LCIndex: A Local and Clustering Index on Distributed Ordered Tables for Flexible Multi-Dimensional Range Queries

Chen Feng*†, Xi Yang‡, Fan Liang*†, Xian-He Sun‡ and Zhiwei Xu*
* *SKL Computer Architecture, ICT, CAS, China*
Email: {fengchen, liangfan, zxu}@ict.ac.cn
†*University of Chinese Academy of Sciences, Beijing, China*
‡*Department of Computer Science, Illinois Institute of Technology, Chicago, USA*
Email: xyang34@hawk.iit.edu, sun@iit.edu

***Abstract*—A lot of Not Only SQL (NoSQL) databases have been proposed in the era of big data. Distributed Ordered Table (DOT) is one kind of NoSQL database that has attracted lots of attention. It horizontally partitions table into regions and distributes regions to region servers according to the keys. Multi-Dimensional Range Query (MDRQ) is a common operation over DOTs. Many indexing techniques have been proposed to improve the performance of MDRQ, but they cannot guarantee high performance on both insert and flexible MDRQ at the same time. In this paper, we propose a novel indexing technique named LCIndex, short for Local and Clustering Index, to solve this issue. Experimental results confirm that LCIndex can achieve high performance on both insert operations and flexible MDRQ.**

***Keywords*-Distributed database; Indexing method; Multi-dimensional range query**

I. Introduction

Data volume grows rapidly in the era of big data. Traditional relational database management systems (RDBMS) are facing challenges, such as high performance, huge storage, high scalability, and high availability. To solve this problem, a series of Not Only SQL (NoSQL) databases have been proposed and widely used in industry and academia, such as Google BigTable [1], Yahoo! PNUTS [2], HBase [3], and Cassandra [4]. Some NoSQL databases, such as Google BigTable, Yahoo! PNUTS and HBase, can be modeled as Distributed Ordered Table (DOT). It horizontally partitions the whole table into regions by continuous keys, distributes the regions to region servers, and replicates the regions for reliability and performance. Each region in DOTs contains data with keys located in a certain range; regions between region servers are ordered by keys and do not overlap. DOTs naturally support range queries on keys by locating and scanning regions with the start key and end key.

Multi-Dimensional Range Query (MDRQ) is a common operation in databases and plays an important role in many applications. For example, to report the real-time traffic jam in a city with millions of cars, the traffic flow detection system must execute MDRQ every minute. The expression of MDRQ may look like: "select ALL from TABLE where latitude $>$ 50.18 AND latitude $<$ 50.25 AND longitude $>$ 100.86 AND longitude $<$ 101.05 AND direction = 'west' AND speed $<$ 20.0". Databases for similar cases expect both high insert and MDRQ performance, while they mitigate the requirements on delete and update operations. Since DOTs only support range queries over keys, MDRQ over non-key dimensions has to scan the whole table to get the proper results. Due to the huge volume of data, this method is inefficient and results in low throughput and high latency.

To improve the performance of MDRQ over DOTs, a series of indexing techniques have been proposed. Some of them, like UQE-Index [5] and MD-HBase [6], built spatial trees on the top of DOTs. However, the performance of spatial trees relies on balanced data whereas the data distribution of indexed columns is usually unbalanced (e.g. limited traffic speed in a jam), which limits the flexibility of MDRQ of these indexing techniques. For example, the MDRQ performance of k-d tree and R-tree degrades when the tree is skew. On the other hand, maintaining a balanced tree such like R^+-tree is complex and decreases the performance of insert [7].

To support MDRQ over DOTs without assumption on data distribution, namely flexible MDRQ in this paper, some non-tree indexing techniques build indexes on table level for each indexed dimension, such as CMIndex [8], IHBase [9], ITHBase [10], Asynchronous view [11], CCIndex [12], HIndex [13], and IRIndex [14]. These indexing techniques consider the trade-offs between implementation complexity, insert performance, MDRQ performance, and storage overhead. For example, CCIndex is easy to implement and has high flexible MDRQ performance, but it suffers from poor insert performance and expensive storage cost; IRIndex is on the contrary of CCIndex. Overall, none of the current techniques can guarantee high performance on both insert and flexible MDRQ simultaneously.

In this paper, we classify the typical non-tree indexing techniques and analyze their pros and cons. Then, we propose a novel indexing technique, LCIndex (Local and Clustering Index), which builds clustering indexes on local file systems. We implement a prototype of LCIndex based on HBase, conduct experiments to compare it with HBase,

CMIndex, CCIndex, and IRIndex to distinguish our contributions.

The contributions of this paper are threefold:

1) Based on studying literatures and projects, we conclude a taxonomy on non-tree indexing techniques targeting flexible MDRQ over DOTs.
2) We design LCIndex, aiming to provide high performance on both insert and flexible MDRQ.
3) We implement a prototype of LCIndex based on HBase, our experiments show that LCIndex has high insert performance, high flexible MDRQ performance, and low network traffic distinguished from other representative indexing techniques.

The rest of this paper is organized as follows. Section II introduces the motivation of LCIndex. Section III elaborates the design and the query optimization of LCIndex in detail. The evaluation results and analysis are presented in Section IV. Section V lists the related work. Section VI concludes this paper and proposes future work.

II. Motivation

To meet the demand of high performance on insert and flexible MDRQ over DOTs, we study a series of indexing techniques. Indexing techniques leveraging spatial trees can provide high insert and MDRQ performance, at the cost of losing flexibility of MDRQ. Since the flexibility is conflict with the inherent design of spatial trees, we review the non-tree indexing techniques to explore the potential of providing high performance on insert and flexible MDRQ simultaneously.

A. Index Classification

1) Secondary/Clustering: It is conventional to divide the ordered index structures into two groups by the layout of index data: secondary index and clustering index [15]. This classification suits DOTs as well. The secondary index provides a mapping between the value of indexed dimension and the key of raw table, while clustering index stores a copy of the raw data in each index. An example is given in Figure 1, in which the key of index table is pieced by the value of indexed dimension and the raw key.

