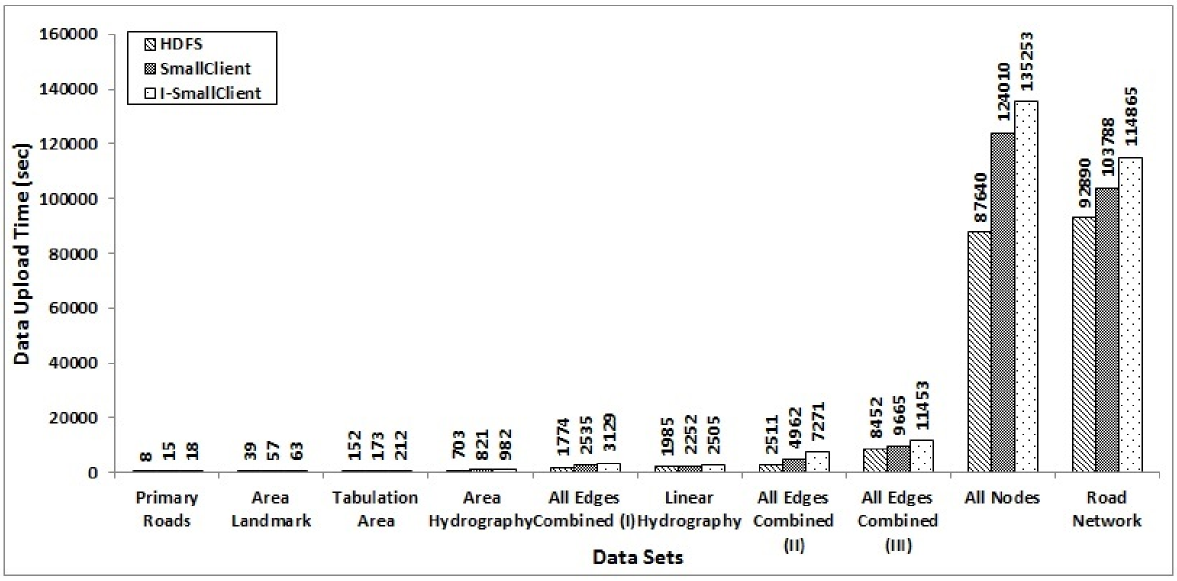


Fig. 6 Data Size and Size Overhead by SmallClient

data as specified in block size. The purpose of designing block creation module in SmallClient is not only to manage contiguous records in the form of a block but also to avoid record split. Therefore, during block creation, each record of a dataset is read which shows that a dataset having more number of records faces more data uploading overhead than other datasets. However, intervening data uploading overhead by SmallClient is only one time and results in data blocks where records are not split. Consequently, the record access delay caused by record split is avoided by SmallClient.

4.2 Indexing Overhead

We will first observe and compare the overhead of index size and later we see the index creation time for both Lucene and SmallClient. For this purpose, we consider one index and then see the effect of increasing number of indexes. We vary the number of indexes from 1 to 5 using both Lucene and SmallClient. However, we did not use Lucene to create indexes for our last dataset (out of memory error after ˜30 mins). The results of index size presented in Fig. 8 show that indexes created using SmallClient are smaller in size than the indexes which are created using Lucene. Thus, indexing overhead in terms of size is clearly reduced for SmallClient (˜50%).



