Feng C, Li CD, Li R. Indexing techniques of distributed ordered tables: A survey and analysis. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 169–189 Jan. 2018. DOI 10.1007/s11390-018-1813-8

Indexing Techniques of Distributed Ordered Tables: A Survey and Analysis

Chen Feng[1,2], *Student Member*, *CCF*, Chun-Dian Li[1,2], *Student Member*, *CCF*, and Rui Li[3]

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Tencent Inc., Beijing 100080, China*

E-mail: {fengchen, lichundian}@ict.ac.cn; rli@tencent.com

Received March 27, 2017; revised May 12, 2017.

Abstract Many NoSQL (Not Only SQL) databases were proposed to store and query on a huge amount of data. Some of them like BigTable, PNUTS, and HBase, can be modeled as distributed ordered tables (DOTs). Many additional indexing techniques have been presented to support queries on non-key columns for DOTs. However, there was no comprehensive analysis or comparison of these techniques, which brings troubles to users in selecting or proposing a proper indexing technique for a certain workload. This paper proposes a taxonomy based on six indexing issues to classify indexing techniques on DOTs and provides a comprehensive review of the state-of-the-art techniques. Based on the taxonomy, we propose a performance model named QSModel to estimate the query time and storage cost of these techniques and run experiments on a practical workload from Tencent to evaluate this model. The results show that the maximum error rates of the query time and storage cost are 24.2% and 9.8%, respectively. Furthermore, we propose IndexComparator, an open source project that implements representative indexing techniques. Therefore, users can select the best-fit indexing technique based on both theoretical analysis and practical experiments.

Keywords database, Not Only SQL (NoSQL), range query, indexing

1 Introduction

Not Only SQL (NoSQL) databases have been widely used in industries, such as Google BigTable[1], HBase[①], MongoDB[②], Cassandra[2] and Redis[③]. Compared with traditional relational database systems (RDBMS), NoSQL databases usually provide high performance, high storage capacity, and high scalability. Distributed ordered table (DOT) is a kind of NoSQL databases. Distributed ordered tables (DOTs) like Google BigTable, Yahoo! PNUTS[3] and HBase have been used in many companies like Google, Facebook[4] and Yahoo![④]. DOTs horizontally partition a table into non-overlapping regions (in this paper we use the terminology of HBase because most of the existing indexing techniques are based on it) and manage regions via a B$^+$ tree whose every leaf node is a region. Regions are distributed on region servers, which provides the scalability to thousands of nodes. Each region contains records whose rowkeys are in a certain range. The size of a region is limited by a threshold, and regions exceeding the threshold will be split. DOTs support quick range scan on rowkey by locating regions covering the target rows and scanning these regions.

Survey

This work is partially supported by the Strategic Priority Program of Chinese Academy of Sciences under Grant No. XDB02040009, the Key Program of the National Natural Science Foundation of China under Grant No. 61532016, the Key Program of Cloud Computing and Big Data of the Ministry of the Science and Technology of China under Grant No. 2016YFB1000200, and Tencent Inc.

[①] http://hbase.apache.org, Nov. 2017.

[②] https://www.mongodb.org, Nov. 2017.

[③] https://redis.io, Nov. 2017.

[④] http://hbase.apache.org/www.hbasecon.com/, Nov. 2017.

However, sometimes users want to query records based on the values of non-key columns. In these cases, DOTs have to process a full-scan with filters on target columns, consuming lots of time and resource, which is unacceptable. Numerous indexing techniques have been proposed in research literature and industrial projects to improve the query performance on non-key columns. Some of them implement middlewares without touching the source code, such as CMIndex[5] and HIndex[5]. IHBase[6] and IRIndex[7] aim at minimizing interference in building indexes. LCIndex[6] provides high MDRQ (multi-dimensional range query) performance with low insert latency. The goal of MD-HBase[7] is acquiring high MDRQ performance on spatial data. UQE-Index[8] builds hybrid three-layer indexes to suit with time-related IoT (Internet of Things) workloads. Apache Phoenix[8] converts SQL queries to HBase operations and supports indexes as well.