The procedures of MDRQ on secondary index and clustering index are also shown in Figure 1. The secondary index takes three steps, it 1) gets the candidate key lists by scanning all index tables, 2) merges the common keys of the candidate key lists, and 3) reads the raw table by the common keys. MDRQ on clustering index is straightforward, it only needs to select an index table and scan the selected table with filter conditions. With the performance consideration, clustering index usually selects one indexed dimension with the minimum number of records to be scanned.

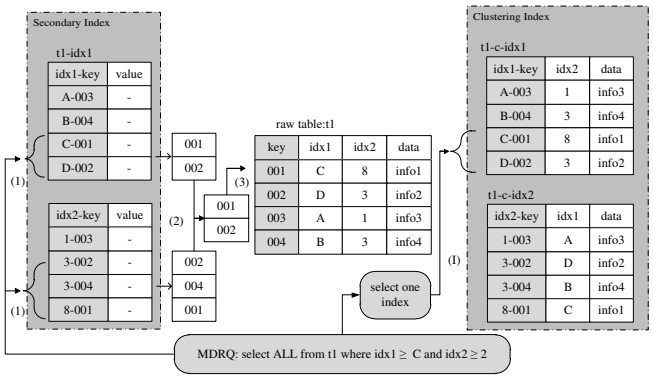

Figure 1. Classification according to index data layout and MDRQ procedure: (1)(2)(3) for secondary index and (I) for clustering index

2) Global/Local: According to the location of index record and raw record, the non-tree indexing techniques can be classified to global index and local index. The global index manages the index table as a common table of DOT, and the regions of the index table will be distributed to different region servers. The local index co-locates the index records and the raw records on the same region server. The storage of raw data is balanced by DOT. While the size of indexes is related to the size of raw data, local index can balance storage inherently.

Figure 2 gives an example of global index and local index in the layout of secondary index. For global index, an index record (D-002 in table G-t1-idx1) may be distributed to another region server compared to the raw record (002 in table t1) while the records of local index are always co-located with the raw records. The insert procedures of global index and local index are the same. During MDRQ, local index must spread the query to all region servers because index records are out-of-order among region servers.

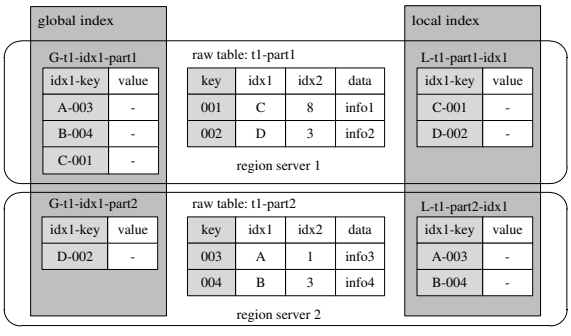

Figure 2. Classification according to index data distribution

B. Pros and Cons

Table I lists the pros and cons of the four classes in four metrics. The concise structure of secondary index leads to higher insert performance, lower storage overhead compared with clustering index. However, its MDRQ performance is

lower due to the considerable cost introduced by a more complicated procedure. The local index has lower network traffic compared to the global index because index records are operated locally. By leveraging the APIs of DOT, the global index can maintain indexes and process MDRQ in a simple way. To co-locate the index records, the local index usually modifies underlying code, which increases the complexity of implementation. For example, HIndex uses a special balancer to co-locate the index tables whereas IRIndex directly writes index files on local file system.

Table I
COMPARISON BETWEEN DIFFERENT INDEXING TYPES

Indexing Type	Pros	Cons
Secondary	High insert performance, low storage overhead	Low MDRQ performance
Clustering	High MDRQ performance	Low insert performance, high storage overhead
Global	Easy to implement	High network traffic
Local	Low network traffic	Complex MDRQ procedure

C. Existing Non-Tree Indexing Techniques on DOT

Based on the taxonomy above, the non-tree existing indexing techniques on DOT are listed in Table II.

Table II
EXISTING INDEXING TECHNIQUES ON DOT

Indexing Type	Global	Local
Secondary	CMIndex, ITHBase, Apache Phoenix	IHBase, HIndex, IRIndex, Apache Phoenix
Clustering	GCIndex, CCIndex	None

1) Global & Secondary: CMIndex (Client-Managed Index) is the basic secondary index, which is completely managed in the application layer. ITHBase (Indexed-Transactional HBase) [10] focuses on transaction support of index. The synchronization overhead for transaction is considerable, which degrades the overall performance.

2) Global & Clustering: GCIndex (Global and Clustering Index) builds clustering indexes and replicates all index tables, which is simple to implement but involves huge storage overhead. Hence, CCIndex (Complemental Clustering Index) sets the replication factor of index table to one (the default value of HBase is three) and creates replicated Complemental Check Table (CCT) to help data recovery from failure. During MDRQ, GCIndex and CCIndex leverage the region-to-server mapping information to estimate the data size covered by each query condition, and select the query condition with the minimum data size to execute the real scan.

3) Local & Secondary: IHBase (Indexed HBase) builds secondary indexes in memory when a region is opened for the first time or a MemStore is flushed. There is no index for the un-flushed data, IHBase directly scans the in-memory data when processing queries. The drawback of IHBase is clear, it consumes considerable memory space and time to maintain indexes. HIndex (Huawei-Index) builds indexes in separate tables, leverages co-processors to build and maintain indexes. HIndex uses a custom region load-balancer to co-locate the index regions with the raw regions. IRIndex (Inside Region Index) builds indexes for each HFile rather than region, it builds indexes when a MemStore is flushed. The index records are written into a customized IndexFile instead of HFile in order to reduce storage cost. When processing MDRQ, IRIndex sorts common keys of raw table before reading them. By converting the massive random reads to a set of sequential reads, IRIndex significantly decreases the total disk seek time at the cost of higher latency to get the first result. Apache Phoenix is an open-source project that can execute SQL queries, which provides secondary index both globally and locally for different use cases. When using local index, Apache Phoenix stores local indexes of a table in a single, separate shared table.