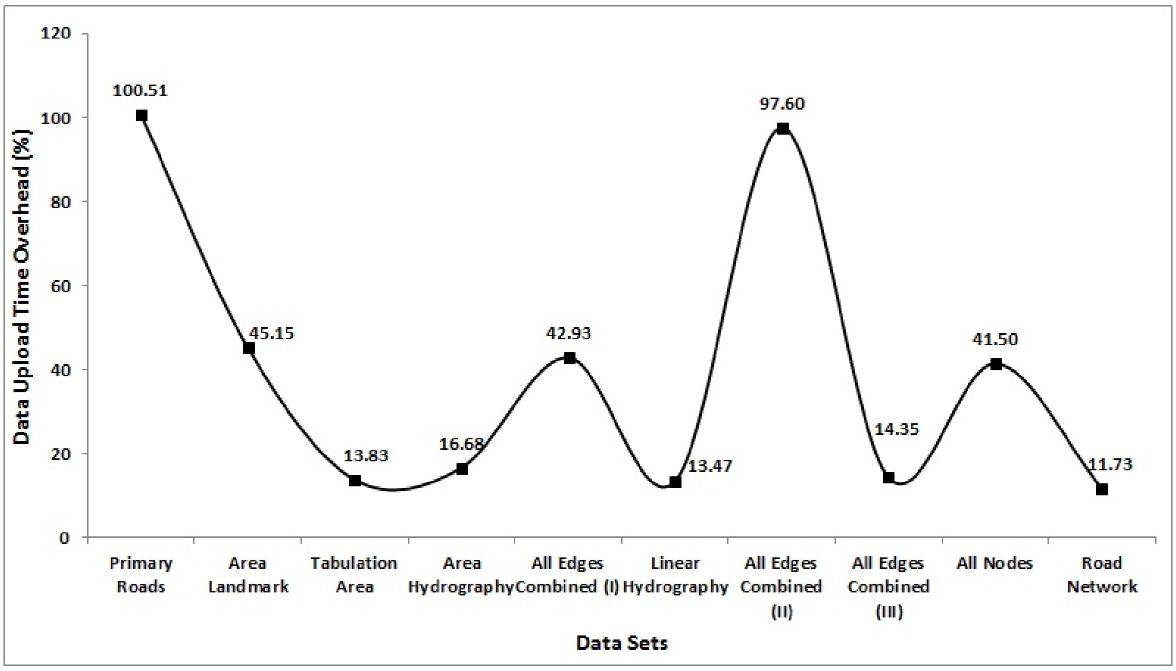


Fig. 7 Data Upload Time and Upload Time Overhead by SmallClient

The second observation to evaluate indexing overhead is index creation time. Fig. 9 presents the results of time consumed in creating one index for Lucene and SmallClient. SmallClient outperforms Lucene in index creation and takes upto 85% less time than Lucene. Another important result which is observed during creating more than one index using Lucene and SmallClient is shown in Fig. 10 as index creation overhead. When up to five indexes are created, SmallClient takes very less time than Lucene. The results of index size for up to five indexes as presented in Fig. 8 show that size of indexes grows with size of dataset and with number of index attributes. However the size for SmallClient indexes is smaller than Lucene indexes. Index size overhead shows that the overhead of indexing on dataset size is less than 15% for all datasets except Edges (II) dataset. The number of records in a certain size dataset plays a significant role in index size overhead. Edges (II) dataset contains large number of records and thus index size overhead is also very high. Other datasets, for instance, Edges (III) dataset has almost same number of records as Edges (II) dataset whereas the size of Edges (III) dataset is much larger than Edges (II)

