A MapReduce-based scalable discovery and indexing of structured big data

Article *in* Future Generation Computer Systems · April 2017

DOI: 10.1016/j.future.2017.03.028

2 authors:

Hari Singh
Jaypee University of Information Technology
21 PUBLICATIONS **51** CITATIONS
SEE PROFILE

Seema Bawa
Thapar University
131 PUBLICATIONS **736** CITATIONS
SEE PROFILE

Future Generation Computer Systems 73 (2017) 32–43

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A MapReduce-based scalable discovery and indexing of structured big data

Hari Singh *, Seema Bawa

Computer Science & Engineering Department, Thapar University, Patiala, Punjab, India

HIGHLIGHTS

- Solution for the lack of indexing in the MapReduce frameworks – Hadoop is proposed.
- Parallel implementation of existing B-Tree index is carried out in MapReduce.
- Parallel implementation of range search query is carried out in MapReduce.
- Cluster size, query coverage, number of map tasks and size of I/O data are analyzed.
- HDFS block size and, heap memory and intermediate data generated are analyzed.
- Proposed parallel B-Tree index provides high scalability and efficient data search.

ARTICLE INFO

Article history:
Received 15 March 2016
Received in revised form
10 March 2017
Accepted 25 March 2017
Available online 1 April 2017

Keywords:
Hadoop
Distributed computing
MapReduce
HDFS
Cluster
B-Tree

ABSTRACT

Various methods and techniques have been proposed in past for improving performance of queries on structured and unstructured data. The paper proposes a parallel B-Tree index in the MapReduce framework for improving efficiency of random reads over the existing approaches. The benefit of using the MapReduce framework is that it encapsulates the complexity of implementing parallelism and fault tolerance from users and presents these in a user friendly way. The proposed index reduces the number of data accesses for range queries and thus improves efficiency. The B-Tree index on MapReduce is implemented in a chained-MapReduce process that reduces intermediate data access time between successive map and reduce functions, and improves efficiency. Finally, five performance metrics have been used to validate the performance of proposed index for range search query in MapReduce, such as, varying cluster size and, size of range search query coverage on execution time, the number of map tasks and size of Input/Output (I/O) data. The effect of varying Hadoop Distributed File System (HDFS) block size and, analysis of the size of heap memory and intermediate data generated during map and reduce functions also shows the superiority of the proposed index. It is observed through experimental results that the parallel B-Tree index along with a chained-MapReduce environment performs better than default non-indexed dataset of the Hadoop and B-Tree like Global Index (Zhao et al., 2012) in MapReduce.

1. Introduction

Over the last several years, a huge amount of data has been generated all over the world as a result of existing automated computer applications. Advancement in computer networks and development of distributed applications such as, e-commerce and social networks have given a faster pace to the generation of data. It is very difficult to process large amount of structured and unstructured data with traditional sequential programming methods. Current strategies, methodologies and architectural models are not sufficient for dealing with big data. Query processing, data modeling and analysis have been some of the crucial challenges to deal with big data [1]. The MapReduce programming model has emerged as a large-scale data analytic applications [2–4]. The MapReduce-based powerful and simple architecture of Hadoop provides good performance for large amount of data processing, when configuration parameters are calibrated properly [5–7]. However, the performance of MapReduce is affected by I/O [8], data locality issues [9], scheduling strategies [10,11], and indexing over input dataset [12–17]. Several factors contribute towards improving MapReduce, such as data access, avoidance of redundant processing, early termination, it-

* Corresponding author.
E-mail addresses: harirawat@rediffmail.com (H. Singh), seema@thapar.edu (S. Bawa).

http://dx.doi.org/10.1016/j.future.2017.03.028
0167-739X/© 2017 Elsevier B.V. All rights reserved.

erative processing, query optimization, fair work allocation, interactive real-time processing, and processing *n*-way operations [18].

A lot of research has been available on MapReduce that improves the performance of Hadoop through modification in its architecture. HadUP-Hadoop with Update Processing presents a modified Hadoop architecture that provides high efficiency for large-scale analytic processing in datasets evolving over time [19]. It detects changes in the dataset at fine-grained level and processes only the changed data and avoids repeated computation on old data. It uses a deduplication-based snapshot differential algorithm (D-SD) which extends the principle of sparse indexing [20] and update propagation in conventional MapReduce algorithms. In another modification to the Hadoop architecture, Flame-MR optimizes the use of memory and Central Processing Unit (CPU) resources through an event-driven architecture. It reduces large data transfer in map- and reduce-stage to the disk through pipelining. The pipelining provides overlapping of data processing with a disk and network transfer, and consequently improves execution efficiency of the system [21].

In another modification to Hadoop architecture, indexes have been integrated on top of default key-value storage of Hadoop. The inverted index is a well-used secondary index for text processing on MapReduce [22,15]. Cassandra supports distributed hash indexes on non-key attributes [23]. The hash indexes support simple equality tests like operations and are faster than B-Tree indexes, however, B-Tree indexes support all comparison operators. But it is an offline index and therefore other online indexes, such as B-tree were explored for MapReduce.

B-Tree index improves data access in a distributed system [24–28]. An efficient indexing framework in a distributed environment provides indexes similar to indexes being used in DBMSs in a centralized environment [29]. It allows users to select multiple distributed indexes using Cayley Graphs, without caring about the underlying network that supports key-based data retrieval. The Cayley Graph reduces index build-time and maintenance cost and allows users to define their indexes. Three types of indexes, distributed hash index, B+-Tree index and distributed multidimensional index, are implemented to provide Database-as-a-Service "DaaS" in a cloud environment. A similar framework ScalaGiST—Scalable Generalized Search Tree processes large analytic queries in the MapReduce system [30]. Hadoop has a sub-optimal performance for random access queries as it does not use any indexing. ScalaGiST provides a general indexing framework that supports indexing such as B+-Tree and R-Tree in distributed environment, without affecting the code of MapReduce-Hadoop. The framework is extensible for data and queries, and provides better efficiency, scalability and fault tolerance.

B-Tree and B+-Tree indexes have been researched for data access optimization of structured data for distributed network and MapReduce clusters [24–27]. A distributed B+-tree index (ScalableBtree) over a data sharing service, Sinfonia, uses a combination of a client library, a set of fault tolerant memory nodes (servers) and mini-transactions that perform atomic updates at servers [31]. The B+-tree is constructed by distributing nodes over the Sinfonia servers. Transactions in the B+-tree that are accomplished with multiple operations are carried out atomically with Sinfonia's mini-transactions. The generated B+-Tree is highly balanced and online management is quite easy, as nodes can be moved from one server to another while a B+-Tree request is in process. The method reduces data transfer, achieves a high concurrency and scales well for hundreds of machines for queries. However, the method is efficient for only simple lookup queries and lacks in efficiency for range search queries.

In a two-tier index, a global CG-Index and local B+-tree index, the information is kept about all local B+-tree indexes for efficient data processing in the cloud [25]. A large dataset is partitioned in cluster and all compute nodes (servers) build a local B+-tree index on the local data. The method uses a structured peer-to-peer BATON [32] network routing protocol between servers to share B+-tree nodes of neighboring servers and achieves high throughput and concurrency. The method uses data replication on multiple servers to improve availability. The CG-Index is found to have a higher throughput for range search queries than the ScalableBTree [24]. The efficiency of the B-Tree index is improved in a cloud environment by improving the degree of concurrency on the basis of node split history and minimizing the cost of update operation through regularly adjusting the node size [27].

The most relevant work that considers B-Tree like algorithm, binary search, in MapReduce is presented in [26]. It uses built-in fault-tolerance and concurrency management features of Hadoop that abstracts complex operations required for parallelization. It reduces range search query time by reducing the search space. Firstly, input dataset is reduced for a search query by including only required fields quoted in the query. Secondly, reduced dataset is distributed on computing nodes with a specific range of indexed fields which builds the Global Index. The technique decomposes input dataset into a number of subsets and manages a Global Index for all distributed subsets over cluster nodes. Range search query uses the Global Index with a binary search algorithm. It reduces the number of map tasks during range query, and consequently, number of I/O and the overhead of task scheduling. It results in reduced query response time and improved resource utilization. The authors compared non-indexed and Global Index based dataset in MapReduce for studying the effect of cluster size and range query size on number of map tasks generated. It was found that the number of map tasks remains constant for a change in query range size and depends only on the size of input dataset for non-indexed dataset, while the number of map tasks shows a linear relationship with query range size. A similar pattern was observed for the number of I/O required for non-indexed and global indexed dataset. The Global Index dataset outperforms a non-indexed dataset for query response time. It is due to reduced number of map tasks that cause low I/O overhead.