4) Local & Clustering: To our best knowledge, there is no local and clustering indexing technique.

D. Comparison

Table III
EXISTING INDEXING TECHNIQUES ON DOT [14]

Metric	CMIndex	CCIndex	HIndex	IRIndex
Insert throughput	27%	23%	71%	78%
Insert latency	412%	444%	16%	10%
MDRQ throughput	4%	330%	47%	137%
Storage overhead	40%	123%	58%	9%

Authors of [14] did experiments to compare performance of CMIndex, CCIndex, HIndex and IRIndex, which is shown in Table III. The performance on non-indexed HBase is the baseline. As the results indicate, MDRQ throughput of CMIndex is only 4% of a raw table scan, because CMIndex needs to get all candidate keys from all index tables when processing MDRQ, which is time-consuming. IRIndex outperforms HIndex in MDRQ because sequential reading is much faster than random reading in HBase. On the other hand, the latency of IRIndex is high due to additional sorting operations.

To sum up, the spatial tree indexing techniques can achieve high insert and MDRQ performance, but not flexible; these non-tree indexing techniques can provide flexible MDRQ, but none of them can obtain high insert and MDRQ performance at the same time. To this end, we propose LCIndex (Local and Clustering Index), which can achieve high performance on both insert and flexible MDRQ at a modest cost of storage.

III. Design and Implementation

A. HBase Architecture

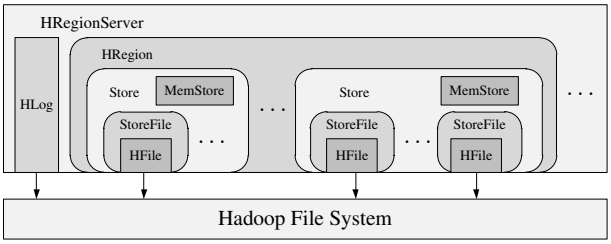

Figure 3. Architecture of HBase [8]

We implemented the prototype of LCIndex based on HBase. The architecture of HBase is presented in Figure 3. In the following text, the term "column" in HBase is the counterpart of "dimension" in MDRQ.

HBase consists of three parts, one HMaster, several HRegionServers, and several clients. A table in HBase will be horizontally partitioned into HRegions. HMaster manages the metadata and distributes HRegions into different region servers. Each HRegion has one Store for each column family. Each Store includes one MemStore and several StoreFiles. Like its name, MemStore is an in-memory structure. Each StoreFile has a corresponding HFile, which is the stored on HDFS. HBase adopts write-ahead logging (WAL) for fault tolerance, operations first write the event into HLog before execution. WAL will be discarded only after the corresponding operations have been applied to HFiles.

B. Challenges

HIndex employs co-processors to build and maintain indexes. The index records are written in a co-located separate table. However, there are two main drawbacks for clustering index that HIndex proposed: 1) co-processor operations introduce high latency to build and insert clustering indexes, and 2) the number of records to be scanned for each condition is hard to estimate, which may decrease the MDRQ performance. Thus, we build and maintain indexes in a way similar to IRIndex, say, building indexes when a MemStore is flushed, and maintaining HFile-level indexes for each StoreFile, which leads to some other challenges.

1) Index storage: IRIndex stores indexes in customized *IndexFiles* to save storage overhead, however, the format of *IndexFile* is not suitable for LCIndex. Leveraging HFile to store LCIndex is feasible, but needs extra work.

2) Index building: IRIndex builds indexes when a MemStore is flushed. It works well for IRIndex but meets problems for LCIndex because generating clustering record is a complicated and time-consuming procedure, which can easily reach the timeout threshold of HBase. In our previous test on a small cluster, IRIndex costs 0.25 seconds to generate indexes while clustering index costs 3.1 seconds. Raising the parameter *hbase.rpc.timeout* can not eliminate the *TimeoutException* problem.

3) Index maintenance: LCIndex must compact index files when HFiles are compacted or split. Since the data size grows bigger in compaction, the timeout threshold may be reached. When a region is split and a child region is partitioned to a remote region server, some index files on the child region may not be found on the hosting node.

4) MDRQ optimization: CCIndex counts the number of regions covered by query condition to estimate the number of records to be scanned. However, this does not work for LCIndex because index records are out-of-ordered between regions. Choosing an index table randomly works but inefficient, an effective way to select a proper indexed column is necessary.

C. Prototype of LCIndex

1) Index storage: HFile as the storage format of index files (IFile) meets our design. Hence, we reuse it rather than designing a new format to avoid unnecessary engineering efforts. By improving existing HBase methods, LCIndex writes index records to local file system in HFile format.

For each HFile, LCIndex builds several IFiles according to the number of indexed columns, as shown in Figure 4. Compared with writing all index records in one IFile, this way has three advantages: 1) each IFile has a similar size with the raw HFile, the time spent on reading or scanning such IFiles is acceptable, 2) it is much easier to estimate the number of records scanned for MDRQ since each indexed column can be calculated separately, and 3) after selecting the indexed column to scan, LCIndex only needs to scan the IFile on that column, which significantly reduces scanning overhead therefore enhances MDRQ performance.

To calculate the result size of MDRQ, LCIndex urges users to describe indexed column like "PRICE(column name) DOUBLE(data type) 0(minValue) 99999(maxValue) 100(flagNum)". When building indexes, LCIndex calculates the *flagNum* flags and writes a statistic file (SFile) together with IFile. Each SFile contains *flagNum+1* lines, where line i indicates the number of records whose value locates in range $(flag_{i-1}, flag_i]$. The SFile is co-operated with corresponding IFile (e.g. building, deleting, etc.) if not stated. By default, LCIndex stores IFiles and SFiles on local file system to accelerate MDRQ.

