2010 Fifth IEEE International Conference on Networking, Architecture, and Storage

Multi-dimensional Index on Hadoop Distributed File System

Haojun Liao*†, Jizhong Han*, Jinyun Fang*
*Institute of Computing Technology, Chinese Academy of Sciences
†Graduate University of Chinese Academy of Sciences
Beijing, China
{liaohaojun, hjz, fangjy}@ict.ac.cn

***Abstract*—In this paper, we present an approach to construct a built-in block-based hierarchical index structures, like R-tree, to organize data sets in one, two, or higher dimensional space and improve the query performance towards the common query types (e.g., point query, range query) on Hadoop distributed file system (HDFS). The query response time for data sets that are stored in HDFS can be significantly reduced by avoiding exhaustive search on the corresponding data sets in the presence of index structures. The basic idea is to adopt the conventional hierarchical structure to HDFS, and several issues, including index organization, index node size, buffer management, and data transfer protocol, are considered to reduce the query response time and data transfer overhead through network. Experimental evaluation demonstrates that the built-in index structure can efficiently improve query performance, and serve as cornerstones for structured or semi-structured data management.**

***Keywords*-Hadoop; HDFS; Multi-dimensional index; Query processing**

I. Introduction

The need to store and manipulate large voluminous multi-dimensional data sets has emerged in many different application domains, such as geographic information system (GIS), computer vision, and CAD/CAM. Data in these applications are usually formulated as geometries that represent objects in two, three, or high dimensional spaces. These applications pose stringent requirements with respect to the storage and query operations that need to be supported. Given the complexity in both data and query types, multiple devices can be utilized in data storages and I/O sub-systems to reduce the query response time and increase throughputs.

Distributed file systems (DFS), like Google file system (GFS) [1], Hadoop distributed file system (HDFS) [2], and Amazon Simple Storage System (S3) [3], provide high performance for a large number of clients in dealing with data intensive applications. These kind of systems are built relying on a large number of low-cost commodity hardware, and have only critical features of the file system and database management system [4], for example, concurrent access, snapshot, and records appending. These essential properties are sufficient to meet the requirements of data storage in a static environment, as in census and cartographic databases, in which the modification operations are rare. In addition, MapReduce [5] distributed computing paradigm provides scalable and high performance computational power that enables MapReduce, in conjunction with HDFS, to serve as basic infrastructure for the efficient query processing of spatial data.

Spatial access methods based on hash are either inefficient or incapable to address complex spatial queries, such as range query, nearest neighbor query, and distance join query. Hierarchical index structures, like R-tree [6] and its variants [7],[8],[9],[10],[11],[12], partition the space by grouping objects in a hierarchical manner and allow for a direct access to subsets. Many efficient query processing algorithms are proposed to handle spatial query based on R-tree-like index structure due to its simplicity and effectiveness.

In this paper, we investigate the benefits that can be obtained from built-in hierarchical index structure of HDFS in dealing with query processing in static environment. HDFS is an open-source implementation of DFS that follows the principles of GFS, and provides a similar collection of functionalities of GFS. Instead of focusing on declustering and replication issues in building distributed multi-dimensional access methods from scratch in a distributed environment, we simplify our work by adopting conventional R-tree in the context of HDFS, meaning that a hierarchical index being invisible to users is constructed for each data set in HDFS to support complex query types. We also investigate several factors to bridge the gap between the design principles of HDFS and the requirements of the block-based index structure and, hence, improve the query performance and query response time.

The underlying file systems, on which the built-in indices are dependent, manage to solve the common issues in distributed system, such as data partition, data replication, and intercommunication among computing nodes, with the result that the index structure can forward the problem of scalability and fault-tolerance to the underlying DFS. Similar to all DFS, HDFS offers an effective way of manipulating and maintenance index as in the single processor environment. Therefore, no significant modifications of the original index structure are required to enable its appropriate function.

Apart from the advantages mentioned above, several other benefits can also be observed by constructing the tree-like structure in HDFS. Since there are no significant differences

978-0-7695-4134-1/10 $26.00 ©2010 IEEE.

978-0-7695-4134-1/10 $26.00 © 2010 IEEE
DOI 10.1109/NAS.2010.44

IEEE computer society

between built-in index for HDFS and its conventional counterpart in database system, several optimization approaches [11] for conventional index to improve the query performance are usually applicable. Moreover, query processing techniques, especially for high selectivity query, can directly apply to this structure without extra modification. Examples of such query types are point query and range query. Despite the fact that complex query(e.g., multi-way distance join [13]) can be answered in such a way as in single processor environment, MapReduce can still be applied in order to exploit the computational power of multiple processors in a cluster towards query efficiency.

Notice that our approach to build hierarchical index can also be applied to other distributed file systems with similar features.

The rest of this paper is organized as follows. We provide motivation in Section II, and the basic approaches and framework of hierarchical index structure in Section III. We then explain the detailed implementation issues in Section IV, followed by preliminary experimental evaluation in Section V. Related work is explained in Section VI. Finally, Section VII concludes this paper.

