# Big Data Analytics: Partitioned B+-Tree-Based Indexing in MapReduce

**Ali Usman Abdullahi, Rohiza Ahmad, and Nordin M. Zakaria**

**Abstract** Big data analytics platforms are designed to improve performance by avoiding the extract transfer load approach. Also, there are techniques which have worked very well in performance optimization for relational databases. Yet these techniques are in the process of integration into big data analytics. Indexing and its data structure is an example of such techniques. Despite its popularity in query optimization for efficient data mining, the indexing was not integrated into the MapReduce platform. By design the MapReduce was made to perform a full scan of the input data. However, there were attempts made to incorporate the indexing for performance improvement in MapReduce in recent years. However, these attempts have not exhausted the potentials of indexing in the MapReduce query processing. Consequently, this chapter presents an indexing approach that uses the partitioned B+-Tree as its data structure to index the InputSplit component of the Hadoop distributed file system. This was done to achieve efficient data mining query processing when used with the Hadoop MapReduce. The results of this study showed that the proposed index method has significantly reduced the index size as well as the execution runtime of all search queries by at least 50% for all the used data sizes when compared with the Normal MapReduce processing and another clustered index approach. Thus, the use of the proposed index approach has the potential to significantly reduce the time taken in mining data within a dataset by half.

A. U. Abdullahi (✉)
Computer Science Education Department, Federal College of Education (Tech), Gombe, Nigeria
e-mail: usmanali@fcetgombe.edu.ng
URL: http://www.fcetgombe.edu.ng

R. Ahmad · N. M. Zakaria
Computer and Information Sciences Department, Universiti Teknologi PETRONAS, Seri Iskandar, Perak, Malaysia
e-mail: rohiza_ahmad@utp.edu.my

N. M. Zakaria
e-mail: nordinzakaria@utp.edu.my

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

H. Chiroma et al. (eds.), *Machine Learning and Data Mining for Emerging Trend in Cyber Dynamics*, https://doi.org/10.1007/978-3-030-66288-2_9

# 1 Introduction

Indexing, as an information retrieval technique, is the process of generating all the suitable data structures that allow for efficient retrieval of stored information [9, 21]. The term index referred to the suitable data structure needed to allow for the efficient information retrieval [21]. The data structures used in index in most cases do not store the information itself rather it uses other data structures and pointers to locate where the data has been stored. Also, the indexing have played a very important role in performance improvement of relational databases. The indexing techniques in relational database management systems (RDBMS) are well developed and matured [9, 18].

However, in big data analytics the indexing concept is still at the developmental stage and needs more attention from researchers as pointed out by Yang and Parker [34], An [2], Richter et al. [25] and Sevugan et al. [28]. There are two indexing concepts when it comes to indexing in big data analytics. The first one is the document indexing, which is the use of the MapReduce process to index the contents of any stored document for easy retrieval. Usually, the most used indexing technique for this task is the inverted index [21]. In this concept the main goal of the MapReduce job is the indexing of content of the document itself [20]. Early works of indexing in MapReduce were concentrated on this aspect. This was supported by the fact that the developers of MapReduce (Google) dealt mainly with searching through large document datasets [6, 7, 31]. Basically the search engine companies are interested in fast retrieval of their queried stored data [15, 18, 19, 30, 35].

The second concept is indexing for speeding up the process of the MapReduce's job execution. The indexing here performs the same function it does with the traditional databases of restricting the query processing to only the input data to be affected by any given query [2, 5, 10, 17, 34]. This indexing is similar to the one found in most structured query language (SQL)-on-Hadoop, and even in other variants of distributed files systems (DFS), like BigTable. However, the way and manner in which the index approach works in the traditional database, SQL-on-Hadoop and even on table-based DFS, is that it is usually based on the storage structure, i.e., table for those platforms that use tables as storage structure [14, 16, 19]. This fact means that indexes are usually provided for each table making the amount of data to be scanned before query processing to increase. On the contrary, the user-defined indexing approach is quite different when it is used directly on Hadoop, where Hadoop distributed files systems (HDFS) components that handle the stored data are indexed [22, 33].

Particularly in the case of MapReduce, this indexing concept ensures the scanning of only those blocks/chunks/splits that are targeted by the query to be processed. The study presented in this chapter is focused on this second indexing concept as a way of improving big data analytics using the MapReduce programming paradigm for data mining [14].

This concept of indexing, if integrated into the MapReduce execution process can help solve the following problems: (1) The scanning of the whole input data, (2)

The same ways of handling both low and high selectivity tasks and (3) The lack of robust access method for information retrieval in online access processing (OLAP) situations.

In an effort to overcome these problems some researchers have attempted to incorporate the indexing mechanism into the MapReduce framework. For instance, Yang and Parker [34] have employed HDFS files as B-Tree nodes to achieve indexing. Also, An [2] used blockIds from the HDFS as B+Tree search keys for determining the start and the end of the contiguous blocks to scan, which is used as index. In addition, Richter et al. [25] used the copies of the replica stored by HDFS, to form a clustered index of different data attributes that may likely be used as incoming query predicates. Clustered indexes are built on those fields, which is subsequently used for query processing.

In all of these studies, the authors used an indexing data structure that scale logarithmically, thereby preventing the MapReduce from performing a full scan of the input data. Thus, guiding the process is to just scan and process the data that corresponds to the output of the indexes. Notwithstanding the prevention of full scanning of the input data by these approaches, their indexes are built using all of the keys from the stored data. So, the full scanning of the indexing structure itself is done due to the nature of the components of the HDFS used. Furthermore, when the B-Tree and B+-Tree as indexing data structures are used in MapReduce it can facilitate the scanning of relevant portions of the stored data to answer range queries covering small sections (low selectivity) or wider sections of the data (high selectivity).

However, the search algorithm in the B-Tree takes a non-uniform time for different types of queries, due to the storing of the search key's corresponding values/records at all levels of the tree [29], while the search algorithm in the B+-Tree involves searching the whole of the tree all the time [3]. This is relevant because the search in the tree structures is the basic process that all other processes are based on. The search algorithm of B-Tree has been improved in the B+-Tree variants and those variants can be used to further improve the performance in the MapReduce query processing.

Lastly, any indexing approach to be used in MapReduce should improve the two problems mentioned for the effectiveness of MapReduce in mining data stored in the warehouse for OLAP. Thus, the indexing approach proposed in the chapter chooses an improved variant of the B+-Tree called partitioned B+Tree introduced by Abdullahi et al. [1]. The partitioned B+-Tree provides an improved search algorithm, which ensures reduced execution time for query retrieval. This fact is what influences a quick record retrieval during mining of big data using MapReduce platforms [16, 26].

## *1.1 Objective of the Chapter*

The main objective of this chapter was to present an indexing approach that puts together partitioned B+-Tree as data structure and its algorithms for inserting and

searching of data. The strategy of the proposed index approach is to restrict the amount of the input data to be processed by the MapReduce platform to only the part of the input data that is required for the query processing.

The remaining sections of the chapter are as follows: Sect. 2 presents the review of the literatures, Sect. 3 discusses the methodology used in the study and Sect. 4 presents the experimental setup and the results of the experiments conducted. The conclusion and future work are discussed in Sect. 6.