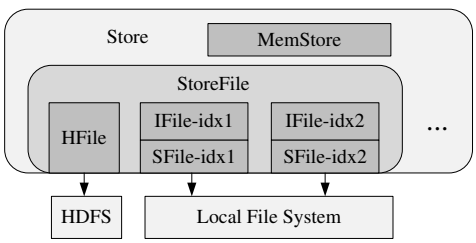

Figure 4. Index storage on local disk

2) Index building and maintenance: Operations in building and maintaining HFiles in raw HBase can be divided into six stages; they are *Flush*, *Commit*, *Compact*, *CompleteCompact*, *Archive*, and *Split*. *Flush* creates HFile under temporary directory and appends records into it, *Commit* moves the HFile to online directory after *Flush*. *Compact* compacts several HFile into one temporary file, *CompleteCompact* moves the compacted temporary file to online directory and *Archive* deletes the HFiles used for compaction. *Split* splits a region into two sub-regions; in HFile level, two reference files are created for a parent HFile in the parent region. The reference does not contain any data, it points to the top/bottom of the parent HFile and will be deleted are the compaction in the sub-region.

To avoid the *TimeoutException* caused by building and maintaining indexes synchronously, LCIndex builds five jobs, say *FlushJob*, *CommitJob*, *CompactionJob*, *CompleteCompactJob*, and *ArchiveJob*; and creates a job queue for each kind of job to manage IFiles in background. *RemoteJob* and *RemoteJobQueue* are introduced to help processing MDRQ and recovering indexes, which will be explained later. Similar to the stage dependencies of different stages, we show the dependencies between jobs in Figure 5. The dependencies must be carefully handled to ensure correctness.

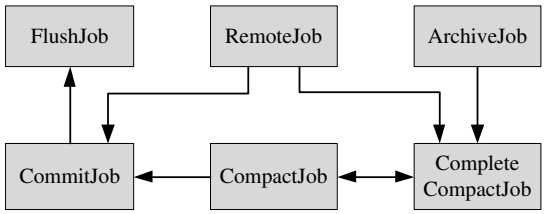

Figure 5. Dependencies between jobs

For the four basic operation *Flush*, *Commit*, *CompleteCompact*, and *Archive*. LCIndex operates IFiles and SFiles according to the operation on the raw HFile. As an exception, LCIndex does nothing for *Split*. This is because HBase can directly read the top/bottom half of the parent HFile of the reference file, but since IFiles are re-ordered by the indexed column, restoring the order on raw keys and splitting the IFile is complex and time consuming, splitting SFiles is also impossible because it only contains statistic data. This exception is handled during the compaction of IFiles, the algorithm is shown in Algorithm 1.

After compacting the raw HFiles, LCIndex creates a new compact job for IFiles and adds it in the *CompactJobQueue*. The action of the job is decided by how many reference files are compacted. If there are any reference files, the *CompactJob* will be marked to *rebuild* and drop all related jobs; then it will generate new IFiles by travering the new generated HFile when triggered by *CompactJobQueue*. Otherwise, LCIndex checks the existence of IFiles. If an IFile is not stored on local file system, LCIndex firstly tries to find the corresponding jobs in *CommitJobQueue* and *CompleteCompactJobQueue*. If there is no such a job, LCIndex will create a new *RemoteJob* to get files from the other region servers. After that, the compact job adds this job as a related job. A *non-rebuild* compact job will check the status of all related job and run after all jobs are finished. If any related job fails, the compact job will run as a *rebuild* compact job. Otherwise, LCIndex leverages *Compact* method of HBase to compact the corresponding IFiles directly because IFiles are in HFile-format, which is shown in Figure 6.

The compaction of SFiles is straightforward, as the SFiles for the same indexed column has the same number of lines. The value of i_{th} line in the new generated SFile is the sum of the values of i_{th} line from the compacted SFiles.

Algorithm 1 Compact IFiles

Require: run $newFile \leftarrow Compact(compactedFiles)$ before compacting IFiles
Input: $compactedFiles \leftarrow$ HFiles to be compacted
Input: $newFile \leftarrow$ the HFile newly generated
1: $cJob \leftarrow$ new $compactJob(compactedFiles, newFile)$
2: $cJob.rebuild \leftarrow false$
3: **for** $file$ in $compactedFiles$ **do**
4: **if** $file.isRef$ **then**
5: $cJob.rebuild \leftarrow true$
6: break
7: **else if** $localfs.has(file.IFiles) = false$ **then**
8: $job \leftarrow inCommmitOrCompleteQueue(file)$
9: **if** $job \neq NULL$ **then**
10: $job \leftarrow$ new $RemoteJob(file.IFiles)$
11: **end if**
12: $cJob.addDep(job)$
13: **end if**
14: **end for**
15: **if** $cJob.rebuild = true$ **then**
16: $cJob.dropDepJobs()$
17: **end if**
18: $CompactJobQueue.add(cJob)$

3) Query processing: Based on SFiles, the MDRQ algorithm is shown in Algorithm 2. Line 3 $\sim$ 11 calculate the number of records covered by query condition on each indexed columns and marks the missing IFile. Line 12 selects the indexed column with the minimal number of records to be scanned while the missing IFile should not exceed a threshold. Line 14 executes MDRQ on this selected column, for missing IFiles, LCIndex will scan the whole raw HFile to get results. If every indexed column misses a lot of IFiles, LCIndex will run MDRQ on raw HBase, as seen in line 16. After processing MDRQ, LCIndex will check the missing IFiles and again adds them to $RemoteJob$.

