On the analysis of big data indexing execution strategies

Aisha Siddiqa[a,*], Ahmad Karim[b] , Tanzila Saba[c] and Victor Chang[d]
[a]*Department of Computer System & Technology, University of Malaya, 50603, Kuala Lumpur, Malaysia*
[b]*Department of Information Technology, Bahauddin Zakariya University, 60000, Multan, Pakistan*
[c]*College of Computer and Information Sciences, Prince Sultan University, 11586, Riyadh, King Saudi Arabia*
[d]*IBSS, Xi'an Jiaotong Liverpool University,100044, Suzhou, China*

Abstract. Efficient response to search queries is very crucial for data analysts to obtain timely results from big data spanned over heterogeneous machines. Currently, a number of big-data processing frameworks are available in which search operations are performed in distributed and parallel manner. However, implementation of indexing mechanism results in noticeable reduction of overall query processing time. There is an urge to assess the feasibility and impact of indexing towards query execution performance. This paper investigates the performance of state-of-the-art clustered indexing approaches over Hadoop framework which is de facto standard for big data processing. Moreover, this study leverages a comparative analysis of non-clustered indexing overhead in terms of time and space taken by indexing process for varying volume data sets with increasing Index Hit Ratio. Furthermore, the experiments evaluate performance of search operations in terms of data access and retrieval time for queries that use indexes. We then validated the obtained results using Petri net mathematical modeling. We used multiple data sets in our experiments to manifest the impact of growing volume of data on indexing and data search and retrieval performance. The results and highlighted challenges favorably lead researchers towards improved implication of indexing mechanism in perspective of data retrieval from big data. Additionally, this study advocates selection of a non-clustered indexing solution so that optimized search performance over big data is obtained.

Keywords: Big Data, Indexing, Big Data processing, Data retrieval

1. Introduction

Big Data refers to collection of huge data sets with a great diversity in types so that it becomes difficult to process by state-of-the-art data processing approaches or platforms [21]. More generally, we can say that it is formidable to perform capture, preparation, analysis and visualization on big data by current technologies. Therefore, big data introduces new challenges for security[14, 23], processing and analytics such as quick and up-to-date responses of a search query and in-time availability of data [24]. For instance, data obtained from sensor networks like urban management, environment and industrial installation [17] introduces storing, cleansing, query execution and other challenges like security, visualization and analytics[15, 22]. Similarly in the field of body sensor networks, increasing costs of healthcare and ageing of population are major subjects for researchers which have critical information retrieval requirements. For these time sensitive applications, efficiency in query execution and analyzing data is very important for faster decision making [5]. Need of fast data processing and timely responses derive to evaluate the performance of search process so that challenges revealed by the emergence of big data can be highlighted.

Indexing is a significant activity even for distributed highly available big data sets to efficiently perform data retrieval operations [8]. It is impractical to apply full scan on millions of records to accomplish search of a specific result [9]. Therefore, efficient techniques are required to improve task execution for

* Corresponding author, Aisha Siddiqa, Department of Computer System & Technology, University of Malaya, 50603, Kuala Lumpur, Malaysia; Email: aasiddiqa@gmail.com

accessing big data. To improve the efficiency of search and data retrieval process for voluminous data records many solutions have been proposed by researchers. For example, vertical partitioning [12], clustered attribute based indexing [6, 9] for distributed parallel processing systems and clustered adaptive indexing [18] for changing query workload. Likewise in medical research, large distributed image data sets face the problem of multi-query optimization and a batch processing based image retrieval system [25] contributes in scheduling multiple query requests and minimized response time is achieved. Consequently for distributed and replicated big data storage systems, an efficient indexing technique is needed to serve more number of queries for improved search performance.

As described above, many indexing frameworks are available to perform fast search operations on big data residing on distributed parallel systems. However, existing indexing frameworks still have unaddressed challenges. For instance, multi-attribute indexing on single site, indexing without intervening physical organization of data and re-indexing are some of the major challenges. Any state-of-the-art solution for big data indexing does not deal with these challenges. Furthermore, visualization of benefits obtained by indexing over high throughput big data processing technologies also lacks in literature. In this research, the achievements and unaddressed areas of indexing are presented to help researchers understand recent indexing advancements in the field of big text data. Significance of indexing is also visualized via experiment on sample text data sets which highlights the to-date indexing challenges in clearer context.

With research viewpoint, we theoretically evaluate clustered indexing mechanisms developed over Hadoop in static and dynamic categories. This investigation shows that creating new replicas for increased number of index attributes[†] is the main storage efficiency barrier. Meanwhile, our empirical analysis of non-clustered approach emphasizes on considering it for big text data indexing. Our main focus is to investigate the performance of big data indexing techniques for growing volume of data. In addition, the impact of involving as much data attributes as possible for index creation in terms of search performance and indexing overhead is also observed in this study. Our evaluation results are twofold: We show that increasing either size of data or number of index attributes has very minor effect on index size, thereby improving overall search performance. The reasoning on differences in performance results will further help researchers to propose a better indexing solution. The results obtained from experiment are further verified with mathematical modeling. We model the analysis approach using CPN tools which leverage Petri nets mathematical modeling language. Based on evaluation, we highlight the significance of indexing and identify the weaknesses of existing solutions which provide guidelines to researchers to explore improved indexing architectures for big data. Concisely, this paper contributes in following:

- Provides an overview of clustered indexing approaches under static and adaptive categories
- Investigates static and adaptive clustered indexing approaches and identifies their benefits and limitations on search throughput for big data
- Implements an in memory indexing approach under non-clustered mechanism on varying size data sets and varying number of index attributes to examine their impact on search performance
- Verifies the experiment results by designing mathematical model of analysis approach through Petri nets
- Based on analysis, this paper highlights challenges in the field of indexing for big data as a motivation for future research
- Finally the paper suggests the implementation of non-clustered approach for big data indexing to maximize Index Hit Ratio

The rest of the paper is organized as follows: Section 2 provides a study of existing clustered indexing approaches implemented over big data under static and adaptive category. Section 3 presents an experimental investigation of non-clustered indexing over varying size data sets with varying number of indexes. Section 4 discusses the results obtained and provides an illustration of clustered and non-clustered approaches. Section 5 highlights challenges and recommendations for future work in indexing implementation over big data. Finally, section 6 concludes the discussion.