## 2 Literature Review

In this section the review of the two concepts of indexing that were used in MapReduce is presented.

### *2.1 Inverted Index in MapReduce*

The first concept of the indexing involves a simple inverted index which was said to be implemented trivially in MapReduce as one of the effective tasks for textual data retrieval. This was highlighted in the original paper on MapReduce [8] cited by Graefe and Kuno [12] and discussed in Cao et al. [4], Stewart et al. [32] and Mofidpoor et al. [22]. The inverted index works by parsing the split of the input data to the map function, which performs the filtering of the data as defined in it and emits a sequence of <*key,value*> pairs. Then, the reduce function accepts all of the pairs of the same key, sorts the corresponding values and emits <*key,list*(*value*)> pair. The set of all the output pairs form a simple inverted index. McCreadie, Macdonald and Ounis [20] concluded that two interpretations of the above scenario can be a per-token indexing or a per-term indexing in relation to the indexing of corpus datasets: The per-token indexing strategy involves emitting <*term,doc−ID*> pairs for each token in a document by the map function, while the reduce function does the aggregation of each unique term with its corresponding doc-ID to obtain the term frequencies (tf), after which the completed posting list for that term is written to a disk.

On the other hand, the per-term indexing is the one in which the map function emits tuples in the form <*term,*(*doc−ID,tf*)> and this reduces the number of emitted operations as only unique terms per document are emitted. The reduce function only sorts the instances by document to obtain the final posting list sorted by ascending doc-ID.

Apart from these two methods the inverted index is also used by MapReduce to index a whole document in what is called per-document indexing. In the per-document indexing, the map function emits tuples in the form of <*document,doc−ID*>, while the reduce phase writes all the index structures. Though this strategy emits fewer key/value pairs, the value of each pair emitted used to have

more data and has a reduced intermediate result. Thus, this approach achieves higher levels of compression than the single-term approaches. Also, the documents are easily indexed on the same reduce task due to the sorting of the document names [20].

In addition, there is the per-posting list indexing. This approach is based on a single-pass indexing, which splits data onto multiple map tasks with each operating on its own data subset. The map task serves as the scanning phase of the single-pass indexing. As this process runs on the document, a compressed posting list is built in the memory for each term. This partial index is flushed by the map task when the memory runs low or when all the documents are processed. The flushing is done by emitting a set in the form of <*term,postinglist*> pairs for all the term present in the memory. In all of these four strategies of document indexing in MapReduce the main task focus is on and carried out by the MapReduce job, which is the indexing of the document's contents [20].

### 2.2 User-Defined Indexing in MapReduce

The indexing here focuses on restricting the query processing to include only those parts of the input data to be effected by the query [2, 34]. For the MapReduce framework, this concept achieves its purpose by only scanning of the blocks or splits or files that contain the input data. This study has focused on the second indexing concept as a way of improving big data analytics using the MapReduce programming paradigm.

In big data analytics, the second indexing concept has been initially seen as being less important or a costly one due to the fact that infrequent updating is not a common characteristic of big data. However, later works in the area indicated that indexing can play a major role in the information retrieval aspect of the big data analytics. This makes its benefit to surpass its cost. Yang and Parker [34], for instance, implemented indexing by employing a HDFS file as B-Tree node. The index used by the authors is considered non-clustered index [27]. This in turn was due to the fact that the reading of the index is not done sequentially. Their approach adds a primitive function to traverse the tree in order to locate the required segment of the data to be processed by their improved Map-reduce-merge-traverse version of MapReduce. This introduction of the B-Tree index has improved the performance of the MapReduce (changed to Map-reduce-merge-traverse). The improvement is up to 50% and above for a three-node cluster when compared to any number of nodes less than 10 with the normal MapReduce.

On the other hand, An [2] used blockId as the B+-Tree's search keys and the offset of the keys as a value to build an index. The built index is stored in a HDFS file as contiguous blocks rising from the node at the bottom to the root at the top. The index-based execution of the MapReduce is done by first searching for the keys that are contained in the queries' predicate from the already built index. This is done to determine the starting and the end of the contiguous blocks that represent the search keys in the index. The blockIds that fall between the two points are those

that are scanned for query processing by the MapReduce. With these two mentioned techniques, the amount of data to be scanned by the MapReduce is restricted. The B+-Tree with blockIds as the keys index scan for MapReduce, outperformed the normal MapReduce by at most 50%, and below for varying percentage of I/O volume used by in the study.

Furthermore, Richter et al. [25] used the copy of the replicas stored by the HDFS to index different data attributes that may likely be used as predicates of incoming queries. Their approach used a clustered index [24] based on each of these replicas. The clustered index is first scanned to get the blockId of the data to be processed from the index. Then, it uses the returned blockIds to locate the input data to be scanned and used by the MapReduce for processing the query in the question. Then, it used its customized input-split policy that helped their approach to determine the part of the input data that could take part in the query processing. After the index result is returned, the results of using their indexing with other Hadoop settings indicated a 20% reduction of the execution runtime.

## 3 Methodology

This section of the chapter discusses the methodology used in the study. It gives the details of how various component of the study come together to achieve its object. Figure 1 shows the overall picture of the proposed indexing scheme.

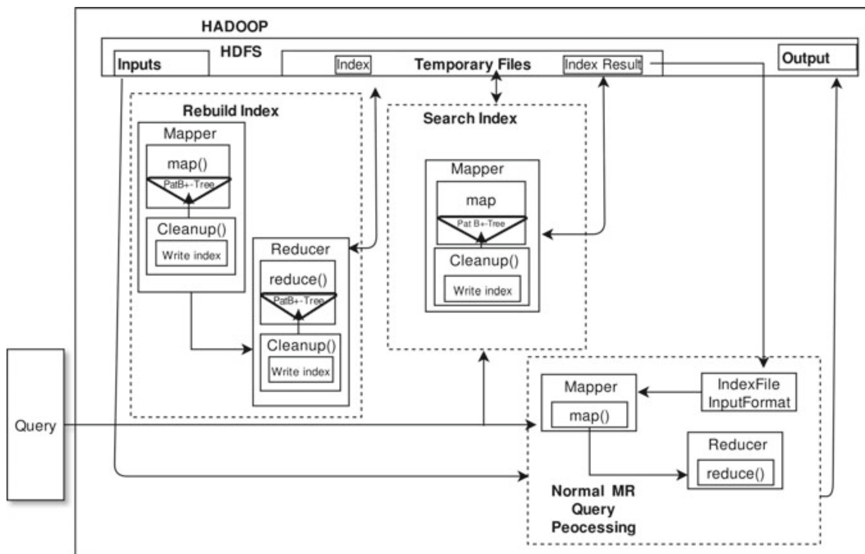


**Fig. 1** The block diagram of the scheme indexing for the MapReduce query processing

The proposed approach basically consists of several components. They include the query, which is the statement that specifies how the process should be done; the rebuild index component, which is used for building and subsequent re-building of the index when needed. Then, there is the search index component, which is used for searching the index using the values provided in the query's predicate. This component searches and returns the location of the data required by the query. Lastly, the scheme also has another component which is the query processing component. This component does the actual processing of the query been issued.