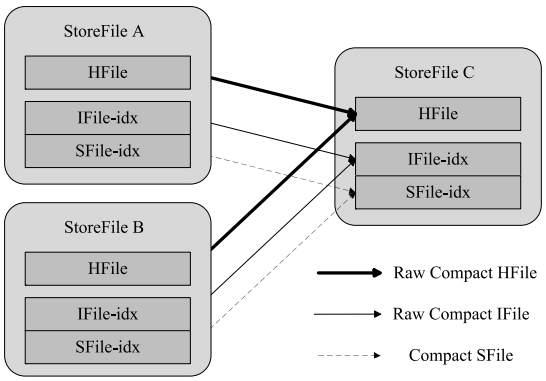

Figure 6. Example of compacting two StoreFiles

Algorithm 2 MDRQ processing based on statistic files

Input: $store \leftarrow$ the store to query
Input: $indexColumns \leftarrow$ index columns build
Input: $columnRanges \leftarrow$ condition on each index column
1: $gCount$[index column number] $\leftarrow 0$
2: $gMiss$[index column number] $\leftarrow 0$
3: **for** $storeFile$ in $store$ **do**
4: **for** $idxCol$ in $indexColumns$ **do**
5: **if** $storeFile.IFiles[idxCol]$ exists **then**
6: $gCount[idxCol] \leftarrow gCount[idxCol]+$ number covered by $storeFile.SFiles[idxCol]$)
7: **else**
8: $gMiss[idxCol] \leftarrow gMiss[idxCol] + 1$
9: **end if**
10: **end for**
11: **end for**
12: $col \leftarrow gCount.MinColNotExceedMissingThreshold$
13: **if** $col \neq null$ **then**
14: execute query on col, scan the whole raw HFile for missing IFiles
15: **else**
16: scan raw HBase with conditions
17: **end if**
18: create $RemoteJob$ for missing IFiles and SFiles

4) RemoteJob: A *RemoteJob* is created when a IFile is not found on local file system, there are three potential causes for the "data lost": 1) the source HFile is a reference, 2) the region is recently migrated and the IFile is stored remotely, or 3) node failure.

Therefore, a *RemoteJob* checks each potential causes in the following order: 1) if the source HFile is a reference, the job checks the parent IFiles. If they are stored on local file system, the job reads corresponding IFiles by adding filter on raw keys to distinguish the top/bottom half; if not, the job treats it as node failure. 2) the *RemoteJob* communicates with all other region servers for the lost IFiles, each region node checks the target files in its local file system, *CommitJobQueue* and *CompleteCompaction-JobQueue*. If the corresponding local IFile is found, the IFile will be transfered via network. If the IFile is in the job queues, which means it will be generated later, the *RemoteJob* will be asked to wait until the file is ready. 3) If the IFile cannot be found anywhere, LCIndex treats it as node failure, which means LCIndex will create a new *CompactJob* to "'compact" the corresponding HFile.

D. Discussion

1) Data Reliability: We have considered building local complementary clustering indexes. It sets the replication factor of raw data and indexes to one, and leverages CCTs to recover data. But soon we found such method cannot work because the raw data and all related indexes are stored on the same node. Once the node fails, the raw data and the indexes will be lost because the data cannot be recovered from CCTs only. Replicating IFiles is expensive due to high storage overhead. A cost-effective way is to set the replication factor of IFiles to one, LCIndex can recover IFile and SFile according to the rebuilt indexes from the *RemoteJobs*.

2) Add/Remove indexes dynamically: Compared with other existing indexing techniques, a remarkable feature of LCIndex is the natural support to adding/removing indexes dynamically. Once a new indexed column is added, LCIndex will "miss" corresponding IFiles when processing compaction or MDRQ. As explained in Algorithm 1, the *CompactJob* will use the new generated HFile to build indexes in compaction; in MDRQ, the *RemoteJob* will also build indexes based on the existing HFiles because no region server contains the target IFiles, which is described in Algorithm 2. Removing indexes is straightforward, LCIndex urges all region servers to delete IFiles and SFiles corresponding to the removed indexed column from their local file systems.

IV. Evaluation

A. Environment

We use a cluster that consists of thirteen nodes interconnected by a Gigabit Ethernet switch as our testbed. One node is used as single NameNode/HMaster, seven nodes are used as DataNode/HRegionServer, and the other five nodes are used as clients to submit requests. Each node is equipped with a Quad-Core AMD Opteron(tm) Processor 2376, 8 GB RAM, and one 150 GB SATA disk. In order to adapt to the existing implementations, such like CMIndex, GCIndex, CCIndex and IRIndex, we implement LCIndex based on HBase 0.94.16.

B. Experiment Design

YCSB [16] is a widely used benchmark to evaluate NoSQL database, it generates data in different distributions and workloads, but the data are not accordant with the actual

case. So we leverage TPC-H, the famous benchmark for relational databases. We select *order.tbl* generated as our input (5 million rows), and build three indexed columns, say *date*, *total-price* and *priority*; other columns are used as non-indexed columns. All columns are designed under a single column family.

In the *Insert* experiment, for each indexing technique, we 1) create an empty table and add indexes on it, 2) insert records via five clients, each client runs 30 threads, and 3) flush the table. Five MDRQ requests with different conditions are submitted by a single client. The client gets 1000 records from server each time.

We measure the execution time, and collect the insert latency and network traffic during the insert and queries, respectively. In order to reflects the modest overhead of indexing techniques, we measure the insert latency when HBase is heavy-loaded. The network traffic is collected every second on the NameNode/HMaster node and the seven DataNode/HRegionServer nodes by $dstat$.

C. Results

1) Insert throughput: Figure 7 shows the throughput by using different techniques. The insert throughput of CMIndex, GCIndex, CCIndex, IRIndex and LCIndex is 18.5%, 13.6%, 17.2%, 90.6% and 72.5% compared to raw HBase, respectively. The insert throughput of LCIndex is 422% of CCIndex and 80% of IRIndex. The insert throughput of CMIndex, GCIndex and CCIndex are very poor because the indexes are built on client side. IRIndex operates indexes with raw HFiles synchronously, though building secondary is fast, it still decreases the insert throughput. LCIndex builds and maintains indexes in background, but the interference to throughput is clear due to the resource competition on region servers.