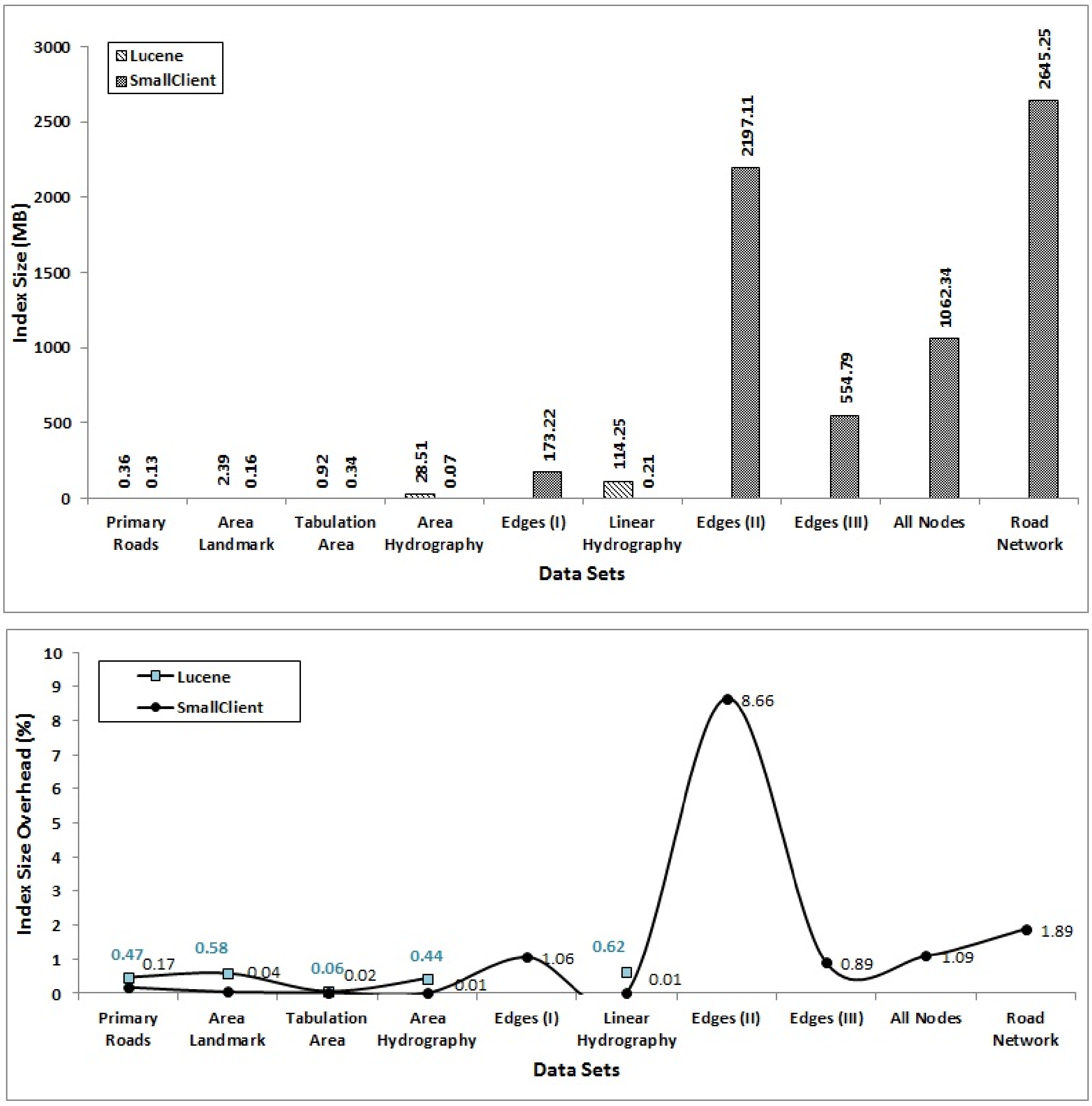


Fig. 8 Index Size Overhead by Lucene and SmallClient

dataset. Therefore, index size overhead is increased for Edges (II) dataset.

Similarly, indexing time results show that indexing time varies with dataset size and number of index attributes (see Fig. 9 and Fig. 10). SmallClient achieves reduced indexing time when same indexes are created using Lucene library. However the effect of number of number index attributes on indexing time is less than dataset size. Furthermore, the indexing time with i-SmallClient is much improved as i-SmallClient creates indexes during block creation and thus time required to access data blocks and loading into memory is saved. Indexing time overhead in Fig. 9 shows that there is a clear difference between Lucene and SmallClient indexing overhead. SmallClient reduces indexing overhead from $40-95\%$ to $15-35\%$. The indexing overhead results also show that the overhead is high for dataset having large number of records (i.e. All Edges Combined (II) dataset).

4.3 Search Performance

Our third objective is to improve search performance of executing queries on different size datasets. Hive executes full scan operations to retrieve results from stored data. For this purpose, MapReduce model divides and combines data search task on more than one site to per-