2. Related Work

Fast query processing and data retrieval are the main challenges for large volume of data distributed over clusters of heterogeneous machines. Researchers are interested to accept the challenge and they have devoted to exploit different methods to optimize search performance for such big data. Clustered in-

[†] We refer attributes of a data set as its index attributes based on which indexes are created

dexing approaches are developed over Hadoop which is a de facto framework for big data processing. These approaches fall in either static or adaptive category according to invocation of index creation process and ability of changing number of index attributes. More explicitly, static indexes are created at data upload time and they do not allow increasing the number of index attributes once created. On the other hand, adaptive indexes are the side effect of query execution with the flexibility of as much index attributes as fed by incoming queries.

Clustered static indexes which are developed for Hadoop framework, offer indexing on single attribute - Trojan index [9] or varying number of index attributes – HAIL [6]. Indexes are created on whole data set in parallel with data uploading. Therefore, query execution process can be carried out immediately when a query is submitted as it does not invoke index creation or updating. However selection of attributes to be indexed should be very wise as these are the only indexes available throughout the data search process and cannot be updated later. Based on anticipated query workload knowledge better indexes are created. Queries having same selection predicate can be executed using static indexes otherwise full scan will be performed. In case of Trojan index, only one particular index is selected to be indexed whereas HAIL can extend number of indexes up to available number of replicas. We elaborate this concept in Eqs 1 and 2 for Trojan Index and HAIL respectively:

$$No.of\ Indexes = 1 \quad (1)$$

$$No.of\ Indexes = No.of\ Replicas \quad (2)$$

In contrast to static indexes, adaptive indexes do not offer pre-created indexes to incoming new queries. These indexes keep on updating with new queries and are being utilized by repeated queries. Data blocks are replicated for each new index attribute. Lazy Indexing (LIAH) is proposed by Richter, Quiané-Ruiz [18] as adaptive indexing using clustered approach. LIAH uses *offer rate* to minimize indexing I/O cost and creates as many indexes as suggested by incoming queries. However, future utilization of those indexes is still unpredictable. Similarly, from *offer rate* perspective, there exists a better tradeoff to minimize index creation overhead when offer *rate value* is set to low. Nevertheless to completely index all data blocks, *low offer* rate will require more MapReduce jobs. Due to this fact, LIAH has to compromise either indexing overhead or number of MapReduce jobs which provides motivation towards dynamically adapting *offer rate* [19]. Although query workload prediction is not required and unlike static indexing there is no replication factor dependency to consider number of index attributes in both of these approaches yet performing full scan for each new query and replicating data block for each new index attribute are the performance bottlenecks of LIAH. Therefore, the proposal by Schuh and Dittrich [20] is to drop the indexes from existing replicas and utilize these replicas for creating new indexes according to changing query workload.

We elaborate static and adaptive clustered indexing approaches in Table 1. Their method, success points and weaknesses are detailed in this table. Furthermore, Index Hit Ratio which is a significant efficiency measure for attribute based indexing is also described for each approach. Problems of static and adaptive indexing lead researchers as challenges and provide insight to come up with an optimum indexing solution for big data. Discussed challenges are the milestones for researchers on the basis of which they can formulate new research objectives towards development of improved indexing mechanism. Hence, efficiency in search operations over big data can be achieved in terms of reduced storage consumption and faster data retrieval from large distributed storage clusters.

Table 1 Existing Clustered Indexing Approaches

	Approach	Method	Achievements	Problems/Un-addressed	Index Hit Ratio
Static	Trojan Index [9]	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 [6]	Change physical data layout	–Reduced Index Miss Ratio up to number of replicas	–High index upfront cost	–In order to improve Index

		on each replica based on index attributes	–Upload cost is negligible by utilizing un-used CPU cycles –Full scan option is still valid for queries on non-indexed attributes	–No knowledge about query workload –Index Miss Ratio is still high –Indexes are replica dependent –Indexes may be unused by queries	Hit Ratio, more number of replicas are required
Adaptive	Lazy Indexing (LIAH) [18]	Indexing is the effect of query execution. Records in data block are reordered during.	–Adaptive to query workload –Query can be executed right after data upload –No Indexing upfront cost –Reduced indexing overhead because of selective block indexing and 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 –Constant offer rate either supports indexing overhead or number of MapReduce jobs to completely index all data blocks	–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
	Adaptive indexing - replace indexes [20]	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 [19]	Introduce cost model for LIAH with varying offer rate. Missing indexes 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 1 summarizes the existing clustered indexing approaches for big data. This illustration will further be used in discussion to provide comparison between the performances of clustered and non-clustered indexing. At present, we demonstrate a proportionate analysis of indexing overhead and impact of indexing over search performance in terms of Index Hit Ratio and compare the performance of search operation in both cases when a query hits index or misses it. We elaborate our analysis approach in next section. We conduct experiments to signify the consideration of query workload for index creation when search queries are executed on big data pool. In order to do so, we implement indexing on selective attributes for big data stored on distributed file system and queries having one of these attributes as selection predicate are applied.

3. Analysis Approach

In this section we present the analysis approach for non-clustered indexing implementation on big data. We elaborate the test bed as an experimental setup to perform analysis. Data sets which are used as input to execute the experiment are also described in this section. Furthermore, we present the mathematical model of analysis approach which is used to verify the results obtained from experiments.