Among various indexing techniques mentioned above, questions arise like what the best indexing technique for a given workload is, and what design issues have to be taken into consideration. To address these questions, in this paper, we provide a comprehensive taxonomy of indexing techniques on DOTs that captures design issues. IRIndex was firstly compared with HIndex, CCIndex[9] and CMIndex through an experiment. Zhang *et al.* presented HST[10] and compared HST with MD-HBase, KR$^+$-index[11] and EDMI[12]. Feng *et al.*[6] simply divided the indexing techniques into spatial tree indexes and single dimensional indexes, but the experiments involved no spatial tree indexes.

In this paper, we present a complete taxonomy of indexing techniques by six indexing issues and give a quantitative analysis on different combinations of the issues. Furthermore, experiments on representative indexing techniques are processed on a real workload to test their performance. This paper has three contributions.

- A taxonomy is proposed to classify indexing issues during building and maintaining indexes.
- A performance model is given to show the query time and storage cost of representative indexing techniques.
- An open source project named IndexComparator containing representative indexing techniques implementations is presented. Users can run experiments based on it to select the proper prototype.

The rest of the paper is organized as follows. Section 2 gives the taxonomy with six indexing issues to classify indexing techniques. Section 3 shows the comparison of existing techniques. Section 4 gives the performance model. Section 5 presents IndexComparator and runs experiments based on it to evaluate the performance model. Section 6 concludes this paper.

2 Taxonomy

About building and maintaining indexes on DOTs, many issues have been considered in literature and projects. By summarizing and comparing these issues, we select the most common and important issues to profile the indexing techniques. Firstly, issues of common indexing techniques like the mapping between indexed columns and the rowkey of index tables, index structure, and index granularity are adopted. Secondly, since DOTs are distributed systems and LSM-tree[13] is a common key structure of DOTs, issues like index placement, time to build indexes, and index persistence are considered as well.

2.1 Mapping Between Indexed Columns and the Rowkey of Index Table

Index data can be stored by the table of DOTs or other data structures. Each data structure must have a unique identifier to identify the index data. For tables of DOTs, the identifier is the rowkey; for R-tree[14], it is the ranges of MBR (minimum bounding rectangle). The unique identifier is called rowkey in this paper because most indexing techniques use tables of DOTs to store indexes. Similarly, the data structures storing index data are called index tables.

The tables of DOTs are single dimensional. DOTs identify records by the rowkey for insert and query operations. The rowkey of an index table may contain one or more indexed columns. Based on the mapping between indexed columns (source) and the rowkey of index table (destination), there are three kinds of mappings: single column to single dimensional, multi-column to multi-dimensional, and multi-column to single dimensional. These mappings determine how the indexes are built and how queries are processed.

[5] https://github.com/Huawei-Hadoop/hindex, Nov. 2017.

[6] https://github.com/ykulbak/ihbase, Nov. 2017.

[7] https://github.com/wanhao/IRIndex, Nov. 2017.

[8] http://phoenix.apache.org, Nov. 2017.

2.1.1 *Single Column to Single Dimensional*

A single column to single dimensional (S2S) index builds one index table for each indexed column and usually stores index data in tables of DOTs. The rowkey of an index table is usually concatenated by the value of indexed column and the raw rowkey. The S2S index is simple to implement, and its drawback is that only one indexed column can be filtered by the rowkey of an index table.

2.1.2 *Multi-Column to Multi-Dimensional*

Spatial data structures like R-tree and R$^+$-tree[15] have been widely used in spatial databases to store and query geometric objects. These structures map multi-dimensional data into multi-dimensional spaces. Multi-column to multi-dimensional (M2M) indexes use spatial data structures to map multiple indexed columns to the rowkey. The M2M indexes can filter multiple indexed columns at the same time, which usually leads to high MDRQ performance. However, their drawbacks are clear. Firstly, there is no default spatial data structure in DOTs, which increases the complexity of implementation, especially when the index data is large enough and has to be stored on disk. The underlying file system of DOTs like HDFS is append-only, making it very difficult in updating the spatial data structure. Secondly, the spatial data structures usually trade off between the performance of insert and query. A simpler spatial index like R-tree leads to high insert performance, but the query performance will be decreased due to overlaps of MBRs. In contrast, nonoverlapping structures like R$^+$-tree perform well in query but cost a lot in insert.