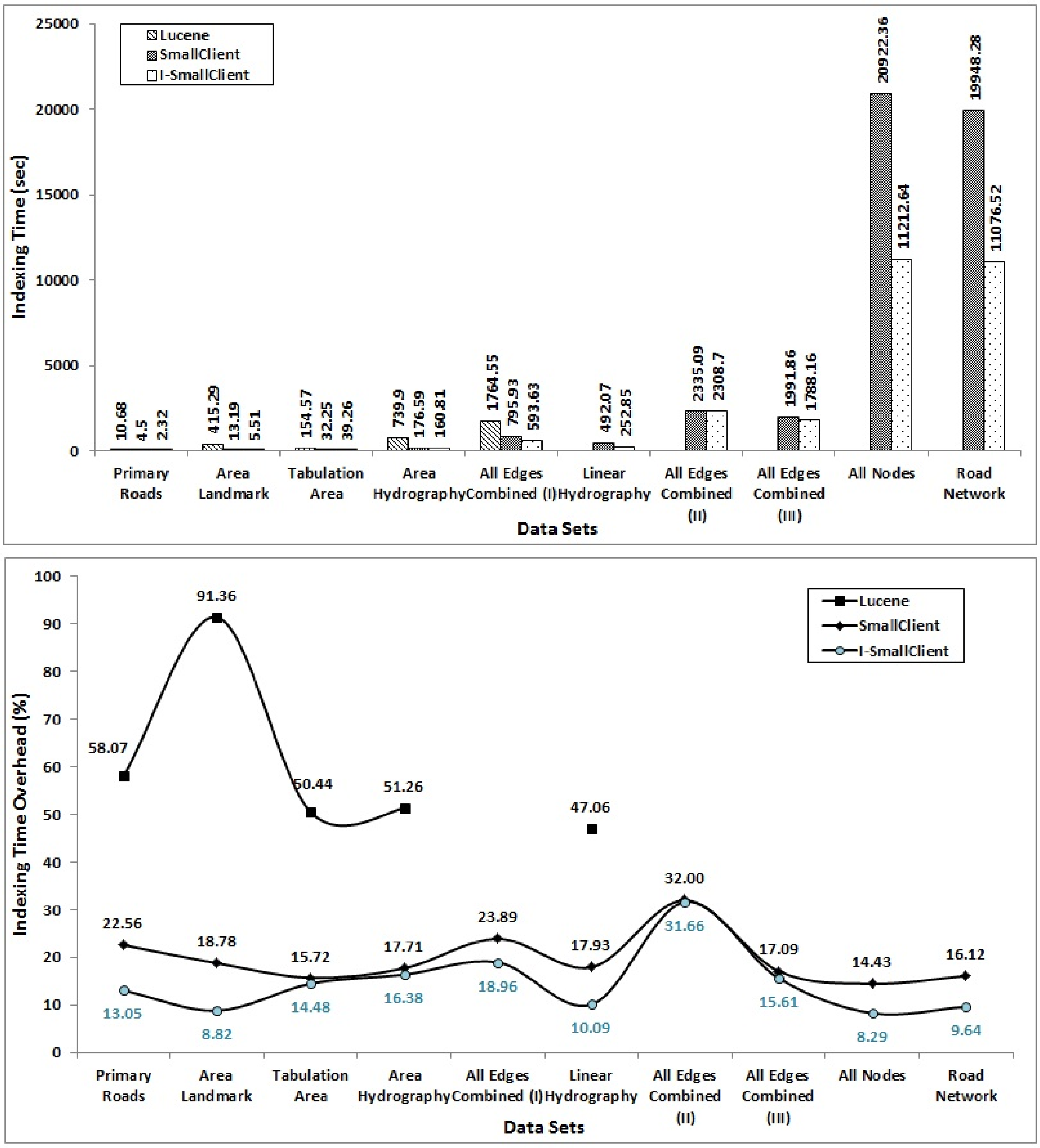


Fig. 9 Indexing Overhead by Lucene and SmallClient

form parallel execution and to minimize search time. However, full scan takes longer time (i.e. TFS) for large datasets. Therefore, indexing is intended to implement on file systems so that full scan operations are avoided. We present the results of executing same queries on Hive using full scan and using indexes created with Lucene and SmallClient. Fig. 11 shows the results of query execution and search performance improvement achieved by indexes. Query execution time using indexes TQ comprises of time to traverse an index TT at all k blocks and to fetch data TF from the position(s) obtained in *value* from traversing the index (as shown in Eq. 9). There is a remarkable difference between query execution times with and without indexing. At the same time, SmallClient executes queries faster than Lucene. The search performance (in Eq. 10 where P_{search} denotes search performance) gains of both Lucene and SmallClient are shown in Fig. 11. Although the difference becomes very small yet overall, SmallClient has achieved improved search performance.