We utilize an in memory attribute based non-clustered indexing to effectively analyze its impact on data retrieval performance from big data in comparison with those big data processing systems which do not provide indexing. To observe search performance for queries which miss index we used Hive warehouse over Hadoop. While we implement indexing that first creates indexes in memory for specified attributes on whole data set and then stores the index on file system for later use. For this purpose, Lucene library is utilized. The detailed experimental environment and our derivations are described as follows:

3.1. Experimental Setup

To evaluate the experiment results, we have established a setup with well-known Hadoop multi-node framework with four commodity servers. Hadoop Distributed File System (HDFS) is utilized for storage in our experiment where storage cluster can be built easily on local commodity hardware. Other renowned files systems such as Amazon S3 and WASB are also available for big data storage. However, both Amazon S3 [1] and WSB [3] are cloud based storage systems which offer paid storage on their web servers. In our setup, we have deployed a four-node cluster on physical machines consisting four slave nodes where one of them acts as both master and slave. Apache Hive is plugged in with Hadoop cluster so

that SQL-like queries can be performed on big data. In this way we will execute queries not having attributes as selection predicates which are used in indexing. Index Hit Ratio will be less in this scenario. Apache Lucene library is used to implement in memory indexing where we keep incrementing number of index attributes on each data set to see the performance and overhead caused by indexing. After each increment, we observe that Index Hit Ratio is increased whereas the indexing overhead is also increased. Fig 1 presents this experimental setup where each slave has TaskTracker and DataNode daemons respectively from MapReduce and HDFS components of Hadoop whereas master node has NameNode and JobTracker daemons.

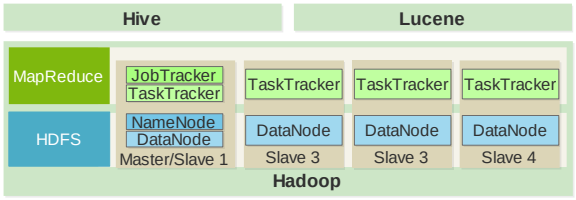

Fig 1 Experimental Setup.

We use Apache Lucene library [16], which is a Java built indexing library widely adopted for full-text search, to observe the impact of indexing on search operations. A java program is coded to create an index and to import Apache Lucene libraries. The code creates an attribute based index in HDFS memory for those data attributes which we specify to consider as index attributes. We consider varying number of attributes to create index to see the increasing overhead of size and increasing Index Hit Ratio with respect to increased number of index attributes. Once the index is created in memory, it is stored in HDFS so that query can utilize this index later for data retrieval requests. Fig 2 describes the work flow of our implementation. The steps involved in data processing are: 1) cleanse data 2) upload data into HDFS cluster, 3) create an index in HDFS memory on particular attributes and finally 4) apply search operation based upon search query initialized inside java code.

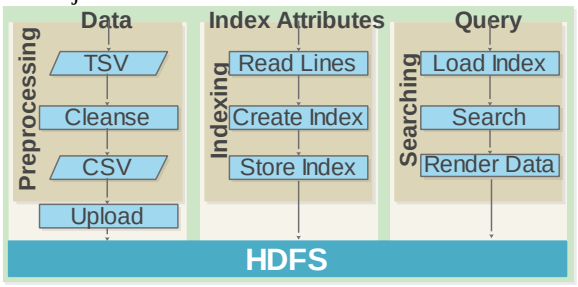

Fig 2 Experimental Workflow

Fig 2 illustrates the process flow of data search operation using indexing. Preprocessing of data depends upon the nature of data set on which we are going to apply search operations. We use MB as a unit of measure for size of data and later for index. To observe search performance when utilizing index in query execution, we apply queries with same attributes as selection predicates which were used in index creation. The sample of executed queries is presented as below:

$$SELECT\ attr1\ FROM\ Data \\ WHERE\ attr2 = data; \quad (3)$$

where $attr1, attr2 \in list\_of\_index\_attribues$

Queries which do not hit indexes are executed using Hive. Following is the definition of Index Hit Ratio with the condition that selection predicates of incoming queries are normally distributed:

$$Index\ Hit\ Ratio = \frac{|M|}{|N|} \quad (4)$$

where $M \subseteq N$,

$N = \{attr_d : attr_d\ is\ attribute\ name\ of\ Data\}$

$M = \{attr_i : attr_i \in N\ \&attr_i\ is\ used\ for\ indexing\}$

The data set we used in this experiment is in CSV (comma separated values) format. However, the indexing method supports TSV (Tab Separated Values). Therefore we first cleanse the data to replace comma with tabs before uploading data to HDFS. During upload process HDFS splits data set into fixed size blocks and locates each block to available DataNodes. Data uploading time on HDFS increases with the size of data. Table 2 presents the number of blocks for each data set and total data upload time taken to upload these blocks. Once data is uploaded, we specify the attribute(s) for indexing and start index creation for data set. In main memory indexing, data is loaded from HDFS storage to main memory and index is created in memory. The index is later stored into HDFS so that incoming queries can access this index for data search operation. When a query is submitted having same attributes as selection predicates the index is loaded in main memory and traversed in search phase according to query. Index returns the location of queried record and the records are loaded in memory.

3.2. Data Sets Used

We used data sets with varying size from web repositories to analyze search performance on different

size data. The data set comprises spatial information collected from US Census Bureau's TIGER database [7]. The database has features like roads, railroads, rivers and other legal and statistical geographic areas. We unzip the downloaded data and perform preprocessing as depicted in Fig 2 Experimental Workflow. We use Hadoop default configuration for block size (i.e. 64MB) and replication factor (i.e. 3) in our experiment. By following the experimental workflow, data is then uploaded over HDFS cluster. Table 2 provides a precise illustration of these data sets. Table 2 shows that the size of data sets is gradually increasing from 'Primary Roads' to 'Road Network. During data upload process, HDFS divides data into fixed size blocks. Table 2 illustrates number of blocks created from each data set and recorded data upload time taken by these data sets.