II. Motivation

A. File access pattern

Loading index nodes into memory during tree traversal requires fetching data from underlying storage system because index structure might be too large to fit in main memory. For example, an R-tree index having fifty million ($50M$) polygons in 2-d space has a total size around 1GB, and there is usually more than one data set in a data repository in which one index is needed for each data set.

Efficient random reads with specified size are required to load index node from disk when page faults occur during tree traversal. Although streaming reads in HDFS are efficient in a sequential manner, we cannot directly adopt the data transfer policy that has been optimized for streaming reads for random reads operations.

Data stream is packed in the form of packet, termed data packet, to get through the network in HDFS. The size of data packet should be exactly equivalent to the index node size; otherwise, the efforts for data transfer will not pay off, since unnecessary transferred data for individual random read will be discarded. However, data packet of $64k$ is not appropriate for individual random reads due to its large size. In addition, random reads will consume many efforts for the initialization of new connections to data nodes, when seeking to a position of more than a specific size ($128k$ by default) away from the current position.

See Figure 1 for example, we perform random and sequential read operation on a HDFS respectively, in which data packet is set $64k$, and $4k$ data block are requested from HDFS each time. Evidently, performance of the random

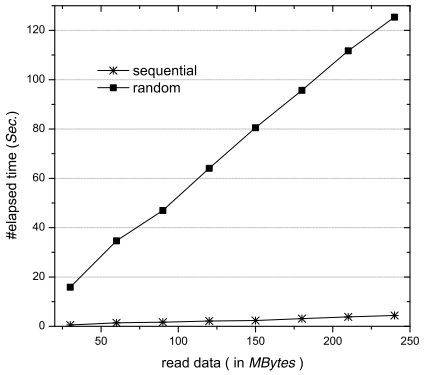

Figure 1. Comparison between sequential read and random read in HDFS

reads is about 30 times slower than that of the sequential reads.

Therefore, an efficient data transfer protocol should be adopted to satisfy access pattern for the disk-based index structure.

B. Why not on key-value store?

Key-value stores are designed to provide high performance reads and writes operation. Each key corresponds to one (or more) value(s) and serves as identification in both data organization and query processing. Key-value storage systems based on consistent hash [14] are not suited for query types, like range query and exhaustive search.

Kanth et. al. [15] and Berchtold et. al. [16] proposed methods to map tree-like indices to a flat table in order to fit in database management system, in which each tuple of a table represents an object of leaf node or sub-tree entry of internal node (directory). More than one attribute for each tuple is required to maintain the tree-like indices, and all tuples are internally organized by built-in indices in database system, for example, a B-tree. The approaches of this kind cannot be directly adopted to keep indices in key-value storage system due to the limitation of the available attributes in each key-value tuple for the relational schema. The other problem with this approach is that in tree traversal, query algorithm will incur remarkable overhead for obtaining sub-tree entries. Given a hierarchical index structure of three levels, where the fan-out is 1000, tree traversal for answering a point query has to issue around 3000 read operations(1000 for each level), or network round trips, which pose significant burden to both CPU and network. Finally, the proximity of objects in index system cannot be retained in key-value storage system to facilitate the query processing, especially for high selectivity query. We tend to think of tree-like indices, which are not appropriate to be kept in key-value storage system, as lower-level corner stones for data organization.

III. Design overview

A. HDFS

As an open-source implementation that follows the design principles of GFS, HDFS serves as an essential repository component in Hadoop platform and is designed to manage large quantities of files of several hundred megabytes or multi-gigabytes with efficient large streaming I/O operations.

A cluster has one master server, which is responsible for the management of the file system namespace and the file access requests from clients, and several data nodes, whose number ranges from several to thousands in a cluster. Data nodes are responsible for the supporting of read and write request from clients. Once a request arrives, a TCP connection has to be established between the data node and the client for data manipulation. Usually, there are more than one data nodes in a cluster having the required data chunk because of data replications. Which data node the client should connect to is determined by the rack-aware algorithm, according to latency, network bandwidth, and distance to the reader. After a specific data node has been chosen, there is only one TCP connection can be realized internally from the client to the chosen one.

In order to obtain the best utilization of available network bandwidth, data transfer from data nodes to client is done in the form of data packet of $64k$ by default, which means at least $64k$ data would be sent to client as a whole each time.

B. Index outline

We start by introducing R-tree structure and then highlight the main components of index structure applied to HDFS. R-tree, a disk-based hierarchical structure based on B-tree, is one of the most common used multi-dimensional index in low or middle dimensional spaces. Each R-tree node is implemented as a disk page. Although the actual geometry might be of any shape, all objects are approximated by minimum bounding rectangles (MBRs) to support efficient indexing. Therefore, each object accounts for a slot of fixed constant size in R-tree leaf node. In addition, each node in R-tree corresponds to one MBR that encloses its children or objects, if it is a leaf node.

The R-tree index is presented and stored as a file in HDFS but invisible to users. The replication, as well as robustness, of the index is inherently dependent on the underlying file system. We adopt the range partition policy to distribute index over a number of servers by exploiting partition mechanism for regular files provided by HDFS, where leaf nodes closer in space are stored in the same data chunk to preserve the proximity.