The approach works first by building an index using partitioned B+-Tree (the data structure) with search keys from the input data and their corresponding values (input-splits) using the rebuild index component. The built index is stored in a temporary file on the HDFS. When a query for certain processing is to be executed, its predicates will first be used to search the index to get the input-splits using the search index component. The search index component also uses the Pat B+-Tree to re-construct the index and use it to search and return the targeted results.

By doing that, only the input-splits that cover the part of the input data that is required by the query are returned and stored in yet another temporary file on the HDFS. The returned input-splits are then read and used by the query processing component to process the given query. The framework view of the approach's working flow, as mentioned, is captured in Fig. 1.

### *3.1 Partitioned B+-Tree*

The partitioned B+-Tree mentioned is an adaptation of the partitioned B-Tree [11], in which the author proposed the addition of an artificial lead key to the BTree. This addition resulted in a logical partitioning of the B-Tree based on run generations that feed the B-Tree during the initial index creation. Graefe [11] implemented his tree based on the traditional B-Tree. However, in this study the same idea was implemented with some improvements on B+Tree. The choice of partitioned B+-Tree for this study was due to its performance in a preliminary experiment in comparison with other variants of B+-Tree as reported in [1].

### *3.2 InputSplit as Component of Choice in the HDFS*

The corresponding values for the search keys used by the proposed indexing approach are the input-split components of the Hadoop distributed file system (HDFS). The input-split is the HDFS's feature that uniquely identifies the chunks of data that the Hadoop made use of for the proper managing of the blocks of the stored data on its HDFS [13, 23, 36].

The reason for choosing the input-split is because it provides a better balance between the index size and searching cost on one side and the size of the input data

to be scanned on the other. So, even if the blockId is to be used for indexing, the input-splits that cover those blocks have to be generated first, before extracting the blockIds from them.

**Index Rebuild in MapReduce** First, the index needs to be built over the complete input data before it is used as mentioned earlier. A complete MapReduce program is used to extract the HDFS components alongside the search key values from the input data in order to build the index.

Figure 2 has shown the expanded version of the rebuild index block of Fig. 1; here, the internal workings of the component are depicted. The rebuild index reads the inputs from the stored data on the HDFS using its Mapper class called the RebuildIndexMapper. This Mapper class has three methods, namely, setup(), map() and the cleanup(). The setup() method takes a context variable of the context class type as the argument. The context class is an optimized record reader/writer handler used by the Hadoop. The setup() method used the variable to extract the Hadoop internal component's settings and properties. One such component is the input-split, which is the targeted component for the proposed index approach as mentioned earlier.

Also, the context variable in the setup() method is used alongside the extracted input-split to further get the properties of other components, such as the filename of the input files. In some cases, the extracted filenames are concatenated with the extracted values from the records in order to form unique search key values. The use of the filename as part of the unique search key values is to avoid the treating of the keys from files with the same values as one and the same entry.

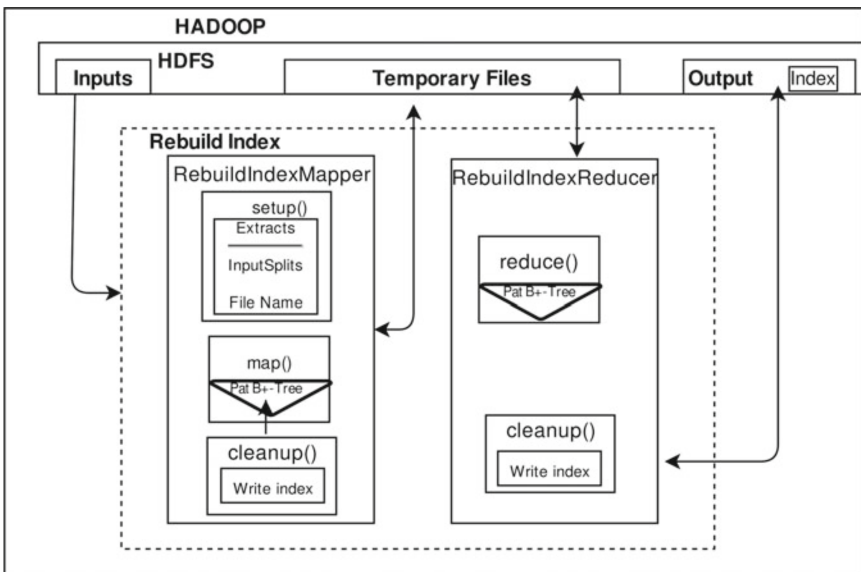


**Fig. 2** Block diagram for index building process

However, if the input data is contained in a larger file, then only the unique key values will be extracted and used as the index search keys.

Then, the map() method of the RebuildIndexMapper class reads each record from the input file(s) using its defined key/value arguments. The method passes the values as arguments to an external method. The function of this external method is just to extract and return the values designated for use as the index search keys. After that, the input data is grouped in small files with similar values in the field intended for use in the index.

Then, the map() method concatenates the returned values with the file names extracted in the setup() method to form the unique search key values; else, the returned values from the extra method are used as the search keys. The search key extracted formed together with the input-split for each record are passed to the Pat B+-Tree for index building.

Furthermore, the index building is done by the build() method of the Pat B+-Tree. The build() builds the tree as the Map() method iterates over the input records and passes the key/value pairs to it. A modification was made to the insertion condition in the insertRecord() of the Pat B+-Tree to only allow the insertion of one search key value for a range of keys having the same input-split value. This was done to prevent multiple insertion of input-split values for all of the keys that fall under it, hence, compressing and compacting the index to a smaller size.

In addition, the cleanup() method was used to capture the final state of the tree and to write the subtree as a single object from a chunk of the stored data. The cleanup() method calls the toString() method of the Pat B+-Tree, which in turn, prints the subtree onto temporary file(s) within the HDFS. The output at the mapper stage produces subtrees from the chunks of data processed by different map jobs.

**Searching the Index using MapReduce** The second step for the integration, which was represented by the second block in Fig. 1, is the searching of the index. Figure 3 presents the details of the internal workings for the processes involved in the searching of the index. Here also, the mapper class's only program is used to read from the index file generated by the previous scheme's component, the rebuild index. This simply means that the program only has a mapper called the SearchIndexMapper class.

Also, this mapper class has the same three methods as the previous one, which are: the setup(), the map() and cleanup() methods. The setup() uses its context variable as an argument to read the range of keys, to be searched for within the index, through the context variable's configuration instance, which helps in accepting arguments at runtime.