This paper proposes to optimize data access on MapReduce with a B-Tree index for structured data, and minimizes range search query execution time. The presented B-Tree index in MapReduce-Hadoop encapsulates the complexity of fault tolerance and parallelization from users, which were handled through much complex operations in earlier works, such as [26]. A proposed B-Tree index, consisting of local B-Tree indexes in computing nodes, reduces range search time. The range search query in a chained-MapReduce environment improves intermediate data access time by consecutive map and reduce functions. Map phase of job1 distributes input dataset among cluster nodes and reduce phase of job1 constructs B-Tree index of local datasets on data nodes. Map and reduce phases of job2 performs a key search over B-Tree indexed dataset. This paper thoroughly analyzes the effect of performance metrics of Hadoop clusters on range search query time. The metrics considered are cluster size (number of computing nodes), range search query coverage, varying HDFS block size of a variable cluster size. Java Virtual Machine (JVM) heap space and, the number of intermediate files and directories generated during processing are also analyzed on a variable cluster size and HDFS block size. The experimental results validate better efficiency of range search query in indexed dataset, especially the B-Tree index over default non-indexed dataset of the Hadoop and B-Tree like Global Index in MapReduce [26]. It is also observed that index-build time and query response time are dependent on the Hadoop performance metrics, and reaches its threshold value for a particular value of these metrics.

The rest of this paper is organized as follows. Section 2 describes the significance of MapReduce-Hadoop and indexing data in the

Hadoop. Section 3 presents a conceptual view of a parallel B-Tree index construction in MapReduce. Section 4 describes the implementation part of the parallel B-Tree index construction and range search queries in MapReduce. Section 5 presents a performance evaluation through experimental design and results. Section 6 presents conclusions and future scope of the work.

2. MapReduce and indexing

This work integrates a B-Tree index in MapReduce that results in a parallel B-Tree index. The significance of MapReduce-Hadoop and indexing in the Hadoop is described below.

2.1. MapReduce

Large volume of data has led to adoption of parallel processing. It provides an efficient processing over a set of collaborating computer machines. It was predicted that parallel processing would provide new ways of thinking about the existing concept of programming language, operating system and storage system for large distributed systems [33]. Parallel processing is complicated, but many frameworks have evolved that provide parallel processing using abstraction to simplify things. Hadoop, a Java implementation of MapReduce, has emerged as one such framework [2]. It works on key-value storage concept and has mainly two components, MapReduce and HDFS [34]. MapReduce part of the Hadoop encapsulates all details of parallel processing from users and they get a very simplified framework for programming. MapReduce has become very popular for parallel processing of arbitrary data. It works on the divide-and-conquer strategy and breaks a computation into sub-computations over a set of computers in a cluster that operate in parallel. Each smaller computation is handled separately and the result of the computation is returned back at a central point. A detailed survey of approaches that support MapReduce for distributed data management and processing, such as MapReduce implementations, High level language support for MapReduce, MapReduce implementation on database operators, DBMS implementation in MapReduce, MapReduce extensions for data-intensive applications, have been discussed [35].

A MapReduce application takes data in the key-value form from HDFS and processes it. It works through two popular functions: map and reduce. The map function accepts input data in the key-value form and produces some intermediate data. Once the map function is finished, reduce function starts. The reduce function takes as input the intermediate data having same key and produces output data that is written back to HDFS. Many map and reduce functions work concurrently on different splits of the input dataset in HDFS. A large amount of data transfer takes place between the map and reduce functions when intermediate data is produced. A combiner function can be used to reduce intermediate data and data transfers by aggregating data on the basis of intermediate key.

2.2. Indexing data with Hadoop

Initially, MapReduce has been used for large scale data intensive applications for information retrieval from semi-structured and unstructured data. The MR-LSI algorithm retrieves scalable information from the unstructured documents quite efficiently [36]. It has been used successfully in the domain of structured data for expressing queries [7,31,37,38]. Hadoop by default stores data in key-value form and distribution of data over computing nodes takes place through a hash function applied on keys. The hash function acts as the main index for quickly accessing data [39–41]. However, search over non-key data requires a secondary index for fast accessing. In the absence of a proper secondary index, map tasks are wasted on the undesired dataset. A proper indexing over the input dataset better organizes the dataset, and minimizes computation. It utilizes map tasks for only the desired and filtered dataset, and consequently, improves query resource time and saves system resources. A description of the concept over indexed dataset is presented in Fig. 1, where a job in the form of a query "Select * from the dataset where a particular condition remains, true" runs over the partitioned input dataset on data blocks in the Hadoop cluster. The query has a restricted access to only the green data blocks "B" where condition described becomes true and other red data blocks "A" remains untouched where condition described becomes false. It reduces unwanted I/O disk accesses for data blocks that are not required to answer queries, and consequently, improves query throughput. However, in the absence of an index structure over dataset, though the query gives correct output, but it accesses all data blocks, and query efficiency becomes low. So, an external secondary index over non-key attribute highly improves the performance.

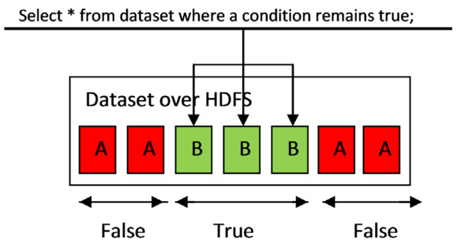

Fig. 1. The effect of indexing on MapReduce processing. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3. Parallel B-Tree index in MapReduce

This section provides a conceptual view of a parallel B-Tree index construction in MapReduce. Firstly, local B-Tree indexes are created on partitioned datasets in datanodes. Secondly, the masternode combines all the local indexes to realize a complete B-Tree index in MapReduce.

3.1. Constructing a local B-Tree index in a datanode of cluster

The traditional B-Tree is a tree structure whose node can hold M records and contains $M + 1$ child pointers to other B-Tree nodes [42]. The order of the tree thus constructed is $M + 1$. Here, we describe the construction of a B-Tree index in a datanode of the cluster. The chunk size of the datanode is taken 64 MB for explaining the B-Tree construction process. A B-Tree node is constructed for a dataset taken from "ResultFile" schema, shown in Table 2 of Section 5, but examinee-id field "E_ID" and question-number "Q_No" are combined to act as a unique key of the B-Tree being constructed. The size of one such tuple is 157 Bytes. An HDFS block of size 64 MB can contain 397 records of 157 Bytes and 398 pointers of 4 Bytes ($64\text{ MB}/(157 + 4) = 397$). Records ($157 \times 397 = 62\,329$ Bytes) and pointers ($158 \times 4 = 632$ Bytes) utilize 64 MB of data chunk. Therefore, a B-Tree node, in a B-Tree of order 157, with ($64\,000-(62\,329+632)) = 79$ spare Bytes in each data chunk, can be constructed. The dataset being considered is not distributed uniformly across the cluster. However, we assume that each datanode gets an equal share of data. In this case, each datanode gets tuples containing any two particular questions for all examinees. However, if data distribution is not uniform, still the process of B-Tree construction remains the same on different amounts of data in datanodes.

Insertion in a B-Tree node that has vacant space to accommodate a new entry is simple, the new entry settles down after rearranging all entries in a sorted order. However, insertion process is different when a new entry arrives to an already full node. The node is split and all the entries are sorted in an intermediate storage. The middle one moves upward while all right side entries are moved to a new node and all left side entries stay in old node. There are two cases for a middle entry that depends on whether node being split is a root node or an internal node. If the split node is a root node, a new blank node is created and middle entry is moved there. If the split node is an internal node, middle entry is moved to its parent node. Now, the middle entry may get a space in parent node or it may see an already full node, in such a case, the process repeats until the entry gets a proper place.

We now discuss how a local B-Tree is constructed on a datanode that contains all data for Q_No 1 and 2. Similarly, local B-Tree index is built on all the participating datanodes. As 397 records can be stored in each B-Tree node, so the first 397 records, 1R1, 1R2, . . . , 1R397, are saved directly in the first root B-Tree node, as shown in Fig. 2(a). Here, we have used the notation *xRy* for the unique key field, where *x* and *Ry* represents Q_No and E_ID, respectively.

When 1R398 is to be stored, the node is split and tuples 1R1, . . . , 1R199 and 1R201, . . . , 1R398 go to left node and right node, respectively. The tuple 1R200 goes to a newly constructed root node as the first tuple of the root node. Subsequent entries 1R399 to 1R 597 can be easily stored in the right node till the count reaches 397. This whole process is shown in Fig. 2(b).

Subsequent entry 1R598 causes the right node overflow and it is split into two nodes, each node gets 1R201, . . . , 1R399 and 1R401, . . . ,1R598. The middle entry 1R400 goes in the root node. This process continues and tuples at positions 20th, 21st, 40th, 381st and 297th are shown in Fig. 2(c). The left and right side pointer of tuple 21st in the root node points to the node containing tuples 2R1, . . . ,2R199 and 2R201, . . . ,2R399 respectively. Similarly, the left pointer of the 40th tuple in root node points to the node containing 2R3801, . . . , 2R3999 and right pointer points to node containing 2R4001, . . . ,2R4199. When 397th entry is filled-up in the root node, it becomes full and subsequent entry would causes node overflow and split the root node. It happens when the right pointer of 397th tuple of root node, 20R3400, that points to a child node, overflows and is split. The split of this child node takes place when a tuple 20R3798 arrives. It causes split of the child node and two new nodes with data 20R3401, . . . ,20R3599 as left node and 20R3601, . . . ,20R,3798 as right node come into existence. The middle tuple 20R3600 moves into the root node. But the root node being already full, and so cannot avoid its split. The complete flow is shown in Fig. 2(c).