Buffer pool is utilized to balance the load and memory. On one hand,the internal nodes of each index account for a fraction of spaces compared with the leaf nodes because of the large fan-out. For example, given an full R-tree having one hundred million ($100M$) objects, when the fan-out is 1000, there are around 100,000 leaf nodes at the lowest level and no more than 101 internal nodes of $20k$. On the other hand, internal nodes are suited to be hold in buffer because it might be bottleneck with high probability in tree traversal. Internal nodes are loaded on demand if they are retrieved but absent in buffer, and organized in a hash table based on the associated node id. In addition to the buffer pages for internal nodes, several buffer pages are allocated to accommodate leaf nodes using LRU policy.

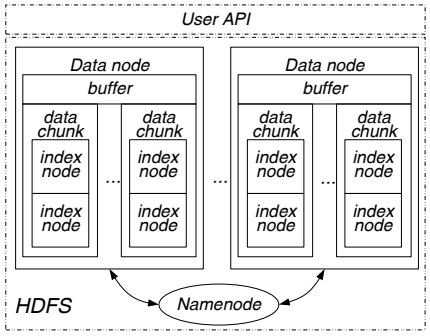

Figure 2. The structure of index framework

Figure 2 shows how the R-tree is organized over several servers (data nodes in HDFS). Note that there are several differences from the conventional R-tree-like index in single processor environment, with respect the index node distribution, node size, and index organization. Index nodes that are close in spatial domain are carefully organized to reside in the same data chunk in each server, since the random access beyond data chunk is rather costly operations. The value of node size is determined in such way that both I/O characteristic of HDFS and CPU costs in answering queries are taken into consideration. Indices are organized to facilitate buffer for internal nodes and to avoid that one node is partitioned into two data chunks.

C. Query processing

R-tree is shallow due to its large fan-out of node. This property helps to handle query efficiently, since tree traversal from root to leaf nodes is required to obtain the objects that satisfy the requirement in answering each query.

Only a small portion of index nodes are involved in handling high selectivity query types, i.e., point query, range query, and nearest neighbor query. Therefore, these sorts of query types can be finished by efficient tree traversal in the presence of index as in conventional single processor environment. Otherwise, exhaustive search over the original data set is performed by using MapReduce computing paradigm. However, for query involving large portion of index (or data set),join query, for example, MapReduce can also be applied instead of conventional tree traversal approach.

IV. Details of implementation

A. Buffer management

We use the in-memory buffer to solve the load and memory balancing problem, since HDFS does not provide built-in buffer strategy. Different strategies are applied to internal nodes and leaf nodes of index, taking into consideration their different access patterns in query processing and the different total size. Due to the property of tree-like structure that internal node accounts for small fraction of total space as we have mentioned above, all the internal nodes are kept in buffer once loaded, and the later access for the internal node would not incur disk access. The other reason why the internal nodes are kept in buffer is because every tree traversal has to start from root node, and the access of internal nodes might potentially become a bottleneck in the context of high concurrent access. In contrast, only a certain number of buffer pages are allocated for leaf nodes using LRU policy, since leaf nodes account for a majority of the index structure and the main memory is not sufficient to accommodate all leaf nodes. The buffer pages that can be allocated for leaf nodes are restricted by the available main memory. Intuitively, more buffer for leaf nodes will decrease the disk access and reduce the query response time as demonstrated in our experiments.

B. Node size

Many factors are involved in the determination of index node size, such as transfer overhead, I/O costs, and CPU time.

Cache conscious access methods, like CR-tree [17], CSB+-tree [18], suggest to keep relative small node size in order to improve query performance by taking advantage of L2 cache in the context of main memory database. Park and Lee [19] proposed a variant of cache-optimized access methods, termed TR-tree. TR-tree is to optimize disk performance of CR-tree by building in-page index nodes, in which several index nodes of small size are constructed within one disk page, and are loaded into memory by a single disk read when retrieved. Small node size yet large fan-out is the common characteristic shared by both CR-tree and TR-tree.

Despite the fact that HDFS is well tuned for streaming reads, I/O costs for loading data from HDFS are more costly than from local storage device. Performance of random reads is even less effective than that of the streaming reads as illustrated in Figure 1, and the gap between CPU and I/O performance continue grows. In this case, we trade-off the CPU effort by filtering more data objects for the reduction of data transfer time, meaning that large index node will be used in order to reduce the data transfer costs, resulting in larger fan-out and lower-and-wider shape of index. Large node requires more data to be processed in query and less overhead for data transfer because of the inherent data transfer protocol of HDFS optimized for stream I/Os.

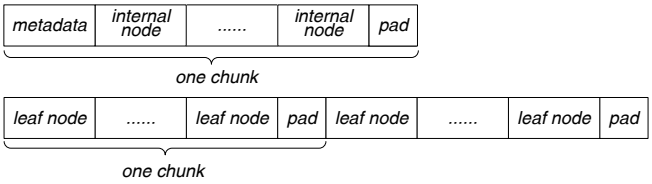

Figure 3. Disk page structure of R-tree

The index node size will be equivalent to the minimum size of underlying data packet in data transfer in order to reduce I/O overhead. Unlike in the sequence reads in which data that has been transferred to client yet not required would be consumed by following needs, random reads simply discard the rest data if not necessary. The costs for loading data from disk and transfer through network to client will not pay off, and may become a bottleneck in dealing with high concurrent access. We prefer index node size smaller than $64k$, the default size of data packet, taking into consideration both the CPU cost and network bandwidth usage and the most efficient node size will be investigated in experiments.