2.1.3 *Multi-Column to Single Dimensional*

A multi-column to single dimensional (M2S) index maps data of multiple indexed columns into the rowkey of an index table. The most common mapping function used is the space filling curve like Z-order curve and Hilbert curve. The advantages of M2S indexes are that they can use the table structure of DOTs to store index data and can filter multiple indexed columns at the same time. However, using the curves alone can result in false positive sub-queries. Fig.1 gives an example of false positive sub-queries of M2S indexes. In this figure, query conditions "$X \in [00_2, 01_2]$ and $Y = 11_2$" will be turned into query "subspaces $\in [0101_2, 0111_2]$". Therefore, three subspaces $0101_2, 0110_2$ and 0111_2 will be queried, as shaded in this figure. Obviously, subspace 0110_2 in the dashed circle is false positive and should not be searched.

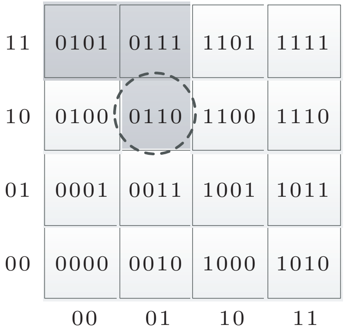
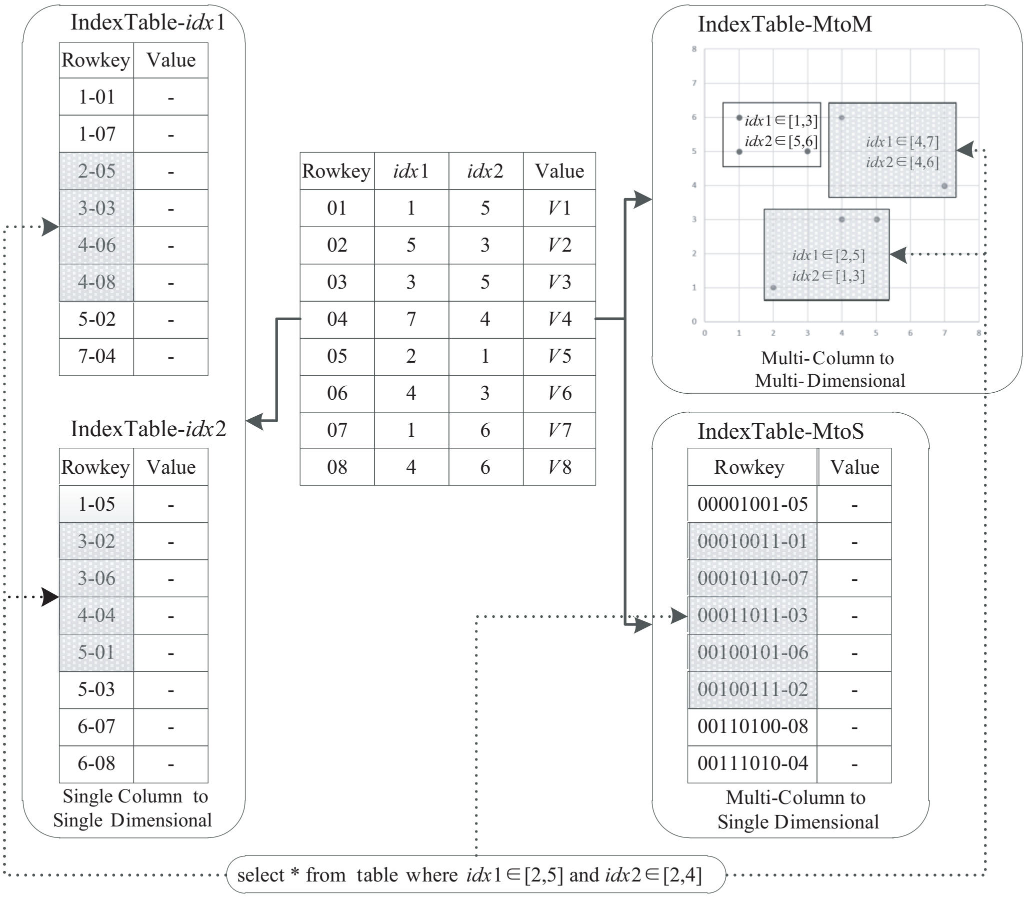


Fig.1. False positive using Z-order curve.