At this point, the current root node gets split into three nodes—two left and right nodes and one middle entry 10R4000 moves into a new root node. The left and right nodes have tuples 1R200, . . . ,10R3800 and 11R200,…,20R3600 respectively. Left pointer of 20R3600 points to the node containing tuples 20R3401, . . . ,20R3599 and right pointer points to node containing tuples 20R3601, . . . ,3797. Now, the rest of the entries from 20R3799 to 20R4000 are also filled-up. Tuples 20R3799 to 20R3997 are adjusted in node already containing tuples 20R3601, . . . ,20R3798. However, arrival of 20R3998 causes split of this node into two left and right nodes containing tuples 20R3601, . . . ,20R3799 and 20R3801, . . . ,20R3998 respectively. The middle tuple 20R3800 moves up into the parent node. Now the last two tuples, 20R3999 and 20R4000 are also adjusted in the node with tuples 20R3801, . . . ,20R3998, and this node now contains tuples 20R3801, . . . ,20R3997, 20R3998, 20R3999, 20R4000. The whole process ends here and a local B-Tree is obtained, as shown in Fig. 2(d).

The efficiency of the constructed B-Tree is high, as during the whole insertion process, either all the nodes are always half full or more than half full, except the root node. A record search with a particular key starts with reading block containing the root into memory. All key values are examined sequentially in the node and whenever a record with a key between two values is required, it moves down to a child node. This process is continued until search is complete. However, if a leaf node is reached and specified key is not found, then it means search is unsuccessful.

The efficiency of a B-Tree is much higher than Binary Tree that can point to only two child nodes. A B-Tree is quite small in height due to high node capacity that makes it efficient, as only few disk accesses are required for a search query. In a B-Tree, efficiency increases with increasing node capacity.

3.2. Constructing a parallel B-Tree index in a Hadoop cluster

Input dataset, either in a single file or in multiple files in a directory, is put on the HDFS in sorted order on a composite index field chosen from the many fields of the dataset. The sorted dataset gets decomposed into data chunks of 64 MB to fit in HDFS data blocks of 64 MB. The size of data blocks is configured on the basis of available RAM size in datanodes of the cluster. Accordingly, it is set to a variable size of 64, 128, 256 or 512 MB. The Master node or Namenode of the cluster maintains a proper record of this data distribution among the cluster nodes. The master node uses inbuilt fault tolerance of the Hadoop and creates replica(s) of distributed data blocks. The number of replicas to be created is configured at the time of cluster set-up. Now, the task of constructing a B-Tree index on the whole dataset gets reduced to sub-tasks of constructing local B-Tree indexes on datanodes.

The distribution of data among datanodes has already been discussed in Section 3.1. Here Q_No field is the key and all other fields of the dataset become value. The map function processes partitioned dataset in the key-value form and produces intermediate data. Once some map tasks are finished, reduce tasks of the JobTracker are started. Each reduce task takes all intermediate records with the same key as input and merges it with first parameter (E_ID) of value part. The combination (Q_No+E_ID) acts as unique key of the B-Tree index to be constructed. It has been assumed that each datanode gets equal amount of data. It means that for a dataset of 200 questions, for all examinees, each datanode gets tuples containing 20 questions. So datanodes Node1, Node2, …, Node10 build the index on the partitioned dataset ranging from 1R1-20R4000, 21R1-40R4000, …, 181R1-200R4000, respectively. A physical view of parallel B-Tree construction in the cluster is shown in Fig. 3.

Logically, JobTracker in Masternode parallelizes B-Tree construction and query execution across datanodes. TaskTrackers in datanodes manages the construction of a local B-Tree. When a job (B-Tree construction) is submitted to Masternode, the JobTracker starts job's map tasks. Each map task independently applies a map function to its split (64 MB data chunk) and makes key-value pair as intermediate data. The intermediate data are sorted and stored into a local disk using a defined partition function. It causes a lot of data transfer between map and reduce stages that consumes a significant time. The formation of a complete B-Tree index from local B-Tree indexes is shown in Fig. 4.

Datanodes Node1, Node2, . . . , Node10 store a part of the distributed dataset. Node1 builds a local B-Tree index for tuples 1R1 to 20R4000, as described in Section 3.1. Similarly, other nodes also build local B-Tree indexes. Lastly, a pointer is returned by each Datanode to the Masternode. The Masternode keeps a record to access root pointer of each local B-Tree index.

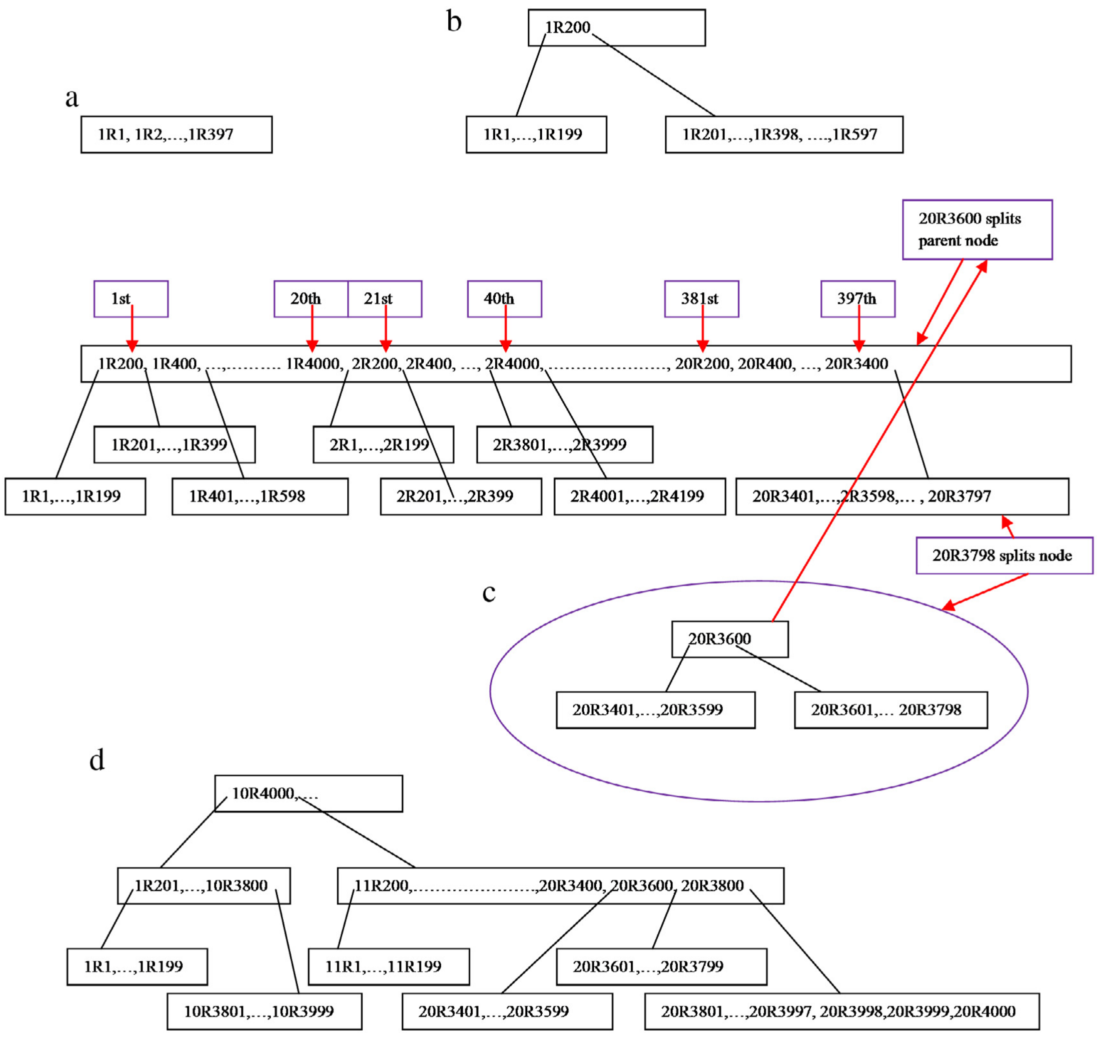

Fig. 2. Constructing a local B-Tree index in a datanode.

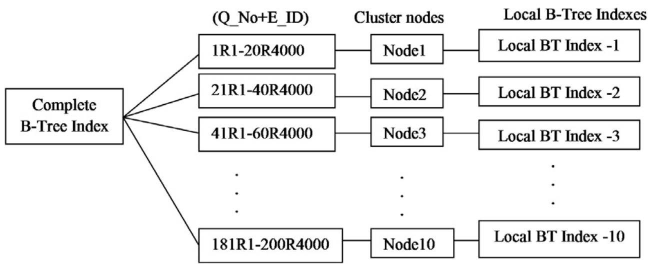

Fig. 3. Constructing a parallel B-Tree index in MapReduce (physical view).