C. Index structure

Figure 3 depicts how the tree-like structures are organized in data chunk of index file. Metadata segment is responsible for keeping the meta-information of the associated index, including index node size, the number of leaf or internal nodes, height of index, total size, the root node id, etc. We designate $1k$ disk space for accommodating the meta-information of current index. Internal nodes, as well as metadata, need to fit in the first data chunk. As we have stated previously, internal nodes accounting for fraction of total size of index can fairly locate in the first data chunk given a reasonable size of the data chunk in HDFS. At the end of internal nodes, the rest space of the first data chunk of the index file is left blank for the extension of internal nodes.

Leaf nodes that keep the data objects information are grouped in several data chunks, according to the proximity of leaf nodes. More precisely, we employ packing algorithms (e.g., STR [20], Hilbert packing [21]) to build spatial index and guarantee the proximity of leaf nodes. If left space in a data chunk is insufficient for a leaf node, the space is padding with blank, termed pad segment in Figure 3. We align the next leaf node to the start position of the next data chunk. The alignment is used to avoid the situation that a leaf node is distributed in more than one data chunk, and hence, more than one data node. The rationale behind is that we should constrain the read requests within each data node to preserve the locality to improve I/O performance and query processing. Evidently, the blank space in each data chunk will be smaller than an index node, and the total size of blank space in each index structure is negligible, according to our experiments.

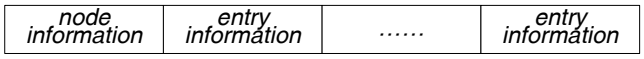

Figure 4. The node structure of R-tree

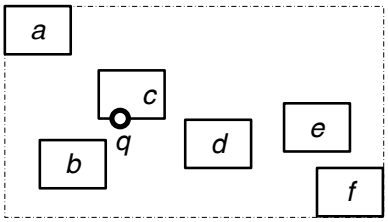

Figure 5. Example of point query

Index node contains the detailed information about sub-tree, including identification, MBR, level information and the number of entries. Figure 4 depicts the index node structure. In internal node, entry information includes the sub-tree identification and the MBR, whereas in leaf node, entry information includes the MBR information and pointer to corresponding data object.

D. Query optimization issues

Cache-conscious indices use small node size, in conjunction with MBR compression technique, to make index node fit in L2 cache to avoid fetching data block from main memory. MBR compression can increase the fan-out by more than 150% [17]. This technique can seamless integrate in this framework, and we employ a different approach to improve the object filter procedure, thereby reducing the CPU cost. Once the entries of each index nodes is sorted according to some spatial criterion (e.g., ascending x-coordinate value of lower-left corner), the order can be well preserved in a static environment. In this case, the point and range query processing within each node cost $O(n/2+w)$ time when dealing with geometry with extent, instead of sequential scanning running in time $O(n)$, where n is the number of entries in each node and w is the cardinality of the object set that satisfy the requirement. Particularly, the running time for each index node can be further reduced to $O(\log n+w)$ when dealing with point data sets in range query or point query. Spatial join [22] is also benefit from ordered data objects because the sort operation needed in plan-sweeping algorithm is avoided. Notice that procedure of ordering data objects does not incur extra overhead, since this process can be applied in STR packing algorithm used to construct static R-tree.

In Figure 5, an example of point query is illustrated, where objects are ordered by ascending x-coordinate of lower-left corner and point q is the query point. The determination of the position, from which we start to filter the objects, is implemented as binary search based on the sorted objects. Query point q locates between lower-left corner of object c and d, and therefore only a, b, and c need to be examined in order to get objects that satisfy the point query.

E. Data transfer model

When the client requests to access files in HDFS, name node sends metadata of the corresponding file to client, including file length, replica number, data chunk number, and the data nodes that hold data chunks of the file. Among all the data nodes that hold the required data chunk at local storage device, a data node is chosen based on rack-aware algorithm to establish a dedicated TCP connection. HDFS is tuned for sequential I/O reads. Data node pushes data to client in the form of data packet of $64k$ by default to obtain the best performance for sequential reads when the client performs read routine.

We implement a new data transfer protocol to facilitate the random reads for block-based index structure. When dealing with the random reads with constant size of fragment, the size of the required fragment is set to be equivalent to the size of data packet. Hence, the requirement of loading one index node can be finished by the transfer of one data packet. After that, server (data node) will be blocked until the client retrieves for a new index node. Notice that the size of required fragment is invariant, since node size is constant for a given index. Client will issue a new offset to the blocked data node and, then, data node performs a seek operation if necessary, followed by the transfer of a single data packet. If the required position does not locate within the range of the current data node, current connection will be closed, and client will initiate a new connection to data nodes holding the required data. Even in consequential reads, the offset value of position is needed by data node but does not invoke seek operation in data nodes.