Table 2 Data sets

Data Sets	Data Size	No. of Blocks	Uploading Time (s)
Primary Roads	77.1	2	7
Area Landmark	406	7	127
Tabulation Area	1,600	25	227
Area Hydrography	6,460	104	814
Linear Hydrography	18,270	293	1984

3.3. Mathematical Model

We model the experimental setup using CPN Tools [10] which is broadly used to design and interpret Colored Petri nets. We implement all three steps i.e. preprocessing, indexing and searching as elaborated in Fig 2 using CPN Tools. Preprocessing performs 'cleanse' operation on data from 'CSV' place and 'upload data' operation on resulting data from 'TSV' place. Indexing executes 'create index' transition based on data from 'Idx Attr' and 'loaded data' places. Searching starts with 'load index' transition and performs 'search' and 'render data' operations to retrieve data. We collect data for index creation time and query execution time. The obtained data is used to verify experimental results. Fig 3 presents the model which comprises places, transitions, and input and output arcs. We implement timed transitions to calculate the effect of time on obtained results. All the places and transition of the model are explained in Table 3 and Table 4 respectively.

Table 3 presents the description of each Place involved in mathematical model. The table also elaborates the initial marking for these places. Table 4 defines the functions as transitions of the model. Table 4 further shows that all the transitions except "discard" are timed transitions and time is defined for each operation.

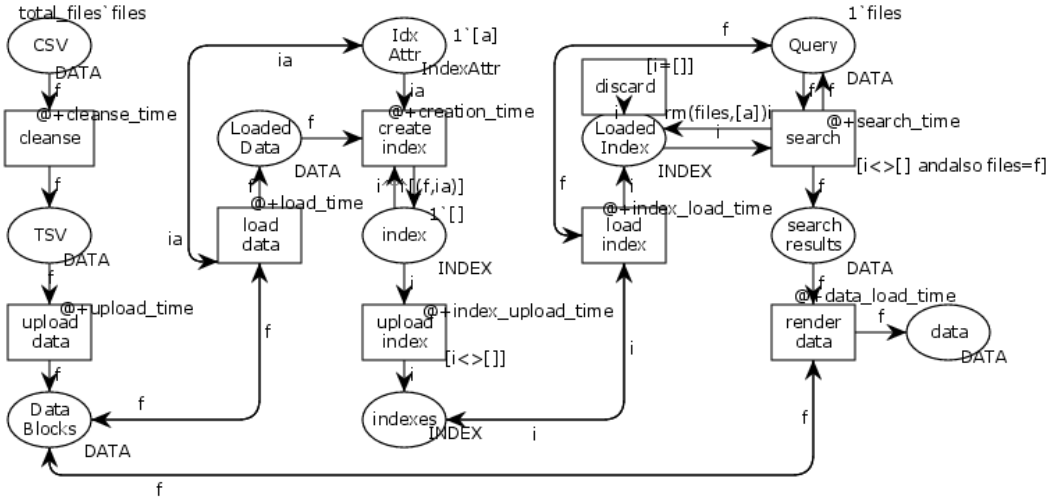

Fig 3 Mathematical Model for Experiment

Table 3 Description of Places

Places	Description	Initial Marking
CSV	Contains data set in CSV format	No. of files
TSV	Contains data set in TSV format	Empty
Data Blocks	Contains data set stored at HDFS in the form of blocks	Empty
Idx Attr	Contains list of data attributes provided by user to create indexes	One token
Loaded Data	Contains data set loaded from HDFS into memory	Empty
Index	Contains created index residing in memory	Empty

Indexes	Contains created index(es) stored at HDFS	Empty
Query	Contains query string to perform search operation	On token
Loaded Index	Contains index loaded from HDFS into memory	Empty
search results	Contains HDFS locations where the required data is residing	Empty
Data	Contains data returned for submitted query	Empty

Table 4 Description of Transitions

Transitions	Description	Timed/ Untimed
cleanse	Converts CSV file into TSV	Timed
upload data	Uploads data set into HDFS	Timed
load data	Loads data from HDFS to memory	Timed
create index	Creates index for data set	Timed
upload index	Uploads created index into HDFS	Timed
load index	Loads index from HDFS to memory	Timed
search	Searches the locations of required data in an index	Timed
discard	Discards empty index file	Untimed
render data	Retrieves required data from HDFS file location	Timed

4. Results and Discussion

In this section we present the results obtained from our experiments. We further verify the experimental results with mathematical modeling and provide a comparative discussion of clustered and non-clustered approaches of indexing for big data. In regard of inspecting search performance due to indexing, we have performed same search queries on both Apache Hive and indexing environment. As depicted in Fig 2, step by step activities such as create index, store index, load index, apply search and load data are performed to accomplish the indexing process. All these activities are also performed with mathematical model and almost similar results are obtained for each data set. The representation of results is two-fold: we first graphically present the overhead caused by these activities for increasing Data set Size whereas we consider maximum five data attributes from each data set as index attributes. Later we compare the experiment results with mathematical model results which strengthen our claim.

4.1. Experimental Results

In this section we present the results for Indexing Overhead, Index Size Overhead and Search Performance while executing experiment on Hadoop four-node cluster. We also present the results for Index Hit Ratio in this section which highlights the significance of considering more attributes in index creation.

- *Indexing Overhead*

Fig 4 shows Index Creation Time for each data set while considering up to five attributes for indexing. Index Creation Time is accumulative to Index Creation Time and Index Upload Time. When Data set Size is small the overhead to create index is also low. We also present indexing time comparison with data upload time in Fig 4 for up to five index attributes and show that index creation takes almost the same time as taken to upload a data set into HDFS.

We can also conclude from indexing time comparison presented in Fig 4 that with the increase of Data set Size, indexing becomes more time consuming. Furthermore, time overhead caused by index creation is very high. Referring to Fig 5, we can see that overall index creation overhead is 40 – 90 %. However, regardless of high One-Time Index Creation Time for larger size data sets index creation overhead is relatively low. This outcome leads in declaration to implement indexing for larger size data sets so that index creation overhead will not be very high.