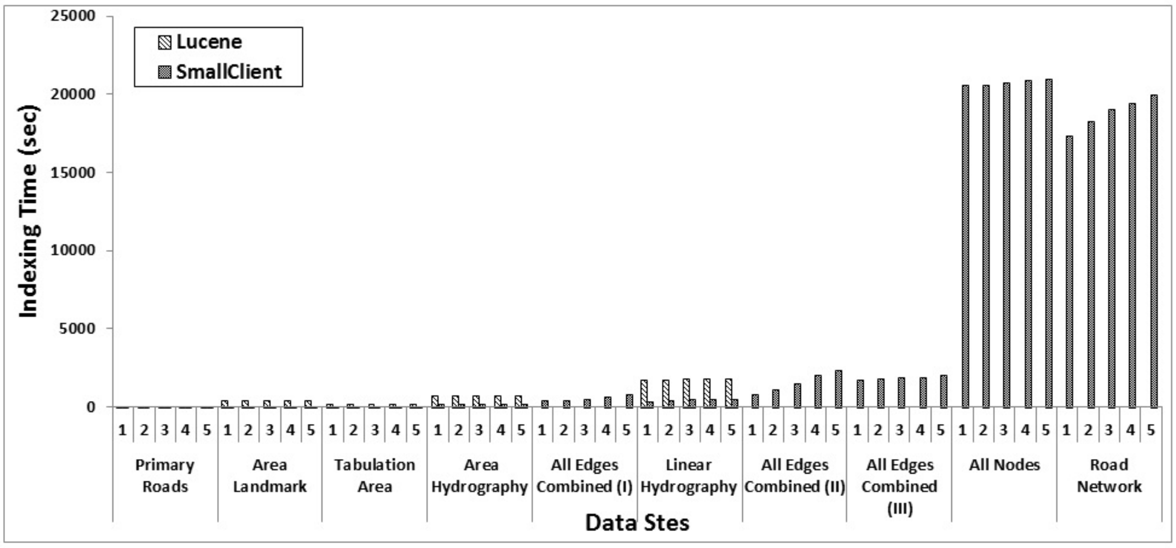


Fig. 10 Indexing Overhead for up to five indexes by Lucene and SmallClient

$$TQ = \sum_{c=1}^{k}(TT_{attr,c}(key) + TF_{sel\_data}(value)) \qquad (9)$$

where $TT_{attr,c}(key) = O(log_n)$ and $TF_{sel\_data}(value) = S_{value}$

$$P_{search} = \frac{TFS - TQ}{TFS} \times 100 \qquad (10)$$

4.4 Index Hit Ratio

Another significant benchmark in our experimental evaluation is index hit ratio. Index hit ratio describes the probable rate of incoming queries which are executed using indexes. This ratio can only be increased when it is possible to create more indexes. We have set of attributes provided as schema of data set i.e. aD from which we have chosen up to five attributes for index creation. We denote the set of index attributes as aI. Furthermore, S_{aD} and S_{aI} denote the size of aD and aI respectively. We define index ratio mathematically as below:

$$HR = \frac{S_{aI}}{S_{aD}} \times 100 \qquad (11)$$

HAIL, an indexing library presented in [14], uses clustered approach to create indexes. In this way, HAIL depends upon replication factor to create indexes and it is impossible to create indexes more than available replication factor. Meanwhile, whenever there is a need to create new index, whole dataset is replicated once more. In contrast, SmallClient is not constrained to replication factor of a dataset or file system to increase number of indexes. In case of SmallClient, indexes are separate small objects which can be created any time when user demands to execute his queries on some attributes as selection predicates. We have already discussed indexing overhead of increasing number of indexes in Fig. 9. Now we present the effect of number of indexes over index hit ratio in Fig 12. The grey area shows where index hit ratio becomes static for HAIL when default replication factor is used which is 3 in HAdoop. With default replication factor, HAIL cannot create more than three indexes and index hit ratio becomes constant after three indexes. While, SmallClient can create more indexes and thus index hit ratio is growing with number of indexes.

5 Discussion

We introduce SmallClient as an indexing framework for big text data to improve performance of indexing and search performance for large volume datasets. SmallClient focuses on growing volume and velocity of processing big datasets. This framework improves query execution and data search performance, and offers maximum number of indexes for datasets regardless of number of replicas. SmallClient also ensures minimized index creation overhead in terms of both index size and indexing time. We present the results obtained from execution of several modules of SmallClient which are: block creation, index creation and query execution. We use varying size datasets in our experiments and the results show that though processing time of SmallClient for large datasets is high, the overall indexing overhead is not high. Similarly, search performance results show