Then, the map() method of the SearchIndexMapper class reads each record from the index file(s) using its defined key/value arguments to rebuild the tree in the same way as described earlier. This is done to ensure that the tree is rebuilt and retained/cached by the Hadoop for faster searching. Algorithm 1 shows the pseudo-code of the map() method highlighting the functions explained

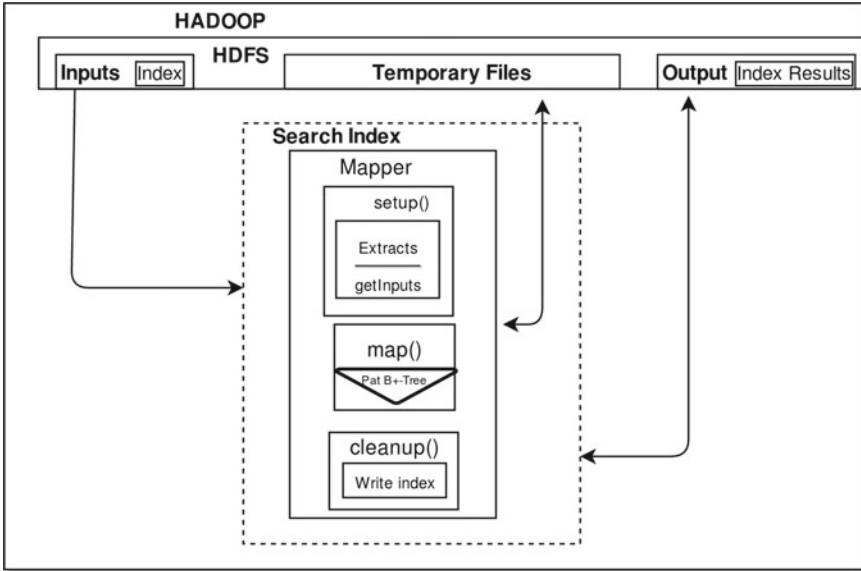


**Fig. 3** Block diagram for the index searching process

---

**Algorithm 1:** Algorithm for SearchIndexMapper Class

---

**Input**: Range of Keys
**Result**: Input-Split values
**SearchIndexMapper extends Mapper**<inkeyType , invalue,outkeyType,outvalueType >{
**Setup**(Context context){ get configuration args at runtime }
**map** (inkey,invalue, contetx){
**if** *indexValue* **then**
  split the indexValue
  **if** *indexValuelength >= 3* **then**
    build Pat B+-Tree
  **end**
**end**
} **cleanup** (Context context){
iterate through the values returned by Pat B+-Tree search
print with context(null, value) }
}

---

After that, the cleanup() method of this class captures the final state of the tree and calls the searchRange() method of the Pat B+-Tree by passing the range of keys from the query's predicates to it. The searchRange() method of the Pat B+-Tree has been modified to search and return the appropriate value that corresponds to any given search key. The search is done by comparing the given search key with a range of separator keys in the tree. This is possible due to the logic used in building and compressing the Pat B+-Tree explained earlier.

Then, the cleanup() method uses its context's write() method to iterate and print the returned values onto the output file(s) within the HDFS as the index's result.

**Using the Index in MapReduce Execution Process:** In order to make use of the index in the main MapReduce query execution process, i.e., query processing of Fig. 1, another complete MapReduce program is required. The program operates in a normal way as an ordinary MapReduce function, with the exception of using a customized FileInputformat. Figure 4 shows the processes, especially the IndexFileInputFormat component.

The query processing uses three classes, namely, The Mapper, the Reducer and the IndexFileInputFormat classes. The Mapper class consists of two methods: the setup() and the map() methods.

The setup() method of the Mapper class uses its context variables' configuration property to get runtime arguments in the same way as explained in the details earlier.

The map() method, on the other hand, iterates through the input data records comparing the keys/values from the records with the lower and upper bounds from the query's predicates. The map() method emits only those records that have met the condition set by the query's predicates.

Then, the class's getSplit() method calls yet another feeder method the readInputSplitFromFile() method that reads the index result which was returned by the SearchIndex class discussed earlier. The readInputSplitFromFile() method iterates

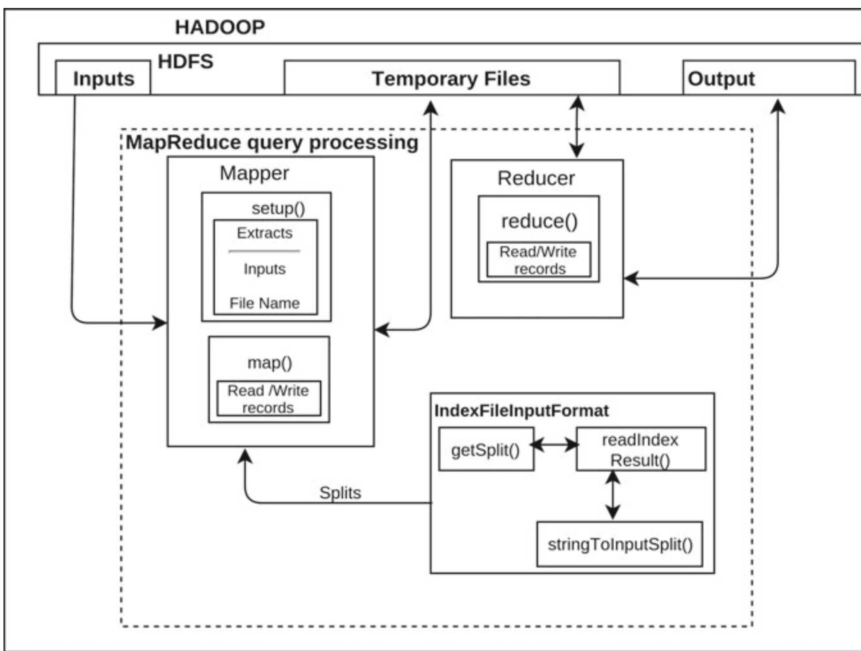


**Fig. 4** Block diagram for integrating the index using IndexFileInputFormat class

through the index of the result entries, which has "null' as the value of all the keys and a string form of the input-splits as their corresponding values. The index results are the ones printed out during the searching of the index using the query's predicates.

The string form of the input-splits is not recognized by the FileInputFormat as valid input-splits, hence another method that transforms the strings into a valid input-split is needed.

# 4 Experimental Results and Discussion

In this section the setup of the experiment as well as the results obtained and their interpretations are discussed.

## *4.1 The Dataset*

In this study, three different datasets were used.

The dataset, which is related to oil and gas domain. Meteorological and Oceanography data (Modata) is the data used by the Oil and Gas (O&G) industries for exploration sites analysis and forecasting. Modata is quite different from the previous datasets discussed, because it comes in multiple small files, as opposed to one large file like the other benchmark datasets. Each file contains data for a single point out of several thousand others in an exploration site.

In addition, the files have varied sizes according to the number of information/columns/parameters that they capture. For instance, files that have the minimum number of columns, i.e., 20 columns, have the size of 60 MB, while files which have maximum number of columns, i.e., 300 columns, have the size of 307 MB.

Furthermore, each tuple in the file represents data taken for a particular hour of the day. For example, in a site called Sea Fine central (SF-Central), there are 14,881 points, and each point will have several tuples of data which are recorded every hour. Since site SF-Central is nearer to the shoreline, there are 20 parameters or features recorded in each tuple such as 'wind direction', 'Wind speed', 'date' and 'time'. Besides raw data, other derived statistical data such as the 'mean' and the 'variation' of the parameters are also included in the data collection.

For sites that are more toward the ocean, the parameters collected will be even more. Moreover, each file contains the tuples described for six decades. From this description of the Modata, one can imagine the hugeness and complexity the data files. Figure 5 shows sample of data from one file of Modata.