- *Index Size Overhead:*

Index Size also increases when indexing is applied on larger size data sets. Like Index Creation Overhead, Index Size is also an overhead on data set size. However despite of increasing Index Size, overall Index Size overhead with respect to Data set Size is very low and decreasing gradually for larger size data sets which is a significant improvement. Fig 6 presents Index Size results for each data set and the comparison of index size with data set size. It is clear from Fig 6 that index size is very smaller than data set size even when up to five attributes are considered in indexing. Therefore, index size overhead is also very low (see Fig 7).

- *Index Hit Ratio*

Another useful parameter in our experiment is Index Hit Ratio. For an efficient attribute-based indexing mechanism, this ratio is supposed to be very high so that most of the incoming queries will be served using index. We have explained Index Hit Ratio in Section 3.1 as Definition 1. Fig 8 shows that Index Hit Ratio increases with number of index attributes. However, Index Size Overhead and Index Creation Overhead are also increased with number of index attributes. Therefore, we recommend to propose an indexing mechanism with which adding more index attributes to obtain increased Index Hit Ratio results in minimum increase in Index Size Overhead and Index Creation Overhead.

- *Search Performance*

Above observations highlight the importance of indexing for big data with growing volume. Although index creation cost becomes high for large size data sets yet indexing overhead with increased number of index attributes relative to data set properties i.e. Data set Size and Data Upload Time, is almost unchanged. Thus increased Index Hi Ratio can be achieved. One-time index creation means that, indexing overhead has to be tolerated only once before starting query execution. Once the index is created and indexing overhead is withstood, the improvement in search performance for indexes search queries will be observed. Fig 9 presents the improved search time results of indexed search queries over full scan query execution. Search Time regardless of Data set Size is decreased when indexing is applied. Therefore, search performance is increased up to 98% when an indexes are available (see Fig 10).

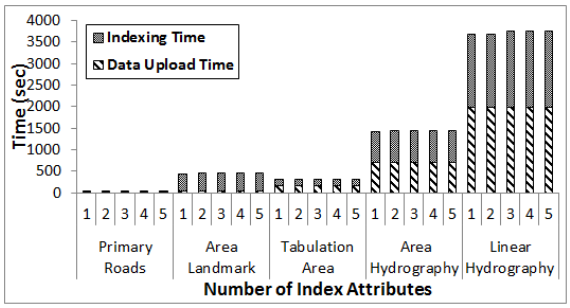

Fig 4 Indexing Time comparison with Data Upload Time and Number of Index Attributes

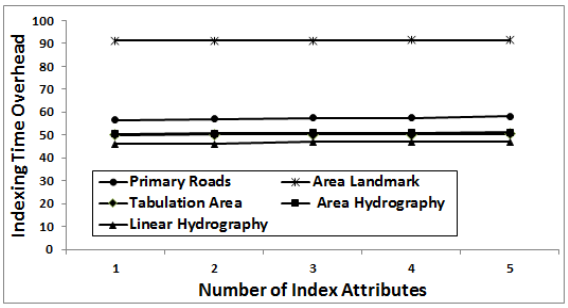

Fig 5 Index Creation Overhead for varying number of Index Attributes

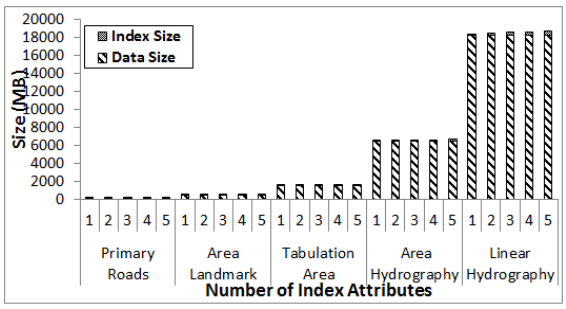

Fig 6 Index Creation Overhead for varying number of Index Attributes

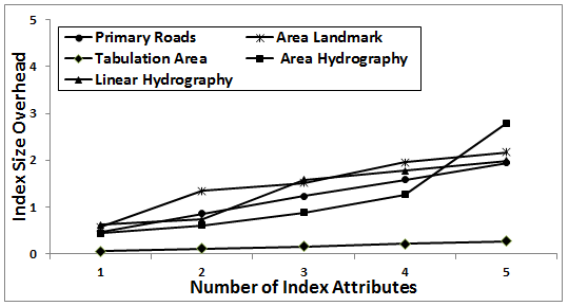

Fig 7 Index Size Overhead for varying number of Index Attributes

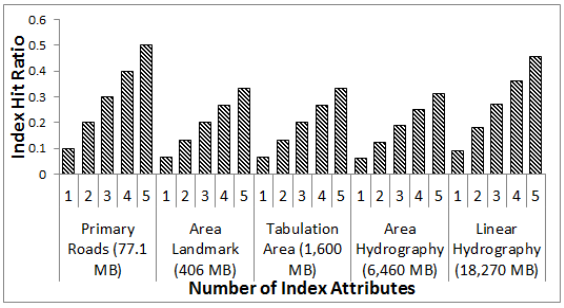

Fig 8 Index Hit Ratio w.r.t. Number of Index Attributes

We provide a discussion of results while implementing indexing on varying number of attributes. We discuss the overhead in terms of time and space taken by performing indexing on data. Furthermore, we present the effect of indexing on search performance as the ultimate gain expected from an indexing mechanism. We illustrate overhead resulted by indexing in terms of index creation time, Index Size and Index Upload Overhead which did not exist in system prior to indexing. As far as selection of index attributes is concerned, we show that there exists tradeoff between Index Size and Index Hit Ratio. We conclude that, there will be an apparent impact on Index Size while considering large number of indexing attributes. Ultimately, more number of index attributes increases Index Hit Ratio which will result in effective index utilization by incoming queries. Based upon these results, we have become able to claim that:

– Indexing is a significant process to improve search performance for relatively large and growing data sets.