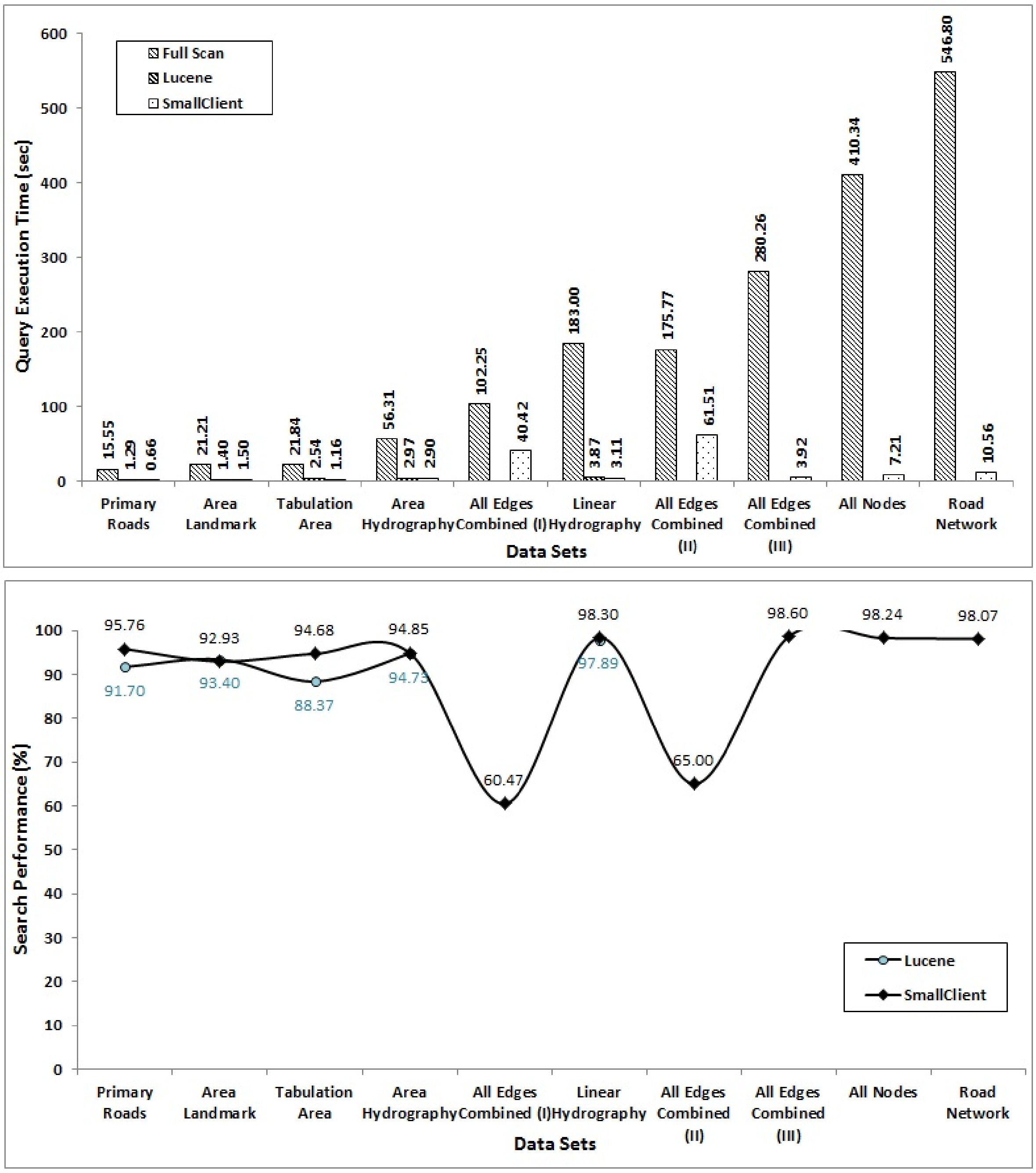


Fig. 11 Query Execution and Search Peroformance using SmallClient

that SmallClient is efficient not only for small datasets (i.e. 77.1MB) but also for big datasets (i.e. 137,000MB). However, the features of datasets such as number of records, number of attributes in schema affect the performance of an indexing and query execution framework.

Block creation module introduces procedure to create blocks having contiguous records instead of bits which minimizes cost of accessing broken records. The data uploading overhead with block creation module is very low on both dataset size and data uploading time. Dataset size overhead become negligible for large volume of data (i.e. less than 1%) whereas data uploading time overhead also decreases with size of data. Data uploading time of block creation not only depends on size of dataset but also on number of records. Datasets having large number of records, for instance Edges (II) dataset, have high data upload overhead.

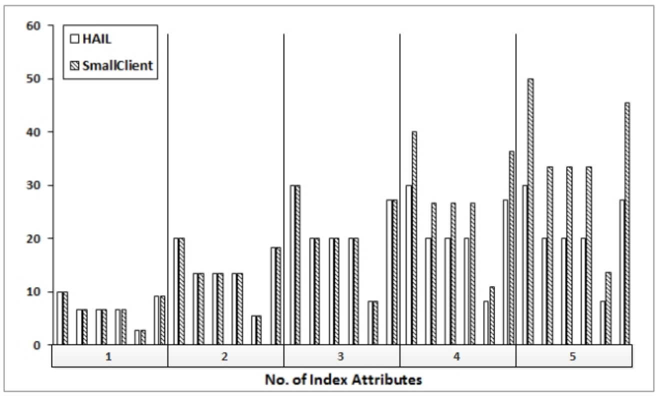


Fig. 12 Index Hit Ratio of HAIL and SmallClient

Index creation module achieves less indexing overhead for big datasets than Lucene indexing library. The results show that indexing depends on dataset size and number of records in a dataset. Both indexing time and index size are high for large size datasets and for large number of records of a dataset. Index size also increases with number of index attributes whereas indexing time slightly increases with number of index attributes. Indexing time is reduced when indexes are created during data uploading (i.e. i-SmallClient). Overall indexing overhead of SmallClient is less than Lucene indexes.

The results of query execution module present the achievement of improved search performance of SmallClient. Query execution time of SmallClient is less than contemporary Lucene library and overall search performance is improved. We execute same queries using both indexes and found that SmallClient takes less time in data search and retrieval. Another advantage of SmallClient is that, indexes store the address of record instead of instance of an attribute in a record which makes it possible to access non-indexed attributes via queries.