In sequential reads, the original *push* mode is a little superior to our approach because data node is not blocked for further instructions and no interactive information is involved. However, our approach outperforms the push model in dealing with the random read, especially for random reads restricted in one data node of a cluster. The reason behind is that in localized random reads, TCP connection is initialized only once, and the efforts for repetitive connection to the identical data node are not required in our data transfer procedure. In addition, trivial overhead is involved since no network bandwidth is wasted.

V. Experimental evaluation

A. Experimental setup

In the sequel, a performance evaluation of the built-in R-tree in HDFS is illustrated. The main objective is to investigate the following issues:

1) the performance of new data transfer protocol for both sequential and random reads,
2) the quality of the produced index, by inspecting its performance against queries, and

3) the appropriate node size of the index structure.

We do not undertake the query performance comparison between the index structure and the exhaustive search methods, since indices can significantly outperform the exhaustive search on whole datasets in terms of query response time. The following real-world datasets are used for the experimentation: CAR contains 2,249,727 road segments of California extracted from Tiger/Line datasets; (b) HYD contains 40,995,718 line segments representing rivers of China and (c) TLK contains up to 157,425,887 points extracted from the elevation data of China. All indices for these three data sets are built by using STR packing algorithm.

Experiments were conducted over a cluster of nine nodes of Dell Power Edge SC430. Each node has an Intel Pentium 4 CPU of 2.8GHz, 1GB or 2GB memory and 80GB or 160GB SATA disk. The operating system is Red Hat AS4.4 with kernel 2.6.23. Hadoop version is 0.19.2 and java version is 1.6.0.

B. Data transfer overhead

To have an idea of how the data transfer protocol incurs overhead for random read, we compare the transferred data with the required data by varying size and counts of read operations on HYD data set. Figure 6 shows the comparison of transferred data and the data that has been required. In Figure 6(a), the data packet is set 16*k* and read count is specified to 12,800 counts. As we can see, when the size of each read operation is the power of the size of data packet, the total amount of data transferred to client from data nodes are around three times more than the required, and the gap is decreasing with the increase of read block. In Figure 6(b), we measure the transferred data by varying data block size for each read, in which the *x*-axis represents the size of total fetched data, and *y*-axis represents the transferred data by HDFS. By varying the data block size, the overhead for data transfer varies significantly and the best result reaches when the data chunk is 64*k*, where data packets have been fully consumed.

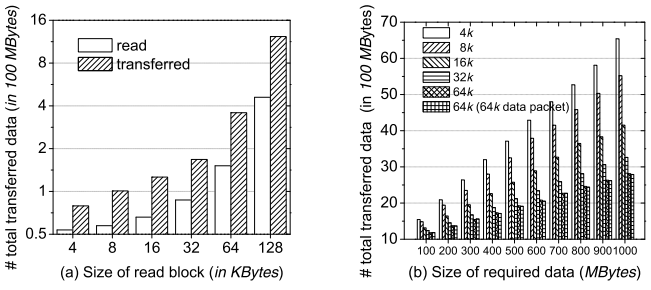

Figure 6. The comparison of the transferred data and the required (a) fixed data reads counts, (b) specified size of fetched data

C. Performance of the new transfer protocol

Figure 7 shows performance of the new data transfer protocol by varying the data packet size, and the read operations are categorized in three types: (1) sequential reads; (2) localized random reads, in which random reads are constrained in single data chunk; (3) random reads that uniformly distribute across data nodes of cluster. This result is based on TLK data set, and the only difference between localized random reads and random reads is that localized random reads are constrained to single data node, whereas the random reads uniformly access a file. The data packet during transfer ranges from 8*k* to 64*k*, and the size of data block that we read is exactly equivalent to data packet each time. Hence, for a specified amount of data, less read operations are involved as the size of data packet increases. Since no waste of data is involved, we only report the performance of transfer protocol in terms of elapsed time. As we have expected, this new protocol offers the best performance for all access patterns when the data packet is set 64*k* because less interaction between client and data nodes is involved during transfer.

D. Effects of index node size

In order to investigate the effect of the size of the index nodes, we perform range query, point query, and *k*-nearest-neighbor query on CAR data set and report the results in Figure 8. All the query processing is performed in such a manner as to in single processor environment, and the *k*-nearest-neighbor query is executed by using algorithm proposed in [23]. The node size of index ranges from 8*k* to 64*k*, and the size of data packet equals to the size of index nodes. Evidently, smaller index node size demonstrates better performance in all three kinds of query types because small index node enables the finer granular partition of data sets, and hence tree traversal can focus on the objects that satisfy the queries. Note that the gap between smaller size of node and larger ones decreases as the query window increases in range query (Figure 8(a)), because the effects of finer partition degenerates when the query involves more volume of spaces.