Introducing additional data structures like Quad tree or k-d tree can quickly prune these false positives, as presented in MD-HBase. Nevertheless, maintaining these structures increases the complexity. By the way, techniques concatenating values of indexed columns by order can also be seen as M2S indexes. However, the query performance on such indexes decreases dramatically when the first indexed column is not included in the query conditions because the query will be degraded into a full scan on the whole index table. Therefore, we neglect such a concatenating method in this paper.

Mapping single indexed column to multi-dimensional is possible but useless since DOTs support operations on single dimensional rowkey; therefore it is ignored in this paper.

Fig.2 shows examples of different kinds of these indexes. The raw table has two indexed columns $idx1$ and $idx2$, and the example records are shown in the middle of the figure. The S2S index built by concatenating is shown on the left side, and it has one index table for each indexed column. Both the M2M index and the M2S index have one index table for all indexed columns. The dotted arrows show the procedures of query processing of these indexes, and index data searched in querying "$idx1 \in [2, 5]$ and $idx2 \in [2, 4]$" is marked by shades. The S2S index must scan two index tables because each of them can only filter one indexed column. The M2M index is built based on R-tree and shown on the right top, and the query is turned to sub-queries on MBRs intersecting the query conditions. The M2S index is based on Z-order curve and shown on the right bottom, and the query will be firstly transformed into single dimensional range $[00001100_2, 00110001_2]$ and then executed on the index table.


IndexTable-*idx*1
IndexTable-*idx*2
Single Column to Single Dimensional
IndexTable-MtoM
*idx*1∈[1,3] *idx*2∈[5,6]
*idx*1∈[4,7] *idx*2∈[4,6]
*idx*1∈[2,5] *idx*2∈[1,3]
0 1 2 3 4 5 6 7 8
Multi-Column to Multi-Dimensional
IndexTable-MtoS
Multi-Column to Single Dimensional
select * from table where *idx*1∈[2,5] and *idx*2∈[2,4]

IndexTable-*idx*1

Rowkey	Value
1-01	-
1-07	-
2-05	-
3-03	-
4-06	-
4-08	-
5-02	-
7-04	-

IndexTable-*idx*2

Rowkey	Value
1-05	-
3-02	-
3-06	-
4-04	-
5-01	-
5-03	-
6-07	-
6-08	-

Rowkey	*idx*1	*idx*2	Value
01	1	5	*V*1
02	5	3	*V*2
03	3	5	*V*3
04	7	4	*V*4
05	2	1	*V*5
06	4	3	*V*6
07	1	6	*V*7
08	4	6	*V*8

IndexTable-MtoS

Rowkey	Value
00001001-05	-
00010011-01	-
00010110-07	-
00011011-03	-
00100101-06	-
00100111-02	-
00110100-08	-
00111010-04	-

Fig.2. S2S indexes, M2M indexes and M2S indexes.

In summary, the S2S index is simple in implementation, but it can only filter one indexed column by the rowkey of an index table. The M2M index can filter multiple indexed columns at the same time, but it is difficult in implementation due to the lack of built-in data structures and the append-only underlying file system. The M2S index can make use of the default table structure of DOTs and filter multiple indexed columns at the same time, while its drawback is that it may result in false positive sub-queries or additional overhead in pruning such sub-queries.

2.2 Index Structure

According to Silberschatz *et al.*[16], the index structure can be secondary, or clustering, which is also applicative for indexing DOTs.