We also present that SmallClient improves index hit ratio. We show that with replication factor of 3, HAIL allows three indexes. However, SmallClient is independent of replication factor settings. SmallClient offers as much indexes on a dataset as needed. Therefore, for upto three indexes in our experimental setup, index hit ratio of HAIL and SmallClient are same. For increased number of indexes SmallClient keeps improving index hit ratio whereas HAIL become unable to create indexes more than number of replicas.

Precisely, SmallClient indexing framework efficiently works for small and big datasets with minimum data upload and indexing overhead in terms of size and time. The evaluation proves that both increasing volume and velocity in processing for big data are well handled by SmallClient. SmallClient outperforms in index creation and offers maximum number of indexes for a dataset as SmallClient works independent of number of replicas. SmallClient also exhibits decreased query execution time and improved search performance when compared with existing Apache Hive and Apache Lucene searches.

6 Conclusion and Future Work

We have presented SmallClient to provide non-clustered indexing solution for big data. SmallClient introduces block creation mechanism such that last record of each block can be accessed from single site. Consequently, SmallClient offers customizable block size and replication factor and thus becomes a generalized indexing framework with faster data access. Besides data upload policy, SmallClient, outperforms in index creation when compared with Lucene indexing library and shows minimized delays between data upload and starting query execution. Apart from faster index creation, size of SmallClient indexes is also smaller than Lucene indexing approach. Furthermore, query execution and data search time is drastically reduced by SmallClient when compared with full sequential scan behavior Hadoop MapReduce processing framework and is evidently less than Lucene. Another noteworthy achievement of SmallClient over Lucene is that, Lucene needs all attributes to be indexed in order to retrieve whole records. However, this is not the case with SmallClient. It is possible with SmallClient to retrieve whole data records even when only one attribute is indexed. The evaluation on small and big datasets has proved that the performance of SmallClient improves with increasing volume of data. We have also compared our indexing client with HAIL, which is a clustered approach of indexing, and show that SmallClient offers more feasible index creation mechanism than HAIL. Therefore, index hit ratio can easily be improved with SmallClient. As far as size overhead of separate indexes associated with non-clustered mechanism is concerned, crating small index objects to serve more queries is preferable to creating replicas of data which is the only option HAIL has.

As a future work, we are working to implement probabilistic machine learning algorithm in collaboration with B-Tree indexing. The probabilistic modeling in our work aims to achieve adaptive index creation through predicting query workload and updating index attribute set accordingly. Furthermore, we are implementing Disaster Recovery [33] to improve fault-tolerance of big data distributed storage systems and ensure restoring.