– Overhead resulted by indexing process is one time and becomes negligible when clear improvement in search performance is obtained.
– Overall indexing overhead is somehow inversely proportional to size of data set but directly proportional to number of index attributes.
– The more the number of attributes considered in indexing, the more the overhead is faced, though Index Hit Ratio becomes high.
– A wise selection of attributes for indexing gives a better tradeoff between Indexing Overhead and Hit Ratio
– Adaptive index updating also supports prior claim.

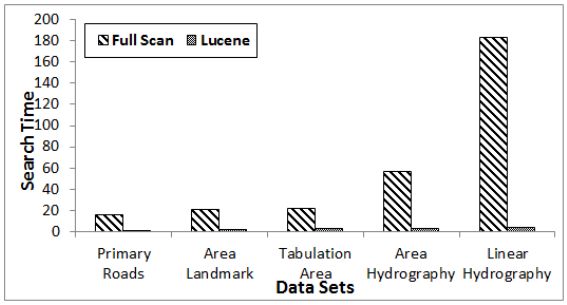

Fig 9 Indexed Search Query Execution Time Comparison with Full Scan

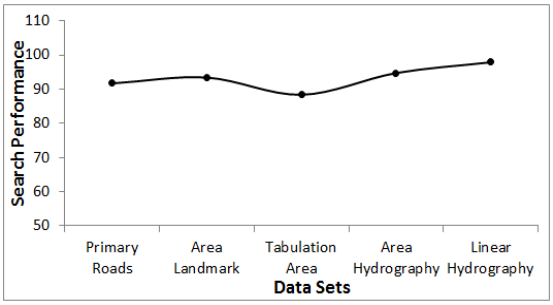

Fig 10 Impact of Indexing on Search Performance

4.2. Validation

We use mathematical modeling results obtained from CPN model to verify experimental results for Index Creation Time and Query Execution Time. For CPN model, we use μ to define execution time of a transition. For instance, $\mu_{indexing}$ denotes time taken in index creation for a data set which is defined as follows:

$$\mu_{indexing} = (\mu_{index\ creation} \times ia \times Loaded\ Data^{\#}) + (\mu_{index\ upload} \times index^{\#}) \quad (5)$$

Indexing Time $\mu_{indexing}$ is accumulative time taken in index creation $\mu_{index\ creation}$ and index uploading $\mu_{index\ upload}$ to HDFS. Index Creation Time depends upon number of index attributes ia and Data Set Size $Loaded\ Data^{\#}$ whereas Index Upload Time $\mu_{index\ upload}$ depends upon Index Size $index^{\#}$. For CPN execution we set $\mu_{index\ creation}$ as 0.12 and $\mu_{index\ upload}$ as 2.

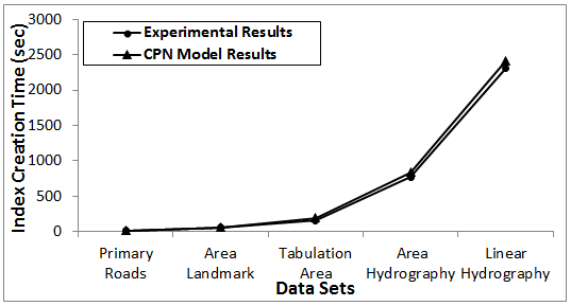

Fig 11 Index Creation Time Validation

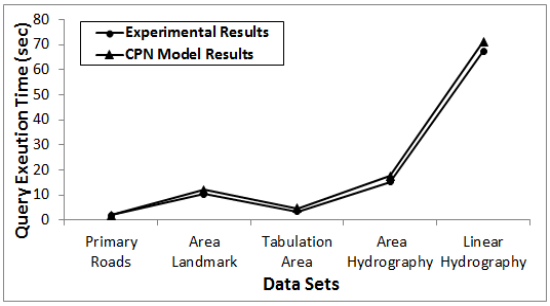

Fig 12 Query Execution Time Validation

The results are almost similar to index creation time results obtained from experiment (see Fig 11).

Furthermore, we denote Query Execution Time as $\mu_{indexed\ search}$ and define it as follows:

$$\mu_{indexed\ search} = \left((\mu_{index\ loading} + \mu_{search}) \times index^{\#}\right) + \mu_{data\ loading} \quad (6)$$

Query Execution Time for indexed search $\mu_{indexed\ search}$ is accumulative time taken in loading index $\mu_{index\ loading}$, searching required data μ_{search} and loading data $\mu_{data\ loading}$ into HDFS. We set values for $\mu_{index\ loading}$, μ_{search} and $\mu_{data\ loading}$ to 3.8, 1.2 and 0.009. The results in Fig 12 verify Query Execution Time for indexed search.

4.3. Discussion

In section 2 we reviewed state-of-the-art clustered indexing techniques for big data. We categorized clustered indexing techniques based on their index creation time as static indexing and adaptive indexing techniques. Static indexes are created at the time of data upload. Once the index attributes are defined at data upload time and indexes are created based on this set of index attributes, this set will never be changed during life time of data residing on file system regardless of considering query workload that whether the queries are utilizing those indexes or not. The only condition to update index is when data is updated. On the other hand, adaptive indexes are the side effect of query execution. Adaptive indexes are created and updated after each new query execution. The main disadvantage of adaptive indexing is that each new query cannot be leveraged with indexes and the index will be created after query execution. This index is only useful when the same query is executed.

Clustered approach except pros and cons of static and adaptive categories has its own limitations due to which clustered approach is not preferable to obtain increased Index Hit Ratio. However, referring to section 3, implementation of in memory indexing which is a non-clustered indexing approach implies that selection of more index attributes to maximize Index Hit Ratio is just a matter of Index Size Overhead. Otherwise, non-clustered approach for indexing is favorable to clustered approach. Therefore, we suggest implementation of non-clustered approaches for indexing big data so that maximum incoming queries will be served using indexing for better search performance. We present our findings of section 2 and section 3 and provide a comparative analysis of clustered and non-clustered indexing approaches in Table 5 to strengthen our suggestion. This analysis serves as a basis to applaud non-clustered indexing proposal. Later we will assert for hybrid approach to exploit both static and adaptive mechanisms so that flexible to query workload indexes are created.