4. Implementation

A distributed B-Tree index is created for structured data available in Comma Separated Value (CSV) format. A random sampling process partitions input dataset into '*n*' partitions and distributes among nodes in such a way that a particular range of the index on an attribute lies with a particular compute node. Where, '*n*' is obtained from division of input file size by HDFS block size. Master node that does distribution, maintains a global index, having information about the index attribute data range of a computing node, data file locations and replicas, and IP address of the computing node. A chained-MapReduce process is used to construct B-Tree index in first round of MapReduce (Job 1 using Algorithm 1), and search query is implemented in the second round (Job 2 using Algorithm 2). The chained process improves intermediate data access time and contributes towards improving efficiency of range query.

4.1. B-Tree index in MapReduce

This section describes a parallel B-Tree index building algorithm in MapReduce. The MR-BtreeMapper procedure takes NonKey_Data variable as the key field that becomes the key for the B-Tree index from the input table. The Other_Data variable represents the remaining data constituting the row tuple and a variable $i = 0$ in line 1. A map function starts with the input key as a Text and value again as a Text on line 2. The two variables "carrier" of type IntArrayWriteable and "innercarrier" of IntWritable are initialized in line 3. A sub-task in the map function reads each row of the partitioned dataset allotted to the compute node and tokens are generated for the input row tuple with a comma-delimiter (,) as the separator. Each token is temporarily stored in the String Tokenizer variable "itr" in line 4. A while loop (line 5–12) emits (NonKey_Data, ArrayWritable data) as the Key-Value pair for the reduce function until all the rows are exhausted. For each row of

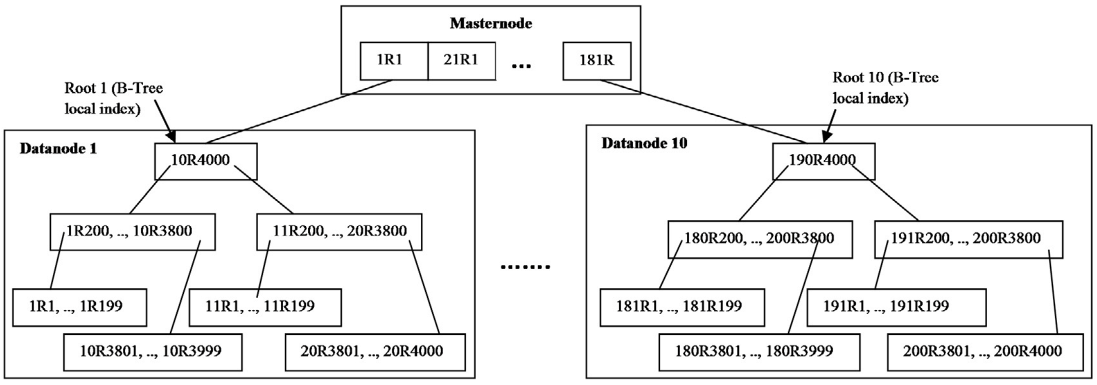

Fig. 4. Constructing a parallel B-Tree index in MapReduce (logical view).

the dataset, the first field of the row is an identifier key and the rest of the fields of the row tuple are an ArrayWritable. Line 6 checks each token as belonging to the first field and sets its value in the variable NonKey_Data. Apart from the first field, line 7 stores other fields in variable Other_Data one by one till the end of the row is reached. Line 8 runs an ArrayWritable collection and accumulates all fields in an ArrayWritable "innercarrier". Line 9 takes loop to next row in the dataset and line 10 sets ArrayWritable "carrier". Line 11 finally emits key-value pairs for reduce function.

Job1 constructs a B-Tree index
Algorithm 1. A Final B-Tree index is constructed.
Procedure: MR-BTreeMapper<LongWritable, Text, Text, ArrayWritable>
Input: CSV file
Output: (NonKey_Data, ArrayWritable data) pairs

```
Begin
Step1.   Initiate NonKey_Data, Other_Data, and i=0;
Step2.   Start map function: map(Text, Text value, Context context)
Step3.   Initiate carrier of IntArrayWriteable type  and innercarrier of IntWritable;.
Step4.   StringTokenizer itr = new StringTokenizer(value.toString(),",");
Step5.   While (itr.hasMoreTokens()) do steps 6-12
Step6.   If(i==0)then NonKey_Data.set(itr.nextToken());
Step7.   If(i>=1) then Other_Data.set(itr.nextToken());}
Step8.   innercarrier[i-1]=new IntWritable(Other_Data);
Step9.   i++;
Step10.  carrier.set(innercarrier);
Step11.  emit  (NonKey_Data,  ArrayWritable  data)  pairs  using  context.write
          (word ,   carrier)   //  Output  Text  and ArrayWritable
Step12.  i=0;
End
```

The procedure MR-BtreeReducer takes the output of the map function (key, ArrayWritable) as input and calls the Btree [42]. The first comma separated parameter, E_ID, is extracted from the value part. A combination of the key, Q_No, and E_ID, that acts as a unique key to the B-Tree, is passed as an index to addNode function (line 3–8) for all the output from the map function. The line 8 emits the root pointer for the constructed B-Tree. All such root pointers, from the B-Tree generated by all the compute nodes involved in the process, are managed and indexed by the master root node for future reference.

Procedure: MR-BTreeReducer <Text, ArrayWritable, Text, ArrayWritable>
Input: (NonKey_Data, ArrayWritable data) pairs
Output: A B-Tree index on Key_Data

```
Begin
Step1.   Start reduce function: reduce (Text key, Iterable<ArrayWritable>
          values, Context context)
Step2.   BTree theTree = new BTree();
Step3.   for (ArrayWritable val : values) do steps 4-8.
Step4.   new = val.get();
Step5.   Extract first comma separated value in variable new1
Step5.   String str = key.toString();
Step6.   Merge str and new1 in str1
Step7.   theTree.addNode(str1, new);
Step8.   emit root of sub-B-Tree with Key_Data identifier.
End
```

4.2. Range search query on B-Tree index in MapReduce

The main aim of processing a range search query on the B-Tree index in MapReduce is to minimize I/O overhead during query processing while enhancing query throughput through parallelization. A range search is carried out in parallel using Job2, the map phase compact search space by eliminating part of the B-Tree that does not include the search key. It does this by comparing the condition specified in the input which is actually a subset of the indexed dataset. The reduce phase performs exact matching of the key with node data, extracts other fields, and return result (Algorithm 2).

The input range to be searched "Data_Search" is partitioned and each partitioned set is mapped onto the B-Tree index through a B-Tree search algorithm. The data for which the mapping result is true is returned and a list of all such data items is prepared that acts as the input to a reduce function. The reduce function finally processes the data and writes the result to HDFS.

Job2: A MapReduce range search on B-Tree indexed dataset.
Algorithm 2: MR-Search
Input: Set of Key_Data values and the B-Tree index from reducer of Job1
Output: the number of Key_Data values that meet the range query using a B-tree search algorithm.

```
Begin
Step1.   Partition Data_Search values into splits.
Step2.   Implement function map and execute  a B-Tree Search for Data_Search
Step3.   For search Data_Search, if successful, return 1; otherwise, return 0;
Step4.   Output (Found, identifier of Found);
Step5.   Implement function, reduce, for input (Found, list(identifier of  Found))
Step6.   Compute number of Data_Search that meet the query.
Step7.   Set path of Input and Output.
Step8.   Return the results of a query
End
```

4.3. Chaining of MapReduce jobs

The jobs are chained together by writing multiple driver methods, one for each job [19]. The input for the first chained MapReduce job1 is args[0] that is input from the command prompt at the time of running the job. The args[0] actually represents the location where the input table in CSV is kept. The output of the job1 is stored in a variable OUTPUT_PATH that is a Temporary_output_path at some location on the HDFS. This OUTPUT_PATH serves as the input for job2 and after the job2 process is over, the output is returned at a location on the HDFS represented by args[1].