On the other hand, for the Modata, a record volume of 10, 20, 40, 80, 160 and 320 million were queried from the data sizes mentioned above, respectively. Also, Table 1 shows the summary of the query sets with their expected returning records.

```
CCYYMM DDHHmm  WD   WS  ETOT   TP  VMD ETTSea TPSea VMDSea ETTSw TPSw VMDSw  MO1  MO2   HS
DMDIR ANGSPR INLINE
195607 010000 107.1 9.04 0.066 4.621 288.5 0.065 4.622 288.5 0.000 6.057 165.9 0.100 0.160 1.024 289.5 0.8748 0.7926
195607 010100 107.8 9.03 0.075 4.854 289.1 0.075 4.854 289.1 0.000 6.047 153.1 0.112 0.177 1.092 290.0 0.8803 0.7996
195607 010200 108.6 9.02 0.082 4.971 289.9 0.082 4.971 289.9 0.000 5.989 132.4 0.122 0.191 1.148 290.7 0.8819 0.8013
195607 010300 109.4 9.01 0.089 5.042 290.6 0.089 5.042 290.6 0.000 5.898 79.7 0.130 0.201 1.192 291.4 0.8816 0.8006
195607 010400 110.2 9.00 0.094 5.114 291.2 0.094 5.114 291.2 0.000 5.847 53.8 0.136 0.209 1.227 292.0 0.8807 0.7990
195607 010500 111.0 8.99 0.098 5.219 291.8 0.098 5.219 291.8 0.000 5.845 44.0 0.141 0.214 1.253 292.5 0.8795 0.7972
195607 010600 111.8 8.99 0.102 5.341 292.3 0.101 5.340 292.2 0.000 5.889 36.9 0.145 0.219 1.274 292.9 0.8784 0.7955
195607 010700 109.4 8.55 0.102 5.422 292.5 0.102 5.421 292.4 0.000 5.940 33.7 0.146 0.220 1.280 293.0 0.8734 0.7885
195607 010800 106.6 8.11 0.100 5.485 292.4 0.099 5.481 292.1 0.001 5.508 334.0 0.142 0.214 1.264 292.9 0.8674 0.7804
195607 010900 103.6 7.68 0.095 5.529 292.1 0.092 5.508 291.1 0.003 5.557 322.1 0.135 0.203 1.235 292.5 0.8607 0.7718
195607 011000 100.1 7.24 0.090 5.561 291.6 0.082 5.285 289.1 0.008 5.643 316.5 0.127 0.190 1.198 292.0 0.8559 0.7659
195607 011100 96.2 6.81 0.084 5.592 291.0 0.069 5.059 286.3 0.015 5.737 312.6 0.118 0.177 1.158 291.5 0.8521 0.7616
195607 011200 91.9 6.37 0.078 5.625 290.5 0.052 4.670 282.8 0.026 5.789 305.3 0.109 0.163 1.117 291.0 0.8493 0.7586
195607 011300 95.3 6.74 0.073 5.660 289.9 0.060 5.113 285.2 0.013 5.839 310.9 0.103 0.154 1.084 290.4 0.8486 0.7583
```

**Fig. 5** Sample of data from Modata files

**Table 1** Data sizes and expected returning records for Modata

| S/no. | Data size | Number of target records |
|---|---|---|
| 1 | 20 GB | 10,000,000 |
| 2 | 50 GB | 20,000,000 |
| 3 | 100 GB | 40,000,000 |
| 4 | 200 GB | 80,000,000 |
| 5 | 500 GB | 160,000,000 |
| 6 | 1 TB | 320,000,000 |

## *4.2 Index Building Using the Datasets*

On the other hand, the search keys for the Modata were formed by concatenating the individual filename to the timestamps. The timestamps are the field with the unique values in each of the files.

However, the same timestamps are found in all other files, which are in their thousands in the Modata dataset. Hence, the search keys in the Modata are lengthier. For the search keys' corresponding values, the proposed index method used the input-split component of the HDFS. The decision to use the input-split comes through the observation made from the reviewed literatures. This is due to the fact that, using any other HDFS components required the authors to develop algorithms to work with the index.

## *4.3 Test Queries*

The queries used in the experiment of this research were formed into sets targeting the restrictive clauses that were relevant to indexing, namely, WHERE, JOIN and

**Table 2** Sample select queries with WHERE, JOIN and GROUP BY for Modata

| Query type no | MOData dataset |
|---|---|
| 1 | select * from Modata.Mdata20 where gpt <= Modata 'SF000150' and datadate >='1956-07-31 00:00:00' AND 'SF000320' and '2000-12-31 00:00:00' distribute by wd |
| 2 | select sc.gpt, sc.wd, sc.datadate, mk.gpt, mk.wd, Modata mk.datadate from Modata.Mdata10a sf JOIN Modata.Mdata10b sc ON sc.datadate = sf.datadate where sc.gpt <='SF000150' and sf.gpt <= 'SF000320' and sc.datadate = '2000-12-31 00:00:00' distribute by sf.wd, sc.wd |
| 3 | select year(sf.datadate), sf.ws, avg(sf.wd) from Mo- Modata data.Mdata20 sf where sf.gpt <='SF000150' and datadate >= '1956-07-31 00:00:00' AND 'SF000320' and '2000-12-31 00:00:00' group by year(sf.datadate) order by dd |

*Note* 'wd' = Wind Direction, 'ws' = Wind Speed, 'gpt' = location 'datadate' = timestamps

GROUP BY. The reason for their choice was due to the fact that they were responsible for determining the data size to be considered in their query process. Furthermore, indexing was also used to restrict the amount of data to take part in the query processing.

The query sets are given in Table 2.

## 4.4 The Experiment and Its Setup

This section describes, in detail, the steps followed in conducting the evaluation of the experiment. The experiment was performed on the datasets mentioned earlier.

The research experiment was setup to evaluate the indexing scheme. The experiment was setup on a cluster of nine nodes, i.e., one Namenode (master) and eight Datanodes (slaves). Each had the following configuration: 8 core CPU, 8 GB RAM, 1 TB HDD, 1 GB bandwidth Ethernet card and running the Ubuntu 12.0.4 (LINUX) Operating System. The platform used was the Hadoop-2.7.1.

## 4.5 Index Creation Performance Evaluation

In order to see how the index creation time of the proposed index fares, it was compared to yet another indexing approach that is based on the same HDFS component of input-splits, but uses clustered index approach. Also, due to the nature of the queries that would use the proposed indexing when created, two different indexes are needed. One for queries, which work on single file, the SELECT … WHERE and SELECT … WHERE… GROUP BY queries, and the other for multiple files queries, that use SELECT… WHERE…JOIN query.

## 5 Results and Discussions

This section presents the experimental results and their discussions for both the index creation and query performance evaluation.

**Modata Dataset Index Creation Performance**

Table 3 shows the execution runtime for both the proposed index and the clustered index approaches to create index for queries on Modata. The same results were presented in Fig. 6 for better view.