Table 5 Comparison of Clustered and Non-clustered Indexing Approaches

Process	Clustered Approach	Non-clustered Approach	Recommended
Index Creation	Physically reorders data rows using Quick sort [2] and stores sorted rows as a block [6]. Complexity of Quick Sort: $O(n \log n)$	Separate index structure containing key-value pairs [11]. Key refers to index attribute and the value is pointer to the row. No physical reordering of data records is required. Complexity of B-Tree: $O(\log n)$	Non-clustered
No. of Indexes	One replica can have only one index. One copy of data block cannot have more than one sort orders [20]. $\lvert Index\ Attributes \rvert = \lvert Replicas \rvert$	Single replica can have more than one index. As index is a separate structure [11], for one copy of data block as many indexes can be created as storage space allows. $\lvert Index\ Attributes \rvert \geq \lvert Replicas \rvert$	Non-clustered
Index Size (single index)	Less size than non-clustered [6]. However, for multiple indexes the size reaches storage capacity [20].	Separate structure [13] needs more space. However, creating index is less space consuming than creating separate replica.	Clustered
Index Maintenance (query log)	Index rebuilding needs to drop and create new data blocks [20].	Index rebuilding is easy [4]. It requires to perform delete operation on previous index and iterate create index operation.	Non-clustered
New Index	Whole data set or specific block(s) should be replicated to create new index [20]. $Index\ Size(attr_i) = Block\ Size$	New index can be created on any of existing replica. There is no need to replicate data $Index\ Size(attr_i) \sim Index\ Size(attr_{i-1})$	Non-clustered
Data Update (insertion)	All copies of last block are updated and record is inserted on its exact location [6]. $O(n \log n)$ [4]	Record is appended on all copies of last block. Each index is updated [13]. $O(\log n)$	Non-clustered
Data read	Apply search algorithm in sorted list $O(\log n)$	First traverse index then jump to record $O(\log n)$	Same complexity
Index Hit Ratio	Depends upon number of replicas	Depends upon number of indexes	Non-clustered
Memory Requirement	Depends upon block size	Depends upon size of index	Clustered

Table 5 presents the comparison between clustered and non-clustered approaches and due to complexity measures resulted by each operation we recommend non-clustered approach to be utilized for designing indexing framework for distributed replicated blocks of big data. Table 5 shows that all operations except single index size are more efficient than clustered approach. Although index size for single index of clustered approach is less than non-clustered approach, yet non-clustered approach is still better. The reason behind is clustered approach requires new copy of whole data or data block for each new index whereas in case of non-clustered indexing approach new index can be created with existing replication factor.

Experiment on non-clustered indexing also shows that, projection of non-indexed attribute using queries is not possible as only index keys are stored. Therefore, when only selective attributes are considered in index creation, queries cannot retrieve data for attributes other (non-indexed) attributes. Similarly, retrieval of whole data row is not possible in this case. Furthermore, loading indexes into memory to search queried data incurs noticeable I/O cost which is only related to non-clustered indexing approaches. Therefore, we suggest in proposing an indexing mechanism for big data where indexing on a limited number of index attributes deals with the problem of accessing non-indexed attributes as well. I/O cost should also be negligible which deteriorates the performance of a non-clustered indexing approach.

5. Our Recommendations

Based upon above discussion we suggest implementation of non-clustered approach for indexing big data so that on existing number of replica improved Index Hit Ratio leading to improved search performance can be achieved. Although replication of data is significant to increase fault-tolerance and availability of data yet each new copy of data increases space requirements and consumes more storage resources. For continuously growing voluminous big data, increasing the replication factor of data storage is not a feasible solution to consider more attributes in indexing. Therefore, improvement in Index Hit Ratio should not be subject to higher value of replication factor. Instead, replication should be utilized to balance the load of indexing via parallel index creation is performed on each replica. In addition, number of indexes can be divided among replicas using either equality-based or efficiency-based strategy. Replication can be also be utilized to increase fault-tolerance if index attributes are replicated.

As far as selection of static or dynamic indexing is concerned, we suggest applying hybrid approach. According to Table 1 we can say that though static indexing is beneficial for each query having attributes as selection predicates similar to index attributes and index growth does not depend on new queries yet one time index creation may not predict future workload of queries. Index may need to be updated according to changing query workload. Therefore, relying solely on static approach is not advantageous. In the meantime, adaptive indexes which grow as side effect of query execution may result in many unused indexes. Any incoming query does not state that whether the same query will be submitted again or not. Therefore, creating indexes blindly for each query with new attribute as selection predicate may result in a large number of replicas. Consequently, indexing will become an overburdening rather than search facilitating activity. Thus, keeping in mind all these factors, we recommend an optimized indexing framework for big data must possess following features:

- Wise selection of index attributes at static indexing stage so that maximum queries will be served by these indexes. User-defined list of index attributes may achieve maximum Hit Ratio for specific period of time.
- Heuristic decision to update list of index attributes so that recent query trends are considered and indexes will not be obsolete. User may not be fully aware of query plan and data search preferences may change with the passage of time. Therefore, adaptive to changing query workload indexes are more efficient.
- Number of indexes should be independent of replication factor. This is only possible if we use non-clustered approach for indexing.
- Faster index rebuilding so that index update cost is not very high. Adaptive stage in indexing may add new indexes and delete unused or rarely used indexes according to changing query workload. Non-clustered approach does not change physical data storage and indexes rebuilding is easy.
- Efficient index update as an effect of data update so that challenge of growing data is accepted. One such mechanism for non-clustered indexing which shows better insertion performance than quicksort mechanism of clustered approach is preferable.