The complete chained MapReduce program that integrates job1 and job2 in sequence is as follows:

```
public class SearchOverIndexedHadoop extends Configured implements Tool {
private static final String OUTPUT_PATH = "Temporary_output_path";
public int run(String[] args) throws Exception {
<Job1>
Configuration conf = new Configuration();
Job job1 = new Job(conf, "Job1");
job1.setJarByClass(MR-BTree.class);
job1.setMapperClass(MR-BTreeMapper.class);
job1.setReducerClass(MR-BTreeReducer.class);
job1.setMapOutputKeyClass(Text.class);
job1.setMapOutputValueClass(ArrayWritable.class);
job1.setOutputKeyClass(Text.class);
job1.setOutputValueClass(Text.class);
FileInputFormat.addInputPath(job1, new Path(args[0]));
FileOutputFormat.setOutputPath(job1, new Path(OUTPUT_PATH]));
job.waitForCompletion(true);
<Job2>
Job job2 = new Job(conf, "Job2");
job2.setJarByClass(MR-BTreeSearch.class);
job2.setMapperClass(MR-BTreeSearchMapper.class);
job2.setReducerClass(MR-BTreeSearchReducer.class);
job2.setMapOutputKeyClass(Text.class);
job2.setMapOutputValueClass(ArrayWritable.class);
job2.setOutputKeyClass(Text.class);
job2.setOutputValueClass(Text.class);
FileInputFormat.addInputPath(job2, new Path(OUTPUT_PATH));
FileOutputFormat.setOutputPath(job2, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);  }
public static void main(String[] args) throws Exception {
ToolRunner.run(new Configuration(), new SearchOverIndexedHadoop(), args); } }
```

Some assumptions are made for efficient functioning of the proposed method. Random Access Memory (RAM) size is calibrated for master and slave compute nodes, as shown in Table 1, and a replication factor of two is considered for data blocks. The namespace on the master node uses primary memory for building the metadata about the data over the HDFS file system. The metadata generally consists of information about the file chunks contained in data blocks on different slave nodes over the HDFS. So its size is determined by the size of the input dataset. A shortage of RAM on Namenode crashes the job, so generally the size of RAM on Namenode is taken more than on slave nodes. It is approximately close to one-third of the total memory of the Hadoop cluster. Providing a fault tolerant system with data replication is a main characteristic of the Hadoop cluster. However, it may degrade performance, so a limit must be kept over the number of replication of data blocks. There are several reasons for it. Firstly, creating replicas over a large cluster becomes an expensive operation. Secondly, when replicas are created and put on different data blocks, then data transfer in the form of replica takes place over the network. It causes network resource consumption and puts load over the network. Thirdly, the generated replicas are put on the data blocks of slave datanodes and hence consume local disk capacity.

5. Performance evaluation

Experiments are carried out for performance evaluation of range search query on B-Tree indexed MapReduce. This section includes experimental set-up and discusses experimental results on predefined performance metrics.

5.1. Experimental design

Hadoop cluster is implemented over ten nodes attached in a star topology and interconnected in a LAN that runs at 100 Mbps, as shown in Fig. 5. All machines have dual core processors with a speed of 2.1 GHz and use Ubuntu 11.04, Java-6-openjdk, and Hadoop-0.21.0. The physical configuration of the created Hadoop cluster is presented in Table 1.

This paper uses a synthetic large dataset from a live Online Test Application (OTA) package for conducting Multiple Choice Question (MCQ) test at our institute. Out of many result files being generated by OTA, we have chosen the file "ResultFile" whose schema is shown in Table 2. The file contains data in CSV format for a number of batches. All such files are put on the HDFS. These files contain a one-to-one result-record for each question, for each examinee. E_ID field is a unique identification of each examinee. E_Name field is the name of the examinee. F_Name field holds the father's name of the examinee. The B_Code field is the department of the examinee. I_Name is the institutes' name of the examinee.

S_No field is the seat allotted to examinee during the conduct of the test. R_No field is the examination room allotted to the examinee. TID field represents a unique identification of a test for one batch of examinees. The Q-No field represents the serial number of questions in the test. Answer field records the option selected by examinee out of the available four choices. The C_Answer field is the correct predefined answer of a question.

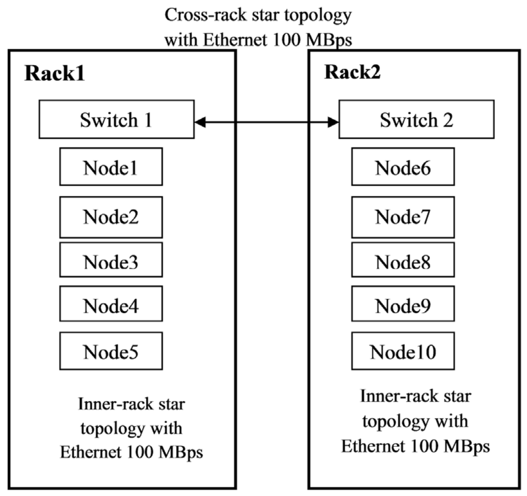

Fig. 5. Topology of the Hadoop cluster created.

Table 1
Physical configuration of the created Hadoop cluster.

Sr. No.	Host name	IP address	Primary memory (GB)
1	Node1 (Master)	172.16.1.151	2
2	Node2	172.16.1.152	1
3	Node3	172.16.1.153	1
4	Node4	172.16.1.154	1
5	Node5	172.16.1.155	1
6	Node6	172.16.1.181	1
7	Node7	172.16.1.182	1
8	Node8	172.16.1.183	1
9	Node9	172.16.1.184	1
10	Node10	172.16.1.185	1

5.2. Results and discussions

This section discusses the performance of range search queries on the default non-indexed dataset in Hadoop (NIQ), the proposed B-Tree index in MapReduce (BTQ) and the Global Index described in [26]. The performance is analyzed with varying cluster size and the amount of data searched in range query. The effect of range query coverage on a map task number and I/O data size is observed. The effect of varying HDFS block size on range query execution time, heap memory and, the amount of intermediate files and directories, is also observed. A set of five similar experiments has been conducted to find the choices marked by examinees for a particular number of questions. The input dataset has been generated with 100 objective type tests conducted for 4000 students. Each test carries 200 objective type questions with four choices and generates around 800 000 tuples. The motive has been

Table 2
The schema of MCQ-Test generated ResultFile dataset.

Fields	E_ID	E_Name	F_Name	B_Code	I_Name	S_No	R_No	TID	Q_No	Answer	C_Answer
Data type	Text	Text	Text	Text	Text	Text	Text	Text	Text	Text	Text
Size (bytes)	20	30	30	3	50	4	5	10	3	1	1

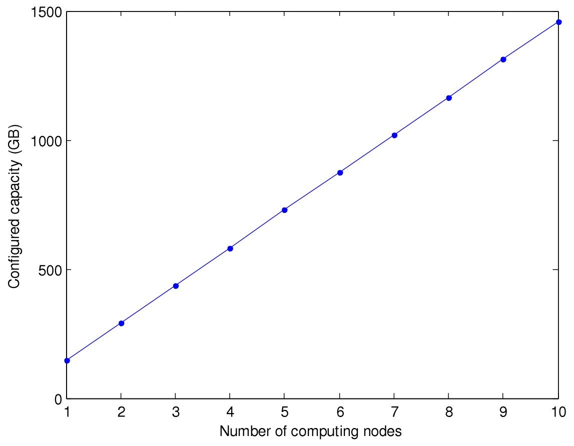

Fig. 6. HDFS configured capacity versus Hadoop cluster size.

to analyze the response of examinees for questions where most of the students opt a different choice than the correct one. The result of the execution, where this number remains considerable, helps in re-discussing the topic in more detail with students. The mean of different experiments has been taken to present calculated results in the following sub-sections.

In one such experiment, a range search query covering 5% of the dataset has been used. If there are 200 objective type questions in the input dataset, then a range search coverage of 5% means 10 specific questions are searched for the opted answers by examinees. Similarly, a range search query covering 10%, 20% and 40% have been used that cover 20, 40 and 80 questions of the input dataset, respectively.

5.2.1. *HDFS capacity of the cluster*

Each node in the cluster has 160 GB of hard disk space and approximately 145 GB is left available after operating system and other software use the memory. The disk capacity of each node is added to the cluster whenever a node becomes part of it. It has been validated through the experiment as shown in Fig. 6.

5.2.2. *The effect of cluster size on range query time*

The range search query execution time of the NIQ, BTQ and Global Index reveals a similar pattern. The time taken to distribute the partitioned data onto cluster nodes and managing a global index of the master node is almost similar in all the three approaches. The query execution time decreases as the size of cluster increases. It is due to parallelization of query. The query execution time is composed of the amount of time elapsed in map and reduce phases. The range query is decomposed into sub-queries that run in parallel, and consequently, execution time decreases. Initially, there is a small rise in execution time when second and third node is added to the cluster. It is due to increased sorting and shuffling in intermediate stages that overcomes the gain in performance due to fast computation achieved with parallelization. But later on, the performance gain achieved with parallel computation exceeds the burden of sorting and shuffling. Consequently, execution time decreases gradually with the addition of nodes in the cluster. The graph shown in Fig. 7(a) describes it for a range query that covers 5% of the dataset.

It is clear from the graph that query execution time on indexed dataset is far better than non-indexed dataset. The BTQ approach performs almost 40%–50% better than NIQ, up to a cluster size of 6, but the efficiency rises gradually to 60% for a cluster size of 10. The Global Index also performs around 20%–30% better than NIQ, up to 5 computing nodes. The performance rises gradually with increasing cluster size and reaches up to a maximum of 48% for a cluster size of 10. Further, the proposed BTQ approach performs better than the Global Index by approximately 15%–20%.

The Global Index sorts input dataset according to an index field and the whole dataset is distributed onto cluster nodes according to a lower and upper limit of the index value. The index keeps record of distributed data on the master node of the cluster. A binary search algorithm executes range search query that only accesses relevant data blocks on computing nodes and leaves irrelevant data, and thereby reduces execution time. However, it needs to build the index each time a range search query is executed. In our proposed B-Tree on MapReduce, the index is built only one time on the complete dataset and only range search queries execute on the index. The indexing of the data reduces query execution time. As, the index need not to be built for each range search query, so, the efficiency of queries is better than the Global Index approach.