Figure 9 shows the response time of range query and point query on CAR data set. Despite the fact that fewer data objects are involved for finer granular partition of data sets, the response time is not necessarily smaller because of the inefficient data transfer for smaller data packet. The worst performance of range query is when the index node size is set 16*k*, and the performance degenerates rapidly with the increase of query window. The response time for point query is longest when index node size is set 8*k* because the index has more level and more node visits are involved with smaller node size. The performance difference between node size of 16*k* and 32*k* is marginal in point query.

In Figure 10, we vary the size of data packet while keeping the index node size a constant value. We report the average time for 100 range queries on CAR data set. The range query performs best when the data packet is 64*k*.

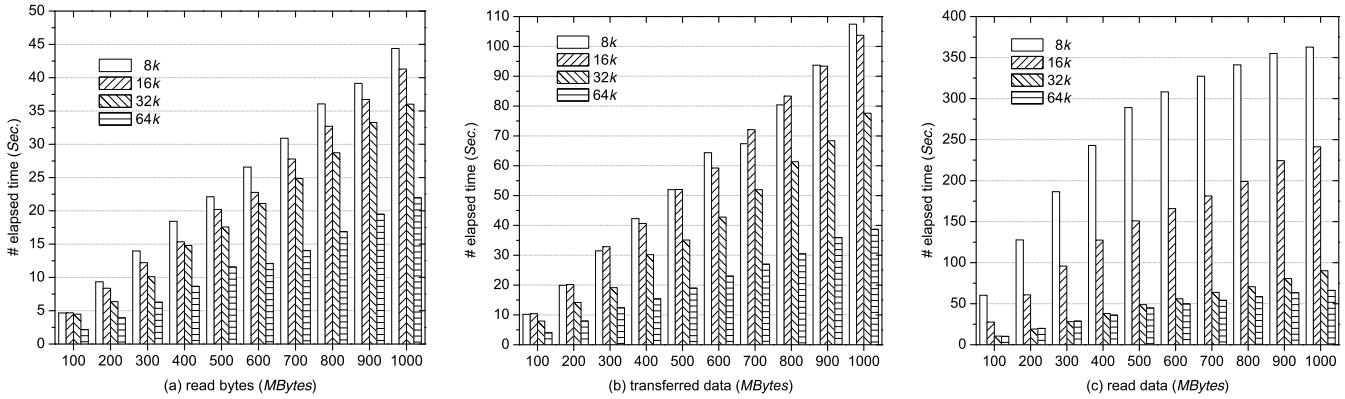

Figure 7. Performance of new transfer protocol by varying data packet (a) sequential reads, (b) local random reads, and (c) random reads

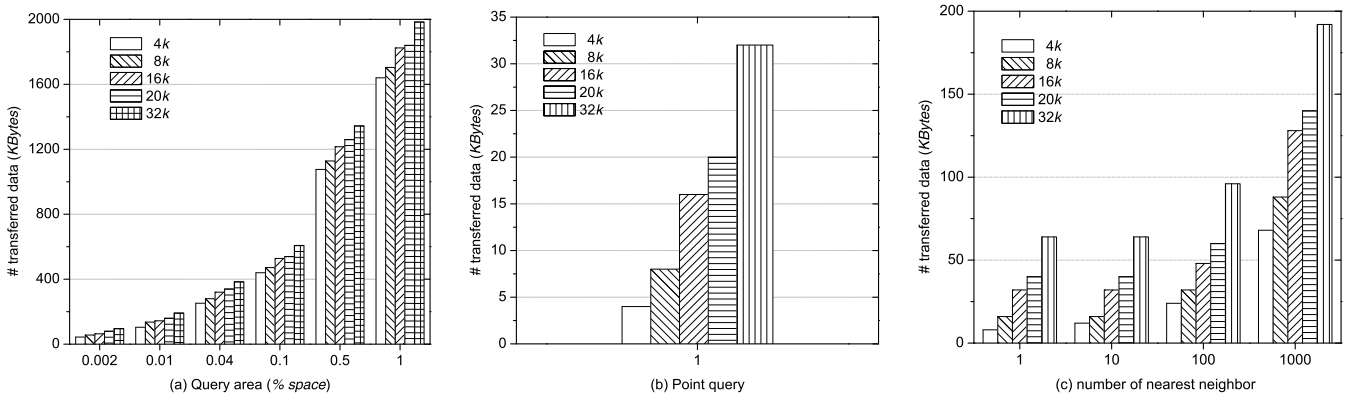

Figure 8. Transferred data in query processing by varying index node size: (a) required data for range query, (b) required data for point query, and (c) required data for *k*-nearest-neighbor query

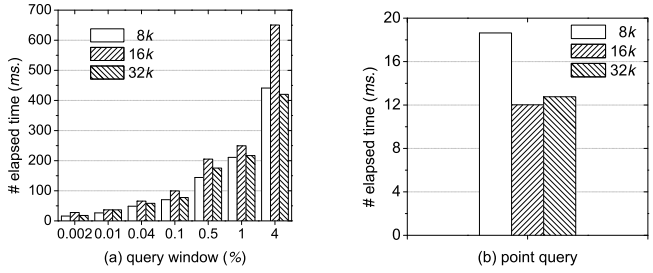

Figure 9. Performance comparison by varying index node size (a) range query, and (b) point query

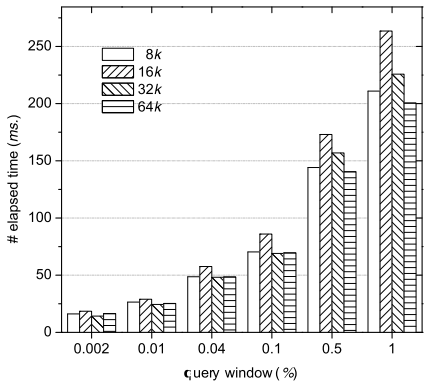

Figure 10. Query performance by varying transfer data packet

Data packet of 16*k* is again the worst due to its poor data transfer efficiency.