From both the table and figure, it can be seen that the index creation for the proposed index was still faster than that of the clustered index. The index for the clustered index takes time more than that of the proposed indexing for the data sizes.

Similarly, Table 4 and Fig. 7 show that the index creation time for all data sizes using the proposed indexing fared better than that of the clustered index when used with join query. The index creation runtime for the clustered index was at least double the time of the creation of the proposed indexing for all the data sizes.

**Table 3** Execution runtime for index creation on single file Modata

| Data size | Index creation time (sec) | |
|---|---|---|
| | Clustered index | Proposed Index Method |
| 20 GB | 1167 | 596 |
| 50 GB | 1673 | 1367 |
| 100 GB | 3860 | 3619 |
| 200 GB | 6731 | 5577 |
| 500 GB | 12209 | 11591 |
| 1 TB | 21821 | 20837 |

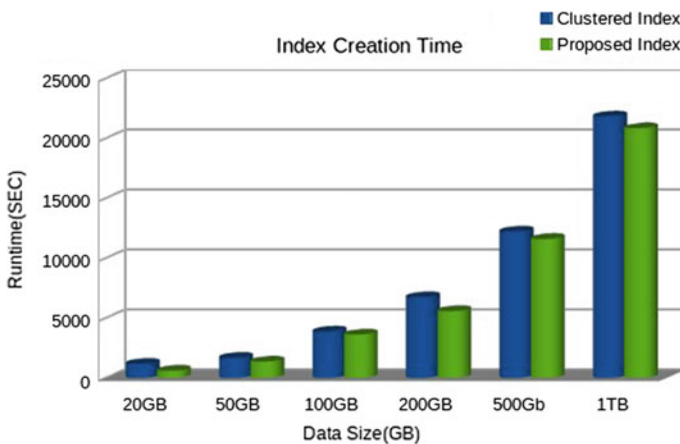


**Fig. 6** Execution runtime for the index creation on single file Modata

**Table 4** Execution runtime for multiple files index creation on Modata

| Data size | Index creation time | |
|---|---|---|
| | Clustered index | Proposed index |
| 20 GB | 1021 | 410 |
| 50 GB | 3025 | 879 |
| 100 GB | 10689 | 1695 |
| 200 GB | 10128 | 3646 |
| 500 GB | 66227 | 8421 |
| 1 TB | 75600 | 17014 |

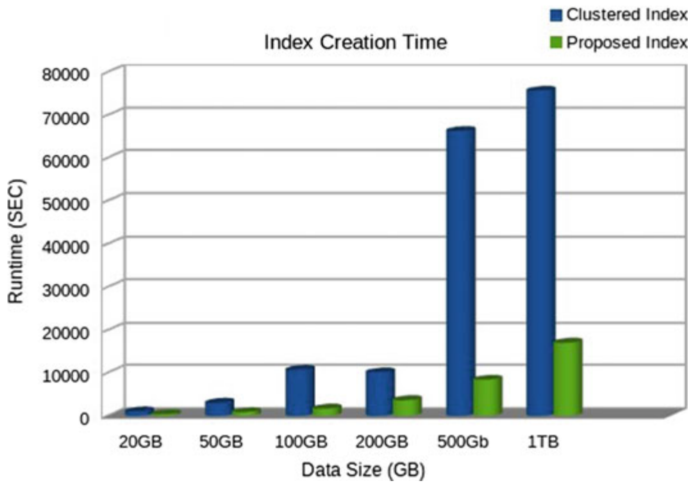


**Fig. 7** Execution runtime for the multiple files index creation when applied to Modata

Thus, from the experiment conducted, we could say that our proposed indexing method performed better by taking lesser time to create its index for both single file query as well as multiple files for all different datasets and different data sizes. As mentioned earlier, this was attributed to the way the approach handled the creation of the index's data structure, the partitioned B+-Tree. The partitioned B+-Tree has an internal sorting that enabled it to compare and store only search keys that have unique corresponding values in the tree. So, for the proposed indexing the major determinant for runtime is the comparison amongst key/value entries, while disk I/O operation was less. On the other hand, the clustered index's runtime is determined by the I/O operation time as all search keys and their corresponding values must be stored.

Also, the high increase in the execution runtime for creating the index noticed, for both index approaches for 500 GB and 1 TB, may be largely attributed to the high increase in the number of map tasks (the total number of map tasks lunched): Data local map task (the total number of map tasks that reside on the same node on

which they are processed) and the rack map tasks (the number of map tasks that need to be transferred to another node for processing). Increase in number of map tasks is translated into increase in time for each of the map tasks to run to its completion. While the increase in the difference between the map tasks launched and that of data local map tasks means there is going to be more of shuffling and sorting, which in turn led to transferring of data across the nodes of the Hadoop cluster, and that also increases the execution runtime.

**Modata Execution Runtime Performance:** Similarly, this subsection also presents the results of running the three queries in Table 2 using the normal MR, the clustered index and proposed indexing. However, the dataset used here is called the Modata dataset. First, the performance of the normal MR was compared with that of the proposed index as well as the clustered index approaches.

**Query Execution Runtimes for the Normal MapReduce and Clustered Index versus Proposed indexing Methods:** Query 1 **in** Table 2
Table 5 shows the results of the MapReduce job execution runtime for the Query 1, i.e., SELECT .. WHERE, on the Modata for all data sizes using the normal Hadoop MapReduce, the clustered index and the proposed indexing approaches. In similar pattern with the previous dataset, Fig. 8 presents the column chart for the results. Based on the results, the normal MR has the highest runtime for all the data since full input scan was used during the query processing. Then, the next highest runtime was that of the clustered index approach.

The runtime of the proposed indexing was 399 s lower than that of the normal MR and 78 s lower than that of the clustered index for 20 GB of data. The runtime was also lower, i.e., 807 s lower than that of normal MR and 31 s lower than that of the clustered index for 50 GB of data. For 100 GB of data size the runtime for the proposed indexing takes 2125 s less than the normal MR and 357 s lower than that of the clustered index. The same goes for 200 GB of data; the runtime of the proposed indexing takes lesser time, i.e., 2575 s to complete than that of normal MR as well as 1044 s lower than that of the clustered index. The runtime of the proposed indexing was 143 s lower than that of the normal Hadoop MR and 11 s lower than that of the clustered index for 500 GB of data. Then, for 1 TB data size, the runtime for the

**Table 5** Query 1 execution runtimes for the normal MR versus clustered index versus proposed index methods (Modata)

| Data size | Query 1 execution runtime (sec) | | |
|---|---|---|---|
| | Normal MR | Clustered index | Proposed index method |
| 20 GB | 724 | 403 | 325 |
| 50 GB | 1641 | 865 | 834 |
| 100 GB | 3578 | 1810 | 1453 |
| 200 GB | 7236 | 5705 | 4661 |
| 500 GB | 12924 | 12892 | 12781 |
| 1 TB | 22451 | 16010 | 15909 |