However, when the index-build time and query execution time is taken together, the performances of B-Tree index become comparable to the NIQ, as shown in Fig. 7(b). The range search query on BT(L+Q) remains around 3%–5% higher than the NIQ up to 5 computing nodes. But for a cluster size of 6–7, the query time of both approaches becomes almost equal. The query time of BT(L+Q) becomes better than the NIQ by an average of approximately 10% than the NIQ. The term (L+Q) in BT(L+Q) denotes the bulk-loading time and query execution time respectively. It is evident from Fig. 7(b) that a significant amount of time is required to build the index. As the index is created on the full input dataset, so, it takes a significant time as compared to the non-indexed dataset. However, query execution becomes very fast on the indexed dataset as compared to the non-indexed dataset. One important point to note here is that the query execution time in indexed dataset becomes better with increased parallelization due to increased cluster size, while the index construction time remains almost constant. So, the (L+Q) time of indexed dataset becomes better than non-indexed dataset after five computing nodes. And it gradually becomes better and better with the increase in computing nodes (cluster size). However, once the index is built, only a small amount of time is taken by the search query. A slight increase in execution time is observed for both the indexes beyond a cluster size of six nodes.

5.2.3. *The effect of range query coverage on execution time*

The amount of data queried, range query coverage, is directly proportional to the query execution time. The execution time of NIQ, BTQ and Global Index was noticed for four range queries, covering 5%, 10%, 20% and 40% of the dataset, respectively. The comparison of three indexes for the four range query coverage is shown in Fig. 8. It was observed that the query execution time of indexed dataset is far better than the non-indexed dataset for the range query coverage of 5%, 10%, 20% and 40%. The reason for it has been explained in Section 5.2.2.

It was observed that for each approach, the query execution time rises with an increase in the range query coverage of the

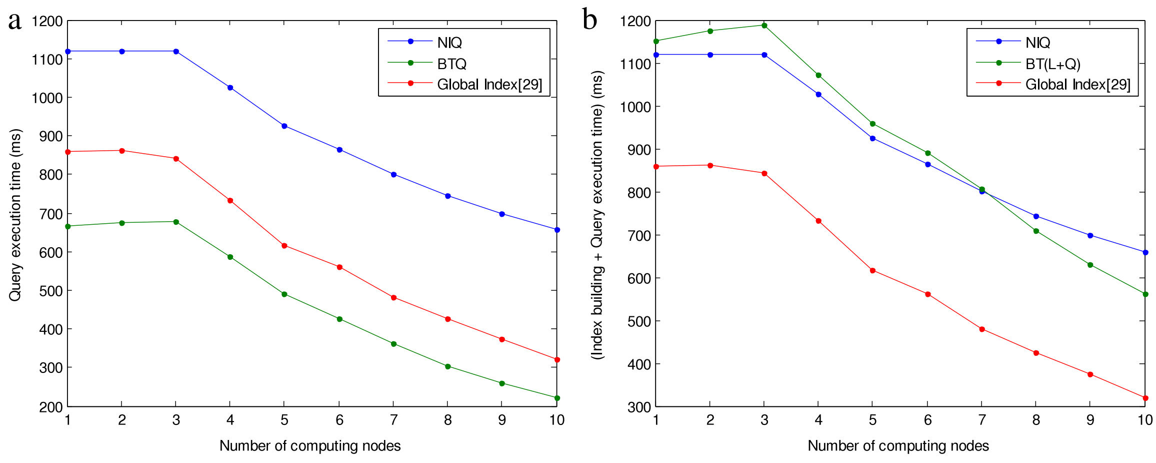

Fig. 7. (a) Range search query time on NIQ, BTQ and Global Index approaches. (b) Range search query time on NIQ, BT(L+Q) and Global Index approaches.

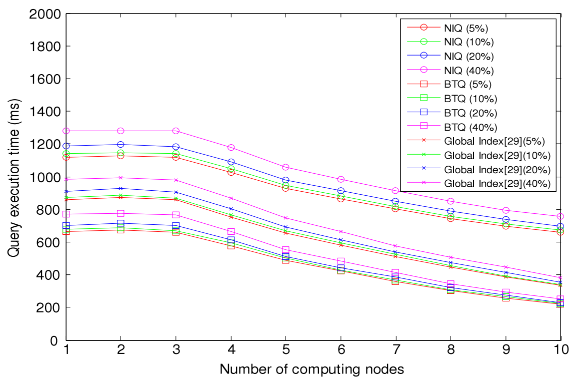

Fig. 8. Comparison of execution time of NIQ, BTQ and Global Index for range query coverage —5%, 10%, 20% and 40% of the dataset.

dataset. The query time of NIQ rises by around 2%, 6% and 14% when the range query coverage increases to 10%, 20% and 40% respectively. The query time of BTQ rises by an average of around 1%, 4% and 8% when the range query coverage increases to 10%, 20% and 40% respectively. The query time of Global Index rises by an average of around 1%, 4% and 9% when the range query coverage increases to 10%, 20% and 40% respectively. It is due to increased data accesses on increasing the range query coverage. The NIQ, Global Index and BTQ follow an order in decreasing order of their performance for range search query. The proposed B-Tree index has the best range query time among the three.

5.2.4. *The effect of range query coverage on the number of map tasks and I/O data size*

The number of map tasks is decided by the number of active mapper functions on the distributed data set. The whole data are needed to be accessed in non-indexed dataset for a range search query irrespective of the size of range coverage of the query. However, the numbers of map tasks vary, with the size of range coverage query in indexed dataset. In our dataset of size 5 GB, for a block size of 64 MB, the whole dataset is stored in 80 data blocks of the Hadoop cluster. It is due to this reason that the number of map tasks in case of NIQ is also around 80. The number of map tasks is found to be equal for both the BTQ and Global Index, as shown in Fig. 9(a). The map tasks increase with data selection which is evident from the Fig. 9(a). Similar to the number of map tasks, the number of I/O is only dependent on the input data and does not depend on the size of data selection. The number of I/O occurred during the range search query is directly proportional to the size of data selection in the query for indexed dataset. The amount of I/O increase with range coverage size from 5% to 40% is presented in Fig. 9(b). The number of I/O is slightly better in case of the proposed B-Tree index as compared to the Global Index. It is due to better indexing provided by B-Tree as compared to the binary search on the Global Index.

5.2.5. *The effect of HDFS block size on execution time*

The client machine puts data file on HDFS data blocks by configuring hdfs-site.xml configuration file of Hadoop cluster. The input data file is split through setting-up the block size. The split size determines the number of map and reduce tasks. The execution time of a job decreases with an increase in the size of the Hadoop cluster for block sizes (64, 128 and 256 MB). It can be seen from the execution time plotted for NIQ, BTQ and Global Index, and NIQ, BT(L+Q) and Global Index in Fig. 10(a) and (b), respectively. The execution time of BTQ is lowest and NIQ is highest, when only query execution time is considered. The execution time of the proposed B-Tree is comparable to NIQ when both bulk-loading and query execution time are considered, and it becomes lowest for the Global Index. The execution time of NIQ becomes poorer to BT(L+Q) after five computing nodes. The reason for it has been given in Section 5.2.2. For a particular number of nodes in the cluster, the execution time for a job is found low when block size is 128 MB. The execution time decreases when the job is switched from a cluster with 64 MB block size to 128 MB block size by an average of 6%. However, the execution time increases when the job is put on a cluster from 128 MB block size to 256 MB block size by an average of 12%. It is due to the insufficient Java heap memory available for the output mapped intermediate data that causes mapped-spill.

5.2.6. *The analysis of heap memory and intermediate data generated*

The intermediate output and metadata are stored in the buffer when task trackers in the Hadoop cluster execute map tasks. The buffer is a part of map JVM heap space and a spill occurs when a defined barrier is reached by either intermediate data or metadata. When block size increases, the mapped output size also increases as mapped output is directly proportional to the block size. The performance of the cluster increases with a rise in the block size up to a particular value of the block size, for which, the Java heap memory can accommodate mapped intermediate output records and metadata. However, performance decreases on further increasing the block size, as Java heap memory space starts spilling the intermediate mapped output records or the metadata.