E. Effects of buffer

Figure 11 shows the effects of buffer in terms of the response time. We perform 100 range queries of 1% of total space on TLK data set and report the average response time. The buffer pool size is the total size of all buffer pages in data nodes, including buffers for internal and leaf nodes of index. As illustrated, the response time decreases as the available buffer increases. The further improvement of response time can be expected with the increase of buffer size.

VI. Related work

Mouza et.al. [24] proposed scalable distributed R-tree (SD-Rtree), a dynamic distributed index framework support-

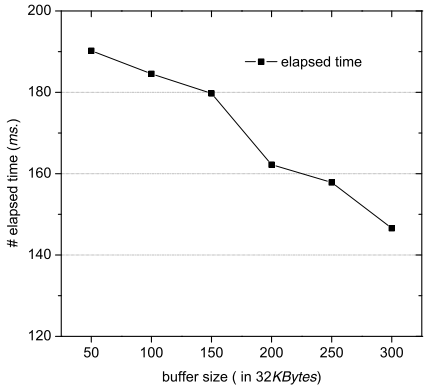

Figure 11. The effects of buffer size

ing incremental index construction. Data nodes in SD-Rtree are conceptually categorized into two types: routing nodes, which keep metadata about all sub-nodes and their siblings to facilitate the query processing, and data nodes, which keep the underlying detailed data objects. The routing nodes and data nodes are organized as a red-black tree to ensure that the height of SD-Rtree is logarithmic in the number all nodes. Further improvement of SD-Rtree [25] with respect to the flexible data allocation policy was proposed. However, dynamic deployment of more servers might be incapable, since the number of servers in SD-Rtree depends only upon the data sets size and the disk capacity of each server.

Aguilera et.al. [26] presented a general and scalable index of B-tree supporting atomically transaction in the context of distributed file system, called Sinfonia [27], which provides the fault tolerance and distributed atomic primitives. The online data migration of B-trees between different data nodes is used for load balancing, and the incremental construction of the index structure was discussed. However, the exploiting of multiple resources towards more efficient query processing for complex query types on this framework remains open.

An indexing framework for query processing was presented in [28]. This framework, consisting of global index in global layer and local index for data chunk, can support both hash index and B-tree and its variants in a hierarchical fashion. All the data nodes are organized in a Peer-to-Peer overlay, and global index is maintained over the structured overlay. Yang and Parker [29] suggested using index to accelerate MapReduce in search engine data processing. Index is composed of several layered files. Index construction and index traversal is implemented by applying Hadoop MapReduce, and experimental evaluations on Hadoop were given. However, this approach is inefficient in handling high selectivity and complex queries. Zhang et.al. [30] proposed a multi-dimensional index, EMINC, for cloud data management purpose. EMINC is a hybrid index structure including R-tree index in master nodes and KD-tree at slave nodes. EMINC can support point query, range query, as well as sort of data mining algorithms, like K-Means and DBScan. However, this framework cannot be applied to typical DFS, like GFS or HDFS, since the master nodes and slave nodes have distinct structure.

Another interesting research direction is the use of distributed key-value storage system as a generic infrastructure to accommodate structured and semi-structured data. BigTable [31] is a distributed scalable column-oriented storage system, providing good performance on read-intensive and write-intensive applications. HyperTable [32] and HBase [33] are open-source implementation under the guide of BigTable principles. Dynamo [14] is a key-value store based on consistent hash to organize data, and multiple datacenters geographically located in different area is supported. In this case, consistency is sacrificed for availability when failure or network partition happens in order to provide acceptable online experiences. PNUTS [34] by Yahoo! is another distributed database system that aims at the supporting of high concurrent read operation with lower latency. PNUTS trades off the consistency for availability and stringent performance suited for web applications. A direct mapping approach is used to identify the location of the required data instead of hashing function, as used in Dynamo. Fawn [35] is a distributed storage system targeting at low-power data intensive computing by employing low-power embedded CPU and local flash storage. Other large scale distributed storage systems include Amazon's SimpleDB services, Microsoft's CloudDB, Facebook's Cassandra [36], and LinkedIn's Voldemort [37].