References

1. Vera-Baquero, A., Colomo-Palacios, R., Molloy, O.: Measuring and Querying Process Performance in Supply Chains: An Approach for Mining Big-Data Cloud Storages. Procedia Computer Science 64, 1026-1034 (2015).
2. Suthaharan, S.: Big Data Analytics. In: Machine Learning Models and Algorithms for Big Data Classification, vol. 36. Integrated Series in Information Systems, pp. 31-75. Springer US, (2016).
3. Karim, A., Salleh, R., Khan, M.K., Siddiqa, A., Choo, K.-K.R.: On the Analysis and Detection of Mobile Botnet Applications. Journal of Universal Computer Science 22(4), 567-588 (2016).
4. Ahmad Karim, S.A.A.S., Rosli Salleh, Muhammad Arif, Rafidah Md Noor, Shahaboddin Shamshirband: Mobile Botnet Attacks an Emerging Threat: Classification, Review and Open Issues. KSII Transactions on Internet and Information Systems 9(4) (2015).
5. Yaqoob, I., Chang, V., Gani, A., Mokhtar, S., Hashem, I.A.T., Ahmed, E., Anuar, N.B., Khan, S.U.: Information fusion in social big data: Foundations, state-of-the-art, applications, challenges, and future research directions. International Journal of Information Management (2016).
6. Hashem, I.A.T., Chang, V., Anuar, N.B., Adewole, K., Yaqoob, I., Gani, A., Ahmed, E., Chiroma, H.: The role of big data in smart city. International Journal of Information Management 36(5), 748-758 (2016). doi:http://dx.doi.org/10.1016/j.ijinfomgt.2016.05.002
7. Kambatla, K., Kollias, G., Kumar, V., Grama, A.: Trends in big data analytics. Journal of Parallel and Distributed Computing 74(7), 2561-2573 (2014).
8. Siddiqa, A., TargioHashem, I.A., Yaqoob, I., Marjani, M., Shamshirband, S., Gani, A., Nasaruddin, F.: A Survey of Big Data Management: Taxonomy and State-of-the-Art. Journal of Network and Computer Applications (2016).
9. Siddiqa, A., Karim, A., Gani, A.: Big data storage technologies: a survey. Frontiers of Information Technology & Electronic Engineering 1 (2016).
10. Chang, V., Wills, G.: A model to compare cloud and non-cloud storage of Big Data. Future Generation Computer Systems 57, 56-76 (2016).
11. Lomotey, Richard K., and Ralph Deters.: Unstructured data mining: use case for CouchDB. International Journal of Big Data Intelligence 2, no. 3 (2015): 168-182.
12. Yu, Shanshan, Jindian Su, Pengfei Li, and Hao Wang. "Towards High Performance Text Mining: A TextRank-based Method for Automatic Text Summarization." International Journal of Grid and High Performance Computing (IJGHPC) 8, no. 2 (2016): 58-75.
13. Yu, Kun-Ming, Sheng-Hui Liu, Li-Wei Zhou, and Shu-Hao Wu. "Apriori-based High Efficiency Load Balancing Parallel Data Mining Algorithms on Multi-core Architectures." International Journal of Grid and High Performance Computing (IJGHPC) 7, no. 2 (2015): 77-99.
14. Dittrich, J., Quian, J.-A., Quian-Ruiz, Richter, S., Schuh, S., Jindal, A., Schad, J.: Only aggressive elephants are fast elephants. Proc. VLDB Endow. 5(11), 1591-1602 (2012).
15. Idreos, S., Alagiannis, I., Johnson, R., Ailamaki, A.: Here are my Data Files. Here are my Queries. Where are my Results? In: Proceedings of 5th Biennial Conference on Innovative Data Systems Research, No. EPFL-CONF-161489 2011, vol. EPFL-CONF-161489 (2011).
16. Gandomi, A., Haider, M.: Beyond the hype: Big data concepts, methods, and analytics. International Journal of Information Management 35(2), 137-144 (2015).
17. Richter, S., Quian-Ruiz, J.-A., Schuh, S., Dittrich, J.: Towards zero-overhead adaptive indexing in Hadoop. arXiv preprint arXiv:1212.3480 (2012).
18. Idreos, S., Kersten, M.L., Manegold, S.: Database Cracking. In: CIDR 2007, pp. 1-8 (2007)
19. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 165-178 (2009).
20. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. 2(1), 922-933 (2009).
21. Jens, D., Jorge-Arnulfo, Q.-r., Alekh, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah. In: (2010).
22. Zhuang, Y., Jiang, N., Wu, Z., Li, Q., Chiu, D.K.W., Hu, H.: Efficient and robust large medical image retrieval in mobile cloud computing environment. Information Sciences 263, 60-86 (2014).
23. Wang, M., Holub, V., Murphy, J., OSullivan, P.: High volumes of event stream indexing and efficient multi-keyword searching for cloud monitoring. Future Generation Computer Systems 29(8), 1943-1962 (2013).
24. Kaushik, V.D., Umarani, J., Gupta, A.K., Gupta, A.K., Gupta, P.: An efficient indexing scheme for face database using modified geometric hashing. Neurocomputing 116, 208-221 (2013).
25. Gani, A., Siddiqa, A., Shamshirband, S., Hanum, F.: A survey on indexing techniques for big data: taxonomy and performance evaluation. Knowl Inf Syst 46(2), 241-284 (2016).
26. Jin, R., Cho, H.-J., Chung, T.-S.: A group round robin based b-tree index storage scheme for flash memory devices. Paper presented at the Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication, Siem Reap, Cambodia (2014).
27. Chi, P., Lee, W.-C., Xie, Y.: Making B¡sup¿+¡/sup¿-tree efficient in PCM-based main memory. Paper presented at the Proceedings of the 2014 international symposium on Low power electronics and design, La Jolla, California, USA (2014).
28. McCandless, M., Hatcher, E., Gospodnetic, O.: Lucene in Action: Covers Apache Lucene 3.0. Manning Publications Co., (2010).
29. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107-113 (2008).
30. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment 2(2), 1626-1629 (2009).
31. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on 2010, pp. 1-10 (2010).
32. Eldawy, A., Mokbel, M.F.: Spatial Hadoop: A MapReduce Framework for Spatial Data. In: 2015 IEEE 31st International Conference on Data Engineering 2015, pp. 1352-1363. IEEE:1352-1363 (2015).
33. Chang, V.: Towards a Big Data system disaster recovery in a Private Cloud. Ad Hoc Networks 35, 65-82 (2015). doi:http://dx.doi.org/10.1016/j.adhoc.2015.07.012
34. McCandless, Michael, Erik Hatcher, and Otis Gospodnetic: Lucene in Action: Covers Apache Lucene 3.0. Manning Publications Co., (2010).