2.2.1 *Secondary*

Secondary indexes provide mapping from the values of indexed columns to the rowkey of the raw table. Queries will be processed by firstly finding rows satisfying all conditions (whose rowkeys are called the final rowkeys in this paper) and then querying the raw table based on the final rowkeys. For M2M and M2S indexes or single dimensional query, the final rowkeys can be queries from index tables directly. For MDRQ on S2S indexes, secondary indexes should scan every index table to get rows satisfying corresponding condition (whose rowkeys are called the candidate rowkeys in this paper), and the final rowkeys are the common ones of candidate rowkeys.

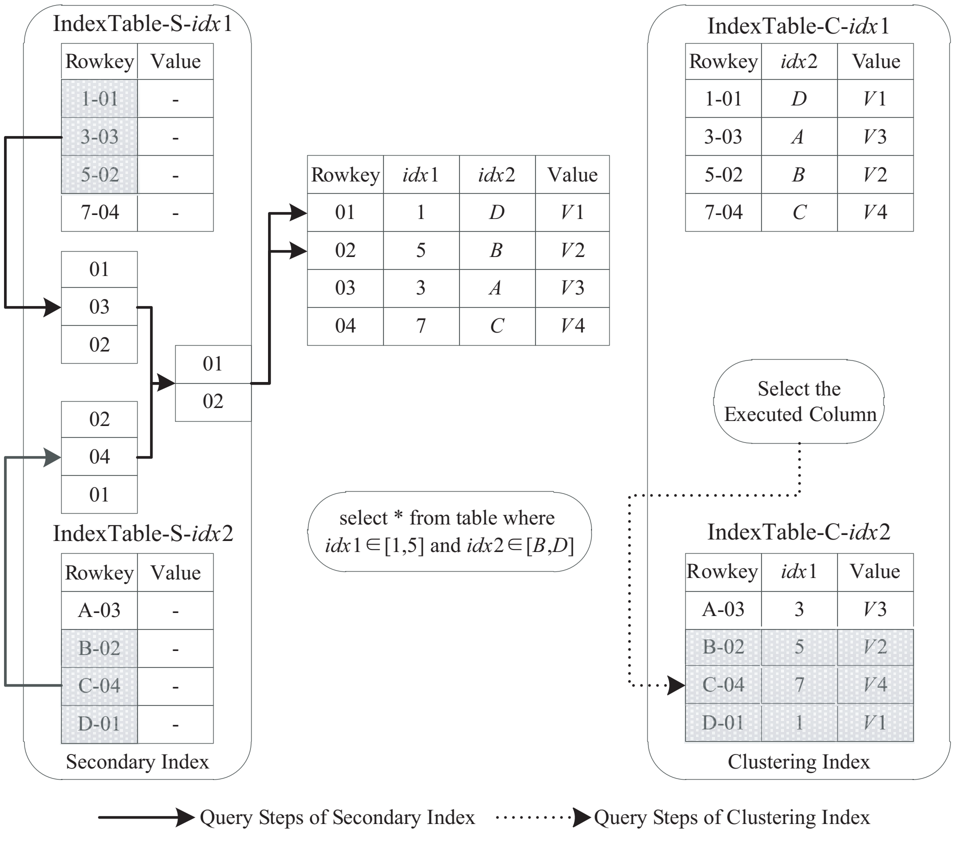
2.2.2 *Clustering*

Clustering indexes have the same rowkey as secondary indexes and store all target columns of queries; therefore results can be directly returned without querying the raw table. This paper considers that all columns are used in queries; hence a clustering index is a "copy" of the raw table. Therefore secondary indexes have higher insert performance and lower storage cost compared with clustering indexes. For MDRQ on S2S indexes, clustering indexes select one indexed column and scan corresponding index table using conditions on other columns as filters. In this case, the selected indexed column is called executed column and the index table is called executed table.

Fig.3 gives an example of a secondary index and a clustering index, where both indexes are S2S. The raw table has two indexed columns and its records are shown in the middle of the figure. For index tables on $idx1$, the secondary index only has the value of $idx1$ and the raw rowkey while the clustering index has all columns. The query procedures of the secondary index and the clustering index are shown in the solid line and the dashed line, respectively. The scanned data in index tables in the query is shaded. For secondary indexes, the candidate rowkeys are $01, 03, 02$ and $02, 04, 01$ from index tables of $idx1$ and $idx2$, respectively. The final rowkeys are $01, 02$. For clustering indexes, the query is turned to a scan on index table of $idx2$.