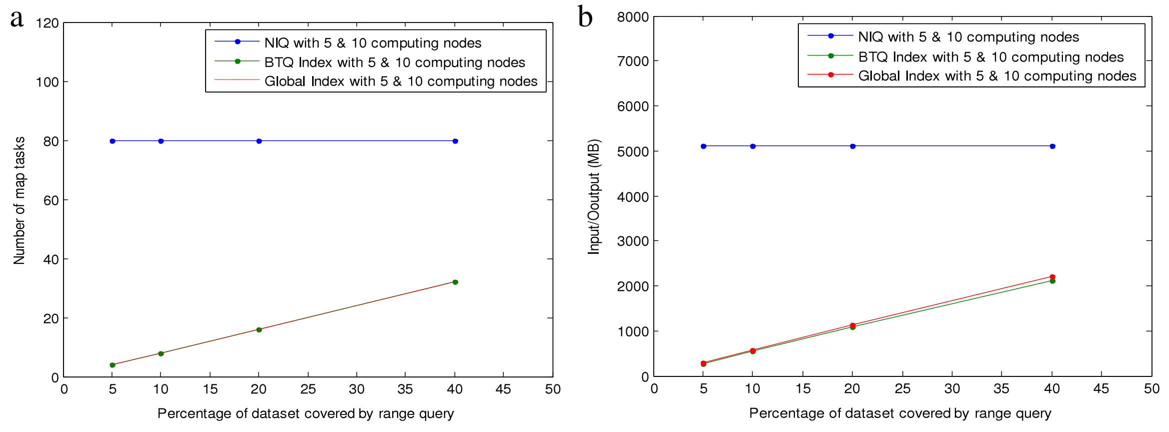

Fig. 9. (a) Number of map tasks versus percentage of dataset covered by the range query (b) I/O versus percentage of dataset covered by range query.

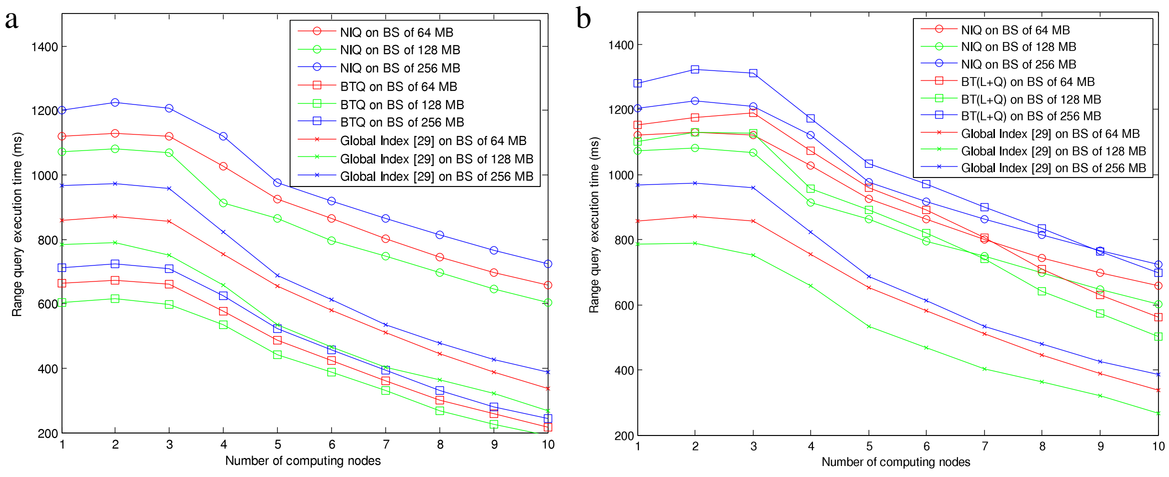

Fig. 10. The effect of block size on the execution time of range search query for (a) NIQ, BTQ and Global Index (b) NIQ, BT(L+Q) and Global Index.

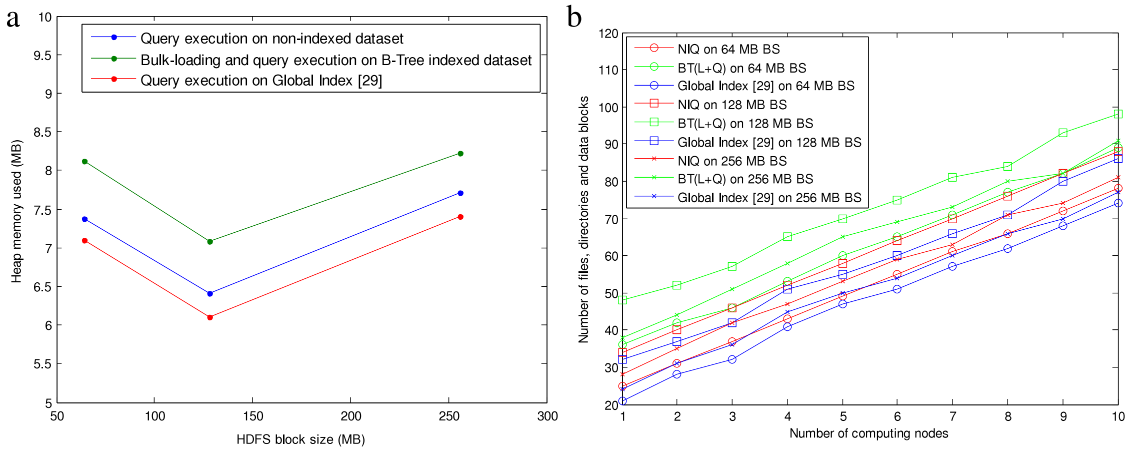

Fig. 11. (a) Block size versus Heap memory on a ten-node Hadoop cluster. (b) Number of files, directories, and blocks generated for range querying NIQ, BT(L+Q) and Global Index with variable cluster size.

Whichever threshold reaches earlier, it lowers the performance in terms of execution time as shown in Fig. 11(a) and (b).

The B-Tree index consumes more heap memory and reveals slightly irregular behavior as compared to the non-indexed dataset due to the increased complexity with indexing. More heap memory is consumed by the B-Tree indexed range search query, as a complete index is built on the input dataset, while only a selected part of the input is accessed by the Global Index approach.

6. Conclusions and future scope

The MapReduce technology has proved very effective for large scale structured, semi-structured and unstructured data, for information processing and retrieval. MapReduce has built-in features, such as high scalability, fault tolerance, and adaptation to heterogeneous clusters. When the cluster grows in size, each node contributes its memory in the HDFS and computing power, consequently, increasing the data holding capacity of the disk and task processing capacity, respectively. A cluster processes a job by decomposing it into sub tasks and executing these subtasks in parallel and consecutively reduces the query time. This paper tries to improve the scalable discovery through indexing in MapReduce. In this connection, a B-Tree index, in a chained-MapReduce process, is designed and implemented. The proposed work compares the default non-indexed data organization of Hadoop and a Global Index [26], for range search queries.

The performance of range search query on default non-indexed dataset of the Hadoop, B-Tree indexed dataset and the Global Index [26] has been analyzed on various performance evaluation parameters. The range search query time in MapReduce for B-Tree indexed dataset outperforms the binary search based Global Index and non-indexed dataset in the Hadoop. The query time is poorest in non-indexed dataset and increases with increasing the percentage of data to be searched.

It is observed that a significant amount of time is required to build indexes. Indexing is beneficial for both static and dynamic datasets. It is especially good for static dataset where index needs to be built once and data retrieval queries are run many times. However, when bulk-loading time is also considered than the total execution time of B-Tree in MapReduce becomes poorer to the Global Index, and it becomes comparable to the non-indexed dataset in Hadoop. The combined bulk-loading and query time of B-Tree indexed dataset starts becoming better than the non-indexed dataset, for a cluster size of more than four. It is due to a considerable improvement in the query time in the indexed dataset as compared to the index-build time. The number of map tasks remains constant for range search query over non-indexed dataset, and it does not change when the amount of data to be searched in query changes. However, the number of map tasks increases with a rise in the amount of data to be searched in the query. The number of map tasks remains same for a particular amount of data to be searched in non-indexed, B-Tree indexed and Global Index datasets irrespective of the size of the cluster. It depends on the input size of the data only. The number of I/O follows a similar trend for non-indexed and the two indexed approaches. However, the graph remains slightly lower for B-Tree indexed dataset. It clearly suggests a small number of I/O occurring in indexed approaches. This can be a reason for a better execution efficiency of B-Tree over the Global Index. It was observed that the data access for range queries on MapReduce improves with the two indexes due to a reduction in the number of I/O overhead.

The experimental results show the effect of HDFS block size on range search query execution time for the three approaches. The query time decreases when block size increases. A lowest time is noticed when the block size becomes 128 MB, but after that it starts rising. A similar pattern is obtained in the graph for query, with and without bulk-loading time. The execution time is lowest at a block size of 128 MB, for the B-Tree index in MapReduce, when only query execution time is considered. The B-Tree index building requires additional time, but it also drops gradually with rising cluster size. The effect of HDFS block size on JVM heap memory is observed for the three approaches. The memory consumption follows a trend similar to the query time with varying block sizes. The impact of block size is related to the JVM heap memory configuration. The query time is linked to the block size and it improves with an increase in block size until JVM heap memory does not spill the intermediate data while processing map and reduce functions. The number of intermediate files, directories and data blocks generated for three approaches with varying cluster size and HDFS block size is also analyzed. The graph reveals a direct correlation between the intermediate data generated and the query time.

While working on this paper, several areas have been identified where this work can be extended. Firstly, we wish to analyze the effect of varying number of data block replicas and index-node size on query performance. Secondly, we wish to extend this work for spatial dataset that is huge in size due to its internal characteristics.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.future.2017.03.028.