6. Conclusion

Faster data retrieval from big data is the main concern of data analysts and users. This motivation has led the research and development industry towards exploration of efficient data processing mechanisms. We present an experimental evaluation of non-clustered indexing on varying size data sets with varying Index Hit Ratio in this paper. We further validate evaluation results using CPN mathematical modeling. The results suggest that existence of index considerably improves the search performance for particularly large data sets. Although indexing process introduces some overhead, yet somehow this overhead is decreased for larger size data sets. The comparative discussion on clustered and non-clustered approaches leads towards a clear recommendation of implementing a hybrid approach of both static and adaptive non-clustered indexing. We further suggest to wisely selecting index attributes for better tradeoff between Indexing Overhead and Index Hit Ratio. Moreover, adaptive indexing where index is updated according to changing query workload also improves this tradeoff. Based on the recommendations provided in this paper, we are moving towards implementation of non-clustered multi-attribute static indexing on user suggested list of index attributes and heuristic analysis of query workload to adaptively improve index hit ratio by creating/deleting indexes on frequently used/unused attributes a as a subsequent future work.

References

[1] 'Amazon'. *Amazon S3*. 2016 14-8-2016]; Available from: http:/s3.amazonaws.com.

[2] Alvarez, V., F.M. Schuhknecht, J. Dittrich, et al., *Main memory adaptive indexing for multi-core systems*, in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*2014, ACM: Snowbird, Utah. p. 1-10.

[3] Calder, B., J. Wang, A. Ogus, et al. *Windows Azure Storage: a highly available cloud storage service with strong consistency*. in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011. ACM.

[4] Chi, P., W.-C. Lee, and Y. Xie, *Making B⁺-tree efficient in PCM-based main memory*, in *Proceedings of the 2014 international symposium on Low power electronics and design*2014, ACM: La Jolla, California, USA. p. 69-74.

[5] Diallo, O., J.J.P.C. Rodrigues, M. Sene, et al., *Real-time query processing optimization for cloud-based wireless body area networks.* Information Sciences, 2014(0).

[6] Dittrich, J., J.-A. Quian, Quiané-Ruiz, et al., *Only aggressive elephants are fast elephants.* Proc. VLDB Endow., 2012. **5**(11): p. 1591-1602.

[7] Eldawy, A. and M.F. Mokbel. *Spatial Hadoop: A MapReduce Framework for Spatial Data*. in *2015 IEEE 31st International Conference on Data Engineering*. 2015. IEEE:1352-1363.

[8] Gani, A., A. Siddiqa, S. Shamshirband, et al., *A survey on indexing techniques for big data: taxonomy and performance evaluation.* Knowledge and Information Systems, 2015. **46**(2): p. 1-44.

[9] Jens, D., Q.-r. Jorge-Arnulfo, and J. Alekh. *Hadoop++: Making a Yellow Elephant Run Like a Cheetah*. 2010. VLDB.

[10] Jensen, K., L.M. Kristensen, and L. Wells, *Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems.* Int. J. Softw. Tools Technol. Transf., 2007. **9**(3): p. 213-254.

[11] Jin, R., H.-J. Cho, and T.-S. Chung, *A group round robin based b-tree index storage scheme for flash memory devices*, in *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*2014, ACM: Siem Reap, Cambodia. p. 1-6.

[12] Jindal, A., J.-A. Quiané-Ruiz, and J. Dittrich. *Trojan data layouts: right shoes for a running elephant*. in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 2011. ACM:1-14.

[13] Kaplanis, A., M. Kendea, S. Sioutas, et al., *HB+tree: use hadoop and HBase even your data isn't that big*, in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*2015, ACM: Salamanca, Spain. p. 973-980.

[14] Karim, A., R. Salleh, and M.K. Khan, *SMARTbot: A Behavioral Analysis Framework Augmented with Machine Learning to Identify Mobile Botnet Applications.* PloS one, 2016. **11**(3): p. e0150077.

[15] Karim, A., R. Salleh, M.K. Khan, et al., *On the Analysis and Detection of Mobile Botnet Applications.* Journal of Universal Computer Science, 2016. **22**(4): p. 567-588.

[16] McCandless, M., E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*. 2010: Manning Publications Co.

[17] Qin, Y., Q.Z. Sheng, N.J. Falkner, et al., *When things matter: A survey on data-centric internet of things.* Journal of Network and Computer Applications, 2016. **64**: p. 137-153.

[18] Richter, S., J.-A. Quiané-Ruiz, S. Schuh, et al., *Towards zero-overhead adaptive indexing in Hadoop.* arXiv preprint arXiv:1212.3480, 2012.

[19] Richter, S., J.-A. Quiané-Ruiz, S. Schuh, et al., *Towards zero-overhead static and adaptive indexing in Hadoop.* The VLDB Journal, 2014. **23**(3): p. 469-494.

[20] Schuh, S. and J. Dittrich. *AIR: Adaptive Index Replacement in Hadoop*. in *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on: 22-29*. 2015.

[21] SIDDIQA, A., A. KARIM, and G. Abdullah, *Big data storage technologies: a survey.* Frontiers, 2016. **1**.

[22] Siddiqa, A., I.A. TargioHashem, I. Yaqoob, et al., *A Survey of Big Data Management: Taxonomy and State-of-the-Art.* Journal of Network and Computer Applications, 2016.

[23] Sookhak, M., A. Gani, M.K. Khan, et al., *Dynamic remote data auditing for securing big data storage in cloud computing.* Information Sciences.

[24] Strohbach, M., H. Ziekow, V. Gazis, et al., *Towards a big data analytics framework for IoT and smart city applications*, in *Modeling and Processing for Next-Generation Big-Data Technologies*. 2015, Springer. p. 257-282.

[25] Zhuang, Y., N. Jiang, Q. Li, et al., *Progressive Batch Medical Image Retrieval Processing in Mobile Wireless Networks.* ACM Trans. Internet Technol., 2015. **15**(3): p. 1-27.