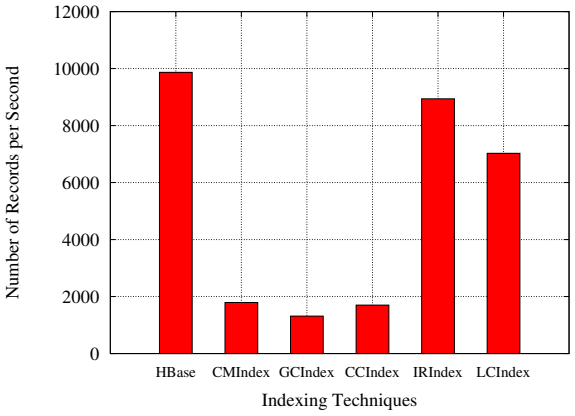

Figure 7. Insert throughput

2) Insert latency: Figure 8 presents the cumulative distribution function (CDF) of latency. Up to 96.9% insert operations return from server for HBase, IRIndex and LCIndex within 0.02 seconds because they only write the raw records into memory. About 96% insert operations of CMIndex and CCIndex return within 0.1 seconds, and more than 5% insert operations of GCIndex return after 0.2 seconds because four tables with three replications are written.

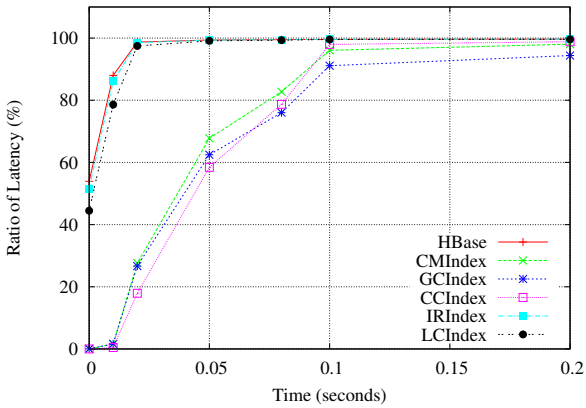

Figure 8. CDF for insert latency

3) Query processing: Figure 9 presents the results of scanning five different queries, the meaning of each query is explained in Table IV. Both IRIndex and LCIndex have lower performance than CMIndex, GCIndex and CCIndex in a single indexed column Scan-A, because they must scan all related MemStore.

In MDRQ-A, B, C and D, the total elapsed time of CMIndex are 606.9, 1748.2, 1413.6 and 1288.3 seconds respectively. This is because CMIndex must collect all candidate raw keys in all conditions, and the condition $priority =$ "3-MEDIUM" covers about 20% records in our test. The MDRQ performance of GCIndex and CCIndex are nearly the same because their procedure are same. From MDRQ B to D, the efficiency (the reciprocal of time) ratio between LCIndex and CCIndex decreases from 54.6% to 43.4%, because the proportion of MemStore scanned by LCIndex is increasing, which is inevitable. On the other hand, the efficiency ratio between LCIndex and IRIndex raises from 183% to 407% because the number of records scanned by LCIndex decreases more rapidly than that of IRIndex.

We measure the time spent on selecting the indexed column during queries on LCIndex, the average and maximum time are 0.0011 and 0.025 seconds respectively. Since the size of SFile is small and the algorithm to compact SFile is straightforward, it involves little overhead to build and maintain SFiles. Thus, it is worthwhile to introduce SFiles because the MDRQ execution can be accelerated.

4) Storage overhead: The storage of CMIndex, GCIndex, CCIndex, IRIndex and LCIndex is 1.35x, 4.58x, 2.76x, 1.44x and 2.31x compared to raw HBase, respectively. The storage cost of HBase without index and the five

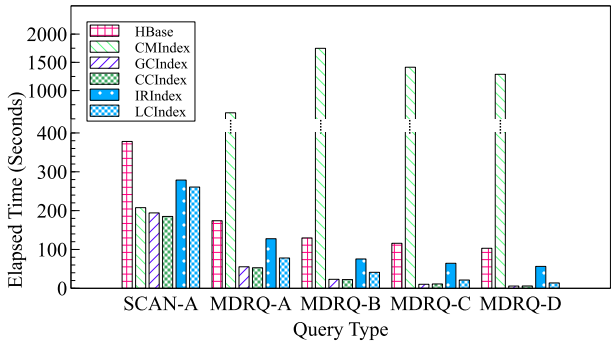

Figure 9. Query runtime

Table IV
EXPLANATION OF QUERIES

Query Type	Conditions	Number of Records
Scan-A	$20000 \leq totalPrice \leq 45000$	390138
MDRQ-A	$20000 \leq totalPrice \leq 45000$ AND $date \geq 19970310$	82892
MDRQ-B	$20000 \leq totalPrice \leq 45000$ AND $date \geq 19970310$ AND $priority =$ "3-MEDIUM"	16508
MDRQ-C	$20000 \leq totalPrice \leq 35000$ AND $date \geq 19971110$ AND $priority =$ "3-MEDIUM"	4889
MDRQ-D	$25000 \leq totalPrice \leq 35000$ AND $date \geq 19980310$ AND $priority =$ "3-MEDIUM"	1818

different indexing techniques is shown in Figure 10. Both CMIndex and IRIndex have small storage overhead because they are secondary indexing techniques. GCIndex has the highest storage overhead because all data has three replicas. CCIndex saves about 40% space compared with GCIndex because the CCIT has only one replication. In theory, the storage cost of LCIndex should be about 2.0x compared with HBase but 2.31x in our test, this is because IFile has longer keys and is not compressed.