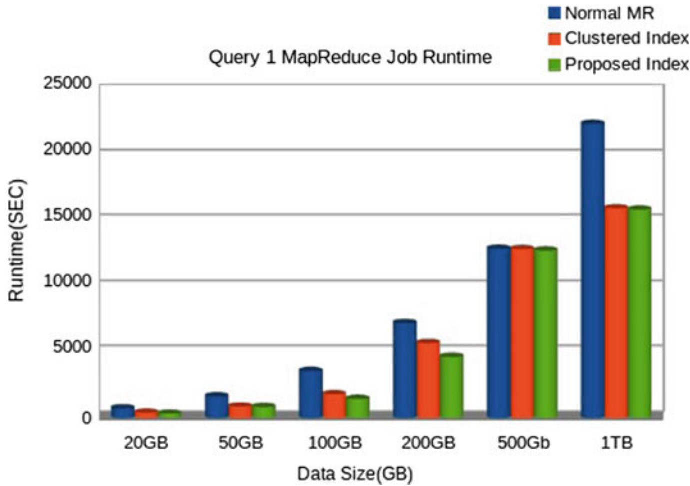


**Fig. 8** Query 1 execution runtimes for the normal MR versus clustered index versus proposed indexing methods (Modata)

proposed indexing was also 6542 s lower than the normal MR and 101 s lower than that of the clustered index. The trend of the runtime increased with the increase of the data size significantly for both approaches. However, the time difference between the clustered index and proposed indexing was not so wide. This was due to the small files nature of the Modata. This may require more resources for their handling as the number of input-splits increased significantly.

**Query 2** in Table 2
Table 6 presents the Query 2 type, i.e., SELECT .. WHERE .. GROUP BY query, job processing runtime for all data sizes using the normal MR, the clustered index and the proposed indexing approaches, and Fig. 9 gives the results in column chart form.

**Table 6** Query 2 execution runtimes for the normal MR versus clustered index versus proposed indexing methods (Modata)

| Data size | Query 2 execution runtime (sec) | | |
|---|---|---|---|
| | The normal MR | Clustered index | ISPB method |
| 20 GB | 744 | 279 | 176 |
| 50 GB | 1711 | 555 | 404 |
| 100 GB | 3850 | 801 | 633 |
| 200 GB | 5707 | 2213 | 2141 |
| 500 GB | 12892 | 4700 | 4607 |
| 1 TB | 19562 | 4841 | 4580 |

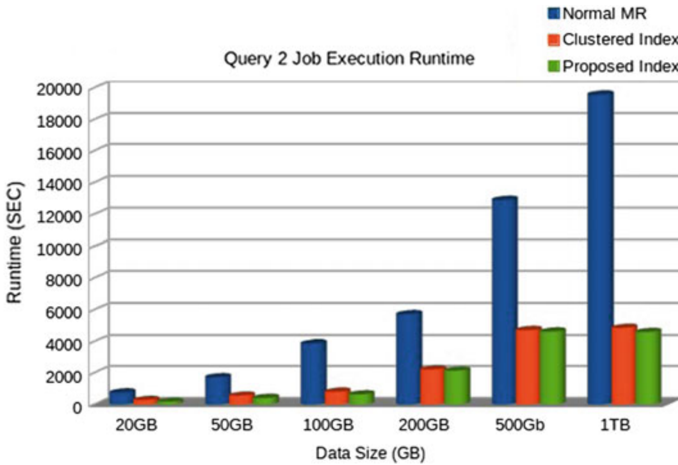


**Fig. 9** Query 2 execution runtimes for the normal MR versus clustered index versus proposed indexing method (Modata)

The runtime results for individual data sizes are generally lower here than those of the Query 1. This was so, because the returned record set for the GROUP BY query was less voluminous. For 20 GB of data size the runtime for the proposed indexing method takes 568 s lesser than that of the normal MR and 103 s lower than that of the clustered index. For 50 GB and 100 GB of data sizes the runtime of the proposed indexing are lower by 1307 s and 3217 s lower than that of the normal MR and 151 s and 168 s lower than that of the clustered index, respectively. As for 200 GB of data size, the runtime was lower by 3566 s than that of the normal MR and 72 s lower than that of the clustered index. At the same time, the runtime of proposed indexing was 8825 s lower than the normal MR and 633 s lower than that of the clustered index for 500 GB of data. Lastly, for 1 TB data size, the runtime for the proposed indexing was also 14982 s lower than the normal MR and 261 s lower than that of the clustered index. Again, the proposed indexing outperformed the normal MR and the clustered index in all cases.

**Query 3** in Table 2
Running Query 3, which was a query with JOIN clause using both normal MR, clustered index and the proposed indexing produced results as in Table 7. The same results are also presented in a chart form in Fig. 10.

The results show that the normal MR has the longest runtime for all the data sizes as compared to the proposed indexing. This was due to the complete scanning of the input during the query processing, followed by the clustered index. The runtime of proposed indexing method was 198 s lower than that of the normal MR and 189 s lower than that of the clustered index for 20 GB of data. The runtime was also 103 s lower than that of the normal MR and 293 s lower than that of the clustered index for 50 GB of data. For 100 GB of data size, the runtime for the proposed indexing was

**Table 7** Query 3 execution runtimes for the normal MR versus clustered index versus proposed indexing method (Modata)

| Data size | Query 3 execution runtime (sec) | | |
|---|---|---|---|
| | Normal MR | Clustered index | ISPB method |
| 20 GB | 569 | 560 | 371 |
| 50 GB | 795 | 985 | 692 |
| 100 GB | 1866 | 1238 | 852 |
| 200 GB | 3269 | 2091 | 1742 |
| 500 GB | 10199 | 6061 | 5865 |
| 1 TB | 23488 | 11714 | 9727 |

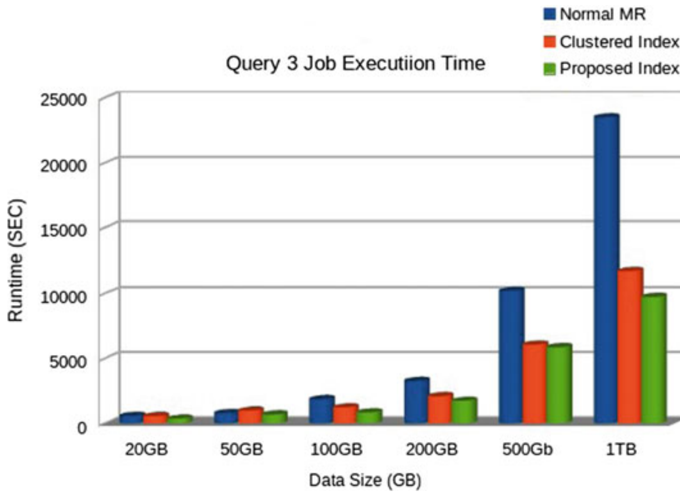


**Fig. 10** Query 3 execution runtimes for the normal MR versus clustered index versus proposed indexing methods (Modata)

1014 s lower than that of the normal MR and 386 s lower than that of the clustered index. Similar trend was observed for 200 GB of data size, where the runtime of the proposed indexing was 1527 s lower than that of the normal MR and 349 s lower than that of the clustered index. Furthermore, the runtime of the proposed indexing method takes 4334 s less than that of the normal MR and 197 s lower than that of the clustered index for 500 GB of data. Then, for 1 TB data size, the runtime of the proposed indexing was 13761 s lower than that of the normal MR and 1987 s lower than that of the clustered index. The trend of the runtime increased significantly with the increase of the data size for both approaches.