References

[1] David Gil, Il-Yeol Song, Modelling and management of big data: challenges and opportunities, Future Gener. Comput. Syst. 63 (2016) 96–99.
[2] Hadoop. in: http://Hadoop.apache.org (Accessed on 13.01.16).
[3] Jimmy Lin, Alek Kolcz, Large-scale machine learning at Twitter, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 2012, pp. 793–804.
[4] Fan Zhang, Junwei Cao, Samee U. Khan, Keqin. Li, Kai Hwang, A task-level adaptive MapReduce framework for real-time streaming data in healthcare applications, Future Gener. Comput. Syst. 43–44 (2015) 149–160.
[5] Owen O'Malley, Arun C. Murthy, Yahoo! Winning a 60 Second Dash with a Yellow Elephant. http://sortbenchmark.org/Yahoo2009.pdf (April 2009) (Accessed on 22.02.16).
[6] Eric Anderson, Joseph Tucek, Efficiency matters!, ACM SIGPOS Oper. Syst. Rev. 44 (1) (2010) 40–45.
[7] F.N. Aftari, J.D. Ullman, Optimizing joins in a MapReduce environment. in: Proceedings of the 13th International Conference on Extending Database Technology, EDBT, 2010, pp. 99–110.
[8] Dawei Jiang, Beng Chin Ooi, Lei Shi, Sai Wu, The performance of MapReduce: An in-depth study, Proc. VLDB Endow. 3 (1–2) (2010) 472–483.
[9] Rajashekhar M. Arasanal, Daanish U. Rumani, Improving MapReduce performance through complexity and performance based data placement in heterogeneous hadoop clusters, in: Distributed Computing and Internet Technology, in: Lecture Notes in Computer Science, vol. 7753, 2013, pp. 115–125.
[10] Burhan Ul Islam Khan, Rashidah F. Olanrewaju, Hunain Altaf, Asadullah Shah, Critical insight for MapReduce optimization in Hadoop, Int. J. Comput. Sci. Control Eng. 2 (1) (2014) 1–7.
[11] Matei Zaharia, Andy Konwinski, Anthony D. Joseph Randy Katz, Ion Stoica, Improving MapReduce performance in heterogeneous environments, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association Berkeley, CA, USA, 2008, pp. 29–42.
[12] Jeffrey Dean, Sanjay Ghemawat, MapReduce: Simplifieddataprocessing on large clusters, in: Magazine Communications of the ACM - 50th Anniversary Issue: 1958–2008, Vol. 51, no. 1, 2008, pp. 107–113.
[13] Ian H. Witten, Alistair Moffat, Timothy C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann, ISBN: 1558605703, 1999.
[14] Steffen Heinz, Justin Zobel, Efficient single-pass index construction for text databases, JASIST 54 (8) (2003) 713–729.
[15] Anthony Tomasic, Hector Garcia-Molina, Performance of inverted indices in shared-nothing distributed text document information retrieval systems. in: Proceedings of PDIS, 1993, pp. 8–17.
[16] Mike Cafarella, Doug Cutting, Building nutch: Open source search, ACM Queue 2 (2) (2004) 54–61.
[17] Richard M.C. McCreadie, Craig Macdonald, Ladh Ounis, MapReduce indexing strategies: Studying scalability and efficiency, Inf. Process. Manage. 48 (5) (2012) 873–888. 2010, Elsevier.
[18] Christos Doulkeridis, Kjetil Norvag, A survey of large-scale analytical query processing in MapReduce, VLDB J. (2013) 1–27.
[19] Daewoo Lee, Jin-Soo Kim, Seungryoul Maeng, Large-scale incremental processing with MapReduce, Future Gener. Comput. Syst. 36 (2014) 66–79.
[20] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, Peter Camble, Sparse indexing: large scale, inline deduplication using sampling and locality, in: Proceedings of USENIX Conference – File and Storage Technologies, FAST, February 2009, pp. 111–123.
[21] Jorge Veiga, Roberto R. Exposito, Guillermo L. Toboada, Juan Tourino, Flame-MR: An event-driven architecture for MapReduce applications, Future Gener. Comput. Syst. 65 (2016) 46–56.
[22] Jimmy Lin, Chris Dyer, Data-intensive text processing with MapReduce, in: Book-Data Intensive Text Processing with MapReduce, Morgan and Claypool Publishers, 2010.
[23] Avinash Lakshman, Prashant Malik, Cassandra: structured storage system on a p2p network, in: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, Canada, 2009, pp. 5–5.
[24] M.K. Aguilera, W. Golab, M.A. Shah, A practical scalable distributed b-tree, PVLDB 1 (1) (2008) 598–609.
[25] Sai Wu, Dawei Jiang, Beng Chin Ooi, Kun-Lung Wu, Efficient b-tree based indexing for cloud data processing, Proc. VLDB Endow. 3 (1) (2010) 1207–1218.
[26] Hui Zhao, Shuqiang Yang, Zhikum Chen, Songcang Jin, Hong Yin, Long Li, MapReduce model-based optimization of range queries, in: Proceeings of the 9th International Conference on Fuzzy Systems and Knowledge Discovery, IEEE, 2012, pp. 2487–2492.

[27] Haung Bin, Peng Yuxing, An efficient distributed B-Tree index method in cloud computing, Open Cybernet. Syst. J. 8 (2014) 302–308.
[28] C. Tang, J. Gao, T. Wang, D. Yang, Distributed B+tree index system and building method. C.N. Patent 101576915, Nov. 11, 2009.
[29] Gang Chen, Hoang Tam Vo, Sai Wu, Beng Chin Ooi, M. Tamer Ozsu, A framework for supporting DBMS-like indexes in the cloud, Proc. VLDB Endow. 4 (11) (2011).
[30] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, ScalaGiST: Scalable generalized search trees for MapReduce systems, Proc. VLDB Endow. 7 (14) (2014).
[31] Marcos Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, Christos Karamanolis, Sinfonia: a new paradigm for building scalable distributed systems. in: Proceedings SOSP'07, Oct. 2007, pp. 159–174.
[32] H.V. Jagdish, B.C. Ooi, Q.H. Vu, BATON: A balanced tree structure for peer-to-peer networks, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment, 2005, pp. 661–672.
[33] David A. Patterson, Technical perspective: the data center is the computer, Commun. ACM 51 (1) (2008) 105–105.
[34] O'Reilly Media, Inc.. Hadoop: The Definitive Guide, 4th Edition by Tom White.
[35] Feng Li, Beng Chin Ooi, M. Tamer Ozsu, Sai Wu, Distributed data management using MapReduce, ACM Comput. Surv. 46 (3) (2014) Article No. 31.
[36] Yang Liu, Maozhen Li, Mukhtaj Khan, Man Qi, A MapReduce based distributed LSI for scalable information retrieval, Comput. Inform. 33 (2014) 259–280.
[37] Hung-Chih. Yang, Ali Dasdan, Ruey-Lung Hsiao, D. Stott Parker, MapReduce merge: simplified relational data processing on large clusters, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, 2007, pp. 1029–1040.
[38] Tim Kaldewey, Eugene J. Shekitaand Sandeep Tata, Clydesdale: Structured Data Processing on MapReduce, EDBT 2012, March 26–30, Berlin, Germany, 2012.
[39] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, in: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, 2007.
[40] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: A distributed storage system for structured data, in: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, OSDI06, 2006, pp. 205–218.
[41] S. Ghemawat, H. Gobioff, S.T. Leung, The Google file system, in: Proceedings of 19th ACM Symposium on Opreating Systems Principles, SOSP, 2003, pp. 29–43.
[42] Data Structures & Algorithms in Java https://books.google.co.in/books?isbn=8131718123, Lafore – 2003.

Hari Singh received his Bachelor of Engineering (with honors) and Master of Technology in Computer Science & Engineering. He has been pursuing Ph.D. in Computer Science from Thapar University, Patiala, India. His areas of interest are Distributed Computing, Cloud Computing, Programming and logic development. Hari Singh is an Associate Professor in the Department of Computer Science & Engineering, N.C. College of Engineering, Israna, Panipat (Haryana). He has 14 years of teaching experience in Computer Science & Engineering. He is a member of IEEE, ISTE, and CSI.

Seema Bawa, holds M.Tech. (Computer Science) degree form IIT Khargpur and Ph.D. from Thapar Institute of Engineering & Technology, Patiala. She is currently Professor, Computer Science and Engineering and Dean (Student Affairs) at Thapar University, Patiala since September 2010. Her areas of research interests include Parallel, Distributed Grid and Cloud Computing, VLSI Testing, Energy aware computing and Cultural Computing.

Dr. Bawa has rich teaching, research and industry experience. She has worked as Software Engineer, Project Leader and Project Manager, in software industry for more than five years before joining Thapar University. She has been Coordinator of two national level research & development projects sponsored by Ministry of Information and Communication Technology. She is the author/co-author 111 research publications in technical journals and conferences of international repute. She has served as Advisor/Track chair for various national and international conferences. She has supervised eight Ph.D. and forty four M.E theses so far.

Prof. Bawa is an active member of IEEE, ACM, Computer Society of India, and VLSI Society of India. She has been rendering her services across the globe as an editor and reviewer of various reputed journals of these societies.