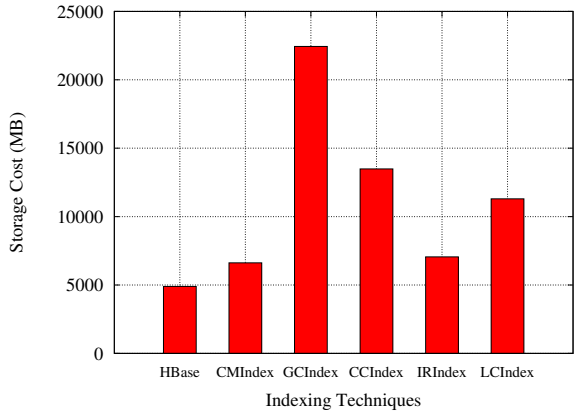

Figure 10. Storage cost

5) Job waiting time: Since the jobs to maintain indexes are executed in background, it is important to know how long one job needs to wait before execution. Figure 11 presents the cumulative distribution function of job wait time for different type of jobs. Both *FlushJob* and *CommitJob* wait no longer than 25 seconds because their dependencies are simple. A *CompactJob* may rely on several *CommitJobs* and *CompleteCompactJob*, which prolongs its wait time. As a result, corresponding *CompleteCompactJob* and *ArchiveJobs* also have long wait time. Though some *CompactJobs* will be extended to 120 seconds, the average wait time of *CompactJob* is as low as 3.55 seconds, which indicates most *CompactJobs* can start in a short time.

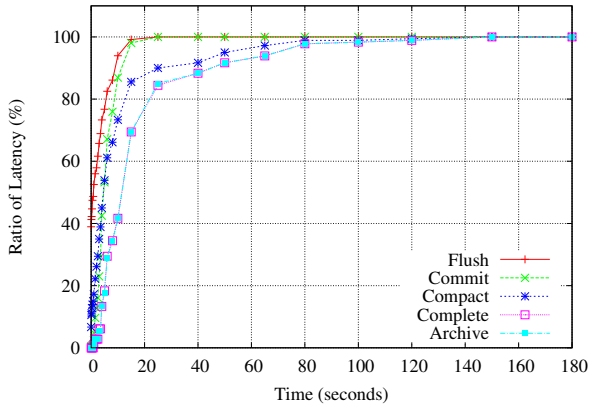

Figure 11. CDF for job waiting time

6) Network traffic: We accumulate the package sizes received during insert and sent during queries on all region servers, which is illustrated in Figure 12.

The total package size received by region servers during insert is nearly the same among HBase, IRIndex and LCIndex since the indexes on IRIndex and LCIndex are built on local file system. The package size in CCIndex is smaller than CMIndex because some HFiles of CCTs are compacted and deleted before replicated by the background thread. The package size sent by region servers during queries is almost the same between HBase, GCIndex, CCIndex, IRIndex, and LCIndex, because all of them execute query in local node and only those records that meet the condition are returned. The size of LCIndex is slightly bigger than others because the *RemoteJobs* sends IFiles between region servers. CMIndex involves about 5x network traffic compared with other indexing techniques, because lots of candidate results are sent to clients during MDRQ.

7) Summary: We summarize the results of insert throughput, MDRQ performance of the MDRQ-B∼D, and the storage cost in Table V. All values are normalized based on the values of HBase. We find that CMIndex is poor on both insert and MDRQ and not suitable for practical applications. Both GCIndex and CCIndex are well suited

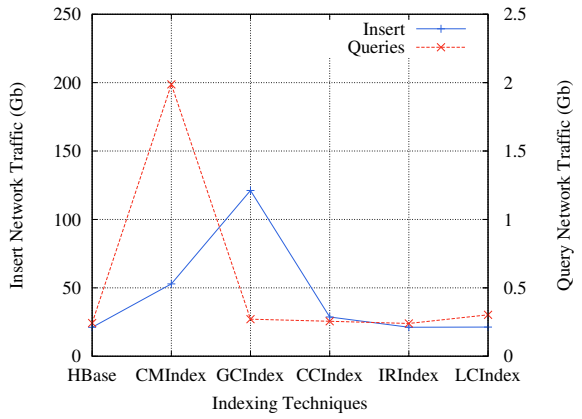

Figure 12. Network traffic during insert and scan

Table V
NORMALIZED RESULTS OF INDEXING TECHNIQUES

Indexing Technique	Insert Throughput	MDRQ Throughput	Storage Cost
HBase	1	1	1
CMIndex	0.185	0.07∼0.08	1.35
GCIndex	0.136	5.6∼17.80	4.58
CCIndex	0.172	5.7∼17.17	2.76
IRIndex	0.906	1.72∼1.83	1.44
LCIndex	0.725	3.15∼7.46	2.31

for workloads that favor high MDRQ performance and do not concern about insert performance. CCIndex performs better in insert and saves lots of storage. IRIndex performs well on insert and medium on MDRQ. Distinguished from these algorithms, LCIndex obtains high MDRQ performance and maintains high insert performance at the cost of high storage space. LCIndex is suitable for workloads with strict performance requirements on both insert and MDRQ.

V. RELATED WORK

For particular workloads, spatial trees can obtain high MDRQ performance, UQE-Index [5] builds a k-d tree for coarse-grained indexes and a set of fine-grained indexes via R-tree for local regions. MD-HBase [6] leverages linearization techniques such as Z-ordering to transform multi-dimensional data into single-dimensional space, and uses Quadtree and k-d tree to split the multi-dimensional space into subspaces.

On improving the performance of HBase, Hybrid HBase [17] enhanced the performance of HBase by using SSD for different components of HBase. Harter et. al presented a comprehensive study about the behavior of HDFS under HBase in a case of Facebook Message, the authors also used SSD to improve the performance of HBase [18]. In [19], Huang et. al extended HBase to leverage RDMA capable network and obtained high throughput.