VII. Conclusion and future work

In this paper, we present a method of hierarchical structures that can be applied to both B-tree and R-tree and their variants in order to facilitate the data organization and query processing on HDFS. By exploiting the natural property of HDFS, we give several enhancement of index structure with respect to node size, buffer strategy, and query processing in order to bridge the gap between these tree-like index designed for single processor environment and distributed file system,like HDFS, and hence, improve the query performance while incurring less overhead in term of both CPU and network transfer. We also propose a data transfer protocol specified for block-wise random reads in HDFS, and study its behavior under real data sets and query processing. We will study the problem of applying MapReduce to index structure to address computational-intensive query, like multi-way spatial join.

Acknowledgment

This work was supported in part by the High Technology Research and Development Program of China (863) under grant No. 2009AA12Z226 and No.2009AA12Z220.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[2] T. A. S. Foundation. (2010) Hadoop distributed file system. [Online]. Available: http://hadoop.apache.org/hdfs/

[3] A. Inc. (2010) Amazon web services, amazon s3. [Online]. Available: http://aws.amazon.com/s3/

[4] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. Beijing, China: ACM, 2007, pp. 1029–1040.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. San Francisco, CA: USENIX Association, 2004, pp. 10–10.

[6] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD*. Boston, Massachusetts: ACM, 1984, pp. 47–57.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, no. 2, pp. 322–331, 1990.

[8] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *Proceedings of the 22th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1996, pp. 28–39.

[9] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *Proceedings of the 13th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1987, pp. 507–518.

[10] B. Sotiris, P. Dieter, and T. Yannis, "Revisiting r-tree construction principles," in *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*. Springer-Verlag, 2002, pp. 149–162.

[11] T. Xia and D. Zhang, "Improving the r*-tree with outlier handling techniques," in *Proceedings of the 13th annual ACM international workshop on Geographic information systems*. Bremen, Germany: ACM, 2005, pp. 125–134.

[12] P. Bozanis and P. Foteinos, "Wer-trees," *Data Knowl. Eng.*, vol. 63, no. 2, pp. 397–413, 2007.

[13] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Multi-way distance join queries in spatial databases," *Geoinformatica*, vol. 8, no. 4, pp. 373–402, 2004.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[15] K. V. R. Kanth, R. Siva, S. Jayant, and B. Jay, "Indexing medium-dimensionality data in oracle," in *Proceedings of the 1999 ACM SIGMOD*. Philadelphia, Pennsylvania, United States: ACM, 1999, pp. 521–522.

[16] S. Berchtold, C. Bohm, H.-P. Kriegel, and U. Michel, "Implementation of multidimensional index structures for knowledge discovery in relational databases," in *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*. Springer-Verlag, 1999, pp. 261–270.

[17] K. Kim, S. K. Cha, and K. Kwon, "Optimizing multidimensional index trees for main memory access," *SIGMOD Rec.*, vol. 30, no. 2, pp. 139–150, 2001.

[18] J. Rao and K. A. Ross, "Making b+-trees cache conscious in main memory," *SIGMOD Rec.*, vol. 29, no. 2, pp. 475–486, 2000.

[19] M. Park and S. Lee, "Optimizing both cache and disk performance of r-trees," in *proceedings of the 14th DEXA*, vol. LNCS 2736. Springer-Verlag, 2003, pp. 137–147.

[20] S. T. Leutenegger, J. M. Edgington, and M. A. Lpez, "Str: A simple and efficient algorithm for r-tree packing," Institute for Computer Applications in Science and Engineering (ICASE), Tech. Rep., 1997.

[21] I. Kamel and C. Faloutsos, "On packing r-trees," in *Proceedings of the second international conference on Information and knowledge management*. Washington, D.C., United States: ACM, 1993, pp. 490–499.

[22] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. Washington, D.C., United States: ACM, 1993, pp. 237–246.

[23] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 265–318, 1999.

[24] C. d. Mouza, W. Litwin, and P. Rigaux, "Sd-rtree: A scalable distributed rtree," in *Proceedings of the 23rd International Conference on Data Engineering*, Istanbul, Turkey, 2007, pp. 296–305.

[25] ——, "Dynamic storage balancing in a distributed spatial index," in *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*. Seattle, Washington: ACM,

2007, pp. 1–8.

[26] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 598–609, 2008.

[27] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. Stevenson, Washington, USA: ACM, 2007, pp. 159–174.

[28] K.-L. W. Sai Wu, "An indexing framework for efficient retrieval on the cloud," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 75–82, 2009.

[29] H.-C. Yang and D. S. Parker, "Traverse: Simplified indexing on large map-reduce-merge clusters," in *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*. Brisbane, Australia: Springer-Verlag, 2009, pp. 308–322.

[30] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *Proceeding of the first international workshop on Cloud data management*. Hong Kong, China: ACM, 2009, pp. 17–24.

[31] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[32] (2010) Hypertable: An open source high performance, scalable database. [Online]. Available: http://www.hypertable.org/

[33] T. A. S. Foundation. (2010) Hbase. [Online]. Available: http://hadoop.apache.org/hbase/

[34] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[35] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Big Sky, Montana, USA: ACM, 2009, pp. 1–14.

[36] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*. Calgary, AB, Canada: ACM, 2009, pp. 5–5.

[37] (2010) Project voldemort: A distributed database. [Online]. Available: http://project-voldemort.com/