In summary, after all these experiments have been conducted, we can say that query processing runtime becomes faster with the involvement of index in all cases.

## 6 Conclusion

As stated earlier, the index created and used by the proposed index was found to be far smaller than those created by clustered index approaches reported in this study. The index of proposed index was at least 1000 times smaller than those produced by other approaches. The reason for a smaller index size was first due to the fact that the data structure used by the proposed index. The partitioned B+-Tree enabled the building of the index using only the search keys, with unique input-splits as a representative of other search keys as the key value pair entry to the tree.

Secondly, the HDFS component chosen for the indexing in the proposed indexing, which was the input-split, helped in reducing size of both index and the input data. By default, the input-split in the Hadoop MapReduce process was what determined which part of the input data will take part in the query processing. Also, the input-split determines the number of mappers that the Hadoop needed to deploy to execute certain task.

Moreover, another reason for the better performance on the side of the proposed indexing during the query processing was due to the restriction brought about by the proposed index of processing only those input-splits that are returned by the index component of the scheme.

In summary, the proposed indexing has through its data structure the Pat B+-Tree, algorithms and approach (proposed index) has been able to significantly improve the MapReduce's query processing capability.

## References


1. Abdullahi, A.U., Ahmad, R., Zakaria, M.N.: Experimental performance analysis of B+-trees with big data indexing potentials. In: International Conference of Reliable Information and Communication Technology, pp. 20–29. Springer (2017)
2. An, M., Wang, Y., Wang, W.: Using index in the mapreduce framework. In: Web Conference (APWEB), 2010 12th International Asia-Pacific, pp. 52–58. IEEE (2010)
3. B+-tree, B-tree: http://scienceblogs.com/goodmath/2008/07/06/btrees-balancedsearch-trees-f/ (2016)
4. Cao, J., Han, H., Zhao, M., Ye, S., Zhu, D., Li, L.: An optimized method oftranslating sql to more efficient map-reduce tasks. Int. J. Grid Distrib. Comput. **8**(4), 249–256 (2015)
5. Chaudhuri, S., Narasayya, V.: Self-tuning database systems: a decade of progress. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 3–14. VLDB Endowment (2007)
6. Chen, C.P., Zhang, C.Y.: Data-intensive applications, challenges, techniques andtechnologies: a survey on big data. Inf. Sci. **275**, 314–347 (2014)
7. Chen, M., Mao, S., Liu, Y.: Big data: a survey. Mobile Netw. Appl. **19**(2), 171–209 (2014)
8. Dean, J., Ghemawat, S.: Mapreduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
9. 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)
10. Glombiewski, N., Seeger, B., Graefe, G.: Waves of misery after index creation. BTW 2019 (2019)

11. Graefe, G.: Sorting and indexing with partitioned b-trees. CIDR **3**, 5–8 (2003)
12. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 371–381. ACM (2010)
13. Hadoop, A.: Apache hadoop. http://hadoop.apache.org/ (2017)
14. He, J., Yao, S.w., Cai, L., Zhou, W.: Slc-index: A scalable skip list-based indexfor cloud data processing. J. Central South Univ. **25**(10), 2438–2450 (2018)
15. Hong, Z., Xiao-Ming, W., Jie, C., Yan-Hong, M., Yi-Rong, G., Min, W.: A optimized model for mapreduce based on hadoop. TELKOMNIKA (Telecommunication Computing Electronics and Control) **14**(4) (2016)
16. Ibrahim, H., Sani, N.F.M., Yaakob, R., et al.: Analyses of indexing techniques onuncertain data with high dimensionality. IEEE Access **8**, 74101–74117 (2020)
17. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In: CIDR. vol. 7, pp. 7–10 (2017)
18. Khasawneh, T.N., AL-Sahlee, M.H., Safia, A.A.: Sql, newsql, and nosql databases: a comparative survey. In: 2020 11th International Conference on Information and Communication Systems (ICICS), pp. 013–021 (2020)
19. Lee, S., Jo, J.Y., Kim, Y.: Performance improvement of mapreduce process bypromoting deep data locality. In: Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on, pp. 292–301. IEEE (2016)
20. McCreadie, R., Macdonald, C., Ounis, I.: On single-pass indexing with mapreduce. In: Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 742–743. ACM (2009)
21. McCreadie, R., Macdonald, C., Ounis, I.: Mapreduce indexing strategies: Studyingscalability and efficiency. Inf. Process. Manage. **48**(5), 873–888 (2012)
22. Mofidpoor, M., Shiri, N., Radhakrishnan, T.: Index-based join operations in hive. In: Big Data, 2013 IEEE International Conference on, pp. 26–33. IEEE (2013)
23. Philip Chen, C., Zhang, C.Y.: Data-intensive applications, challenges, techniquesand technologies: a survey on big data. Information Sciences **275**, 314–347 (2014) 24
24. Ramakrishnan, R., Gehrke, J., Gehrke, J.: Database management systems, vol. 3. McGraw-Hill New York (2010)
25. Richter, S., Quian´e-Ruiz, J.A., Schuh, S., Dittrich, J.: Towards zero-overhead staticand adaptive indexing in hadoop. VLDB J. **23**(3), 469–494 (2014)
26. Roy, S., Mitra, R.: A survey of data structures and algorithms used in the contextof compression upon biological sequence. Sustain. Humanosphere **16**(1), 1951–1963 (2020)
27. Rys, M.: Xml and relational database management systems: inside microsoft sqlserver 2005. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pp. 958–962. ACM (2005)
28. Sevugan, P., Shankar, K.: Spatial data indexing and query processing in geocloud. J. Testing and Eval. **47**(6) (2019)
29. Silberschatz, A., Korth, H.F., Sudarshan, S., et al.: Database system concepts, vol. 4. McGraw-Hill New York (1997)
30. Silva, Y.N., Almeida, I., Queiroz, M.: Sql: From traditional databases to big data. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education, pp. 413–418. ACM (2016)
31. Statista: Volume of data worldwide from 2010-2025. https://www.statista.com/statistics/871513/worldwide-data-created/ (2020)
32. Stewart, R.J., Trinder, P.W., Loidl, H.W.: Comparing high level mapreduce querylanguages. In: Advanced Parallel Processing Technologies, pp. 58–72. Springer (2011)
33. Suman, A.K., Gyanchandani, M.: Improved performance of hive using index-basedoperation on big data. In: 2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS), pp. 1974–1978. IEEE (2018)
34. Yang, H.C., Parker, D.S.: Traverse: simplified indexing on large map-reduce-mergeclusters. In: International Conference on Database Systems for Advanced Applications, pp. 308–322. Springer (2009)

35. Zhang, Q., He, A., Liu, C., Lo, E.: Closest interval join using mapreduce. In: DataScience and Advanced Analytics (DSAA), 2016 IEEE International Conference on, pp. 302–311. IEEE (2016)
36. Zikopoulos, P., Eaton, C.: Understanding big data: analytics for enterprise classhadoop and streaming data. McGraw-Hill Osborne Media (2011)