Some literatures focus on indexing techniques for MDRQ on cloud system. Literature [20] proposed by Liao et al. built multi-dimensional indexes on HDFS based on R-tree. Dehne et al. introduced CR-OLAP [21], which can achieve real time response for OLAP based on distributed PDCR tree. HD Tree [22] is built over complete k-ary tree to support MDRQ for peer-to-peer networks. In [23], Wu et al. proposed a general indexing framework for peer-to-peer network, which consists of global index and local index. The global index is a subset of local index selected and published by some rules. Each node builds local indexes for data stored on it and maintains a subset of the global index. RT-CAN [24] is proposed to build multi-dimensional indexes, it leverages CAN to maintain the global indexes and builds local indexes via R-tree. CG-index [25] is proposed to execute range queries on one dimension, it builds local B+-tree indexes and organizes the nodes into a BATON network. However, maintaining peer-to-peer network is complex and introduces communication overhead, especially that master-slave systems have been well deployed in cloud systems. Thus it is natural to build a single global index on master and build a set of local indexes for each slave. Zhang et al. proposed EMINC [26], which builds local k-d tree, and further constructs the node as a cube in which contains information of the ranges of indexed dimensions. The cubes are organized on master nodes by R-tree for fast queries.

VI. CONCLUSION AND FUTURE WORK

Numerous techniques have been proposed to build indexes on Distributed Ordered Tables (e.g. HBase), aiming at improving the performance of flexible MDRQ. However, none of them can support high performance on both insert and flexible MDRQ.

In this paper, we propose a novel indexing technique called LCIndex, and implement its prototype on HBase. Experiments show that the insert throughput of LCIndex is 422% of CCIndex and 80% of IRIndex, and the MDRQ efficiency ratio between LCIndex and CCIndex is 43.4% to 54.6% and that between LCIndex and IRIndex is 183% to 407%. Besides, LCIndex features low network traffic during insert and query, and dynamic adding/removing index support. We plan to improve the IFile format to reduce the storage cost and accelerate scan operations, and apply LCIndex to Cassandra to improve its MDRQ performance.

ACKNOWLEDGMENT

This work is supported in part by the Hi-Tech Research and Development (863) Program of China (Grant No. 2013AA01A209, 2013AA01A213), the Strategic Priority Program of Chinese Academy of Sciences (Grant No. XDA06010401), the Guangdong Talents Program (Grant No. 201001D0104726115), NSF under NSF grants CNS-0751200, CCF-0937877, and CNS-1162540. The authors acknowledge financial support from China Scholarship Council.

REFERENCES

[1] 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 Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[2] 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," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[3] "The Apache HBase Project," http://hbase.apache.org.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[5] Y. Ma, J. Rao, W. Hu, X. Meng, X. Han, Y. Zhang, Y. Chai, and C. Liu, "An efficient index for massive iot data in cloud environment," in *Proceedings of the 21st ACM international conference on Information and Knowledge Management (CIKM'12)*, 2012, pp. 2129–2133.

[6] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: a scalable multi-dimensional data infrastructure for location aware services," in *Proceedings of the 12th IEEE International Conference on Mobile Data Management (MDM'11)*, vol. 1, 2011, pp. 7–16.

[7] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, "R-trees have grown everywhere," Technical Report available at http://www.rtreeportal.org, Tech. Rep., 2003.

[8] L. George, *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[9] "Indexed HBase," https://github.com/ykulbak/ihbase.

[10] "Indexed Transactional HBase," https://github.com/hbase-trx/hbase-transactional-tableindexed.

[11] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for vlsd databases," in *Proceedings of the 15th ACM International Conference on Management of Data (SIGMOD'09)*, 2009, pp. 179–192.

[12] Y. Zou, J. Liu, S. Wang, L. Zha, and Z. Xu, "Ccindex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries," in *Proceedings of the 7th IFIP International Conference on Network and Parallel Computing (NPC'10)*, 2010, pp. 247–261.

[13] "Hindex: Secondary indexes for faster HBase queries," https://github.com/Huawei-Hadoop/hindex.

[14] "Inside region secondary index for HBase," https://github.com/wanhao/IRIndex.

[15] A. Silberschatz, H. F. Korth, S. Sudarshan *et al.*, *Database system concepts*. McGraw-Hill Hightstown, 1997, vol. 4.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, 2010, pp. 143–154.

[17] A. Awasthi, A. Nandini, A. Bhattacharya, and P. Sehgal, "Hybrid hbase: Leveraging flash ssds to improve cost per throughput of hbase," in *Proceedings of the 18th ACM International Conference on Management of Data (SIGMOD'12)*, 2012, pp. 68–79.

[18] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, 2014, pp. 199–212.

[19] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda, "High-performance design of hbase with rdma over infiniband," in *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS'12)*, 2012, pp. 774–785.

[20] H. Liao, J. Han, and J. Fang, "Multi-dimensional index on hadoop distributed file system," in *Proceedings of the 5th IEEE International Conference on Networking, Architecture and Storage (NAS'10)*, 2010, pp. 240–249.

[21] F. Dehne, Q. Kong, A. Rau-Chaplin, H. Zaboli, and R. Zhou, "A distributed tree data structure for real-time olap on cloud architectures," in *Proceedings of the IEEE International Conference on Big Data (BigData'13)*, 2013, pp. 499–505.

[22] Y. Gu and A. Boukerche, "Hd tree: A novel data structure to support multi-dimensional range query for p2p networks," *Journal of Parallel and Distributed Computing*, vol. 71, no. 8, pp. 1111–1124, 2011.

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

[24] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proceedings of the 16th ACM International Conference on Management of Data (SIGMOD'10)*, 2010, pp. 591–602.

[25] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient b-tree based indexing for cloud data processing," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1207–1218, 2010.

[26] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *Proceedings of the 1st ACM International Workshop on Cloud Data Management (CloudDB'09)*, 2009, pp. 17–24.