We can see that during querying, secondary indexes have to read more records compared with clustering indexes. Besides, querying the raw table based on the final rowkeys is usually processed by random gets (random get operations) on DOTs, which is much slower than the scan operation used in clustering indexes. Therefore the query performance of secondary indexes is slower than that of clustering indexes.

Secondary indexes have a simple procedure in updating or deleting a record. Examples are given based on values in Fig.3. When updating a non-indexed column, no operation is needed for secondary indexes, while the clustering indexes must update all indexed tables. For example, updating record $(01, 1, D, V1)$ to $(01, 1, D, VX)$ means clustering indexes must update the record with rowkey $(1 - 01)$ in index table for $idx1$ and the record with rowkey $(D - 01)$ in index table for idx2. When updating an indexed column, secondary indexes should delete the existing index record and insert a new one. Meanwhile, S2S and clustering indexes must update values in other index tables in addition. For example, when updating record $(01, 1, D, V1)$ to $(01, 1, X, V1)$, secondary indexes must delete the record with rowkey $(D - 01)$ and insert a new record $(X - 01)$ on the index table of $idx1$. The clustering indexes must delete the record with rowkey $(D - 01)$ and insert a new record $(X - 01, 1, V1)$ on the index table of $idx1$. Besides, the clustering indexes must update the record


IndexTable-S-idx1
01
03
02
02
04
01
01
02
IndexTable-S-idx2
Secondary Index
IndexTable-C-idx1
Select the Executed Column
IndexTable-C-idx2
Clustering Index
select * from table where idx1∈[1,5] and idx2∈[B,D]
Query Steps of Secondary Index
Query Steps of Clustering Index

Rowkey	Value
1-01	-
3-03	-
5-02	-
7-04	-

Rowkey	Value
A-03	-
B-02	-
C-04	-
D-01	-

Rowkey	idx1	idx2	Value
01	1	D	V1
02	5	B	V2
03	3	A	V3
04	7	C	V4

Rowkey	idx2	Value
1-01	D	V1
3-03	A	V3
5-02	B	V2
7-04	C	V4

Rowkey	idx1	Value
A-03	3	V3
B-02	5	V2
C-04	7	V4
D-01	1	V1

Fig.3. Secondary index and clustering index.

$(1-01, D, V1)$ to $(1-01, X, V1)$ in the index table of $idx2$. The procedures of deletion are similar to those of updating.

In summary, secondary indexes are simpler in updating and deletion, and they have higher insert performance, lower storage cost, and lower query performance. On the contrary, clustering indexes are complex in updating and deletion, and they have lower insert performance and higher storage cost.

2.3 Index Granularity

Most indexing techniques are fine-grained, namely, each raw record has its index records. They can locate the target records directly at the cost of high storage cost. On the contrary, coarse-grained indexes divide raw data into units (region, StoreFile or records within a range). They build indexes for each unit, which reduces storage cost. During query processing, if one record meets the conditions, the whole unit must be searched, incurring unnecessary queries.

Fig.4 presents an example of a fine-grained index and a coarse-grained index, and there is only one indexed column. Both of the two indexes are StoreFile level. The fine-grained index has one-to-one mapping between the raw record and the index record. The coarse-grained index divides the range of indexed column and gathers the number of records on each span of the range. For example, the line for StoreFile-A means that there are one record whose value is within $[1, 25]$, two within $[26, 50]$, three within $[51, 75]$ and no record within $[76, 100]$. When querying "$select * fromtablewhereidx1 \in [20, 50]$", the fine-grained index directly points to the rows within this range while the coarse-grained index has to scan the two StoreFiles for results. Coarse-grained indexes are mostly used in querying pre-defined aggregation data. For example, when querying "$count(*)where$idx1$\in [76, 100]$", coarse-grained indexes have tremendous performance while fine-grained indexes must get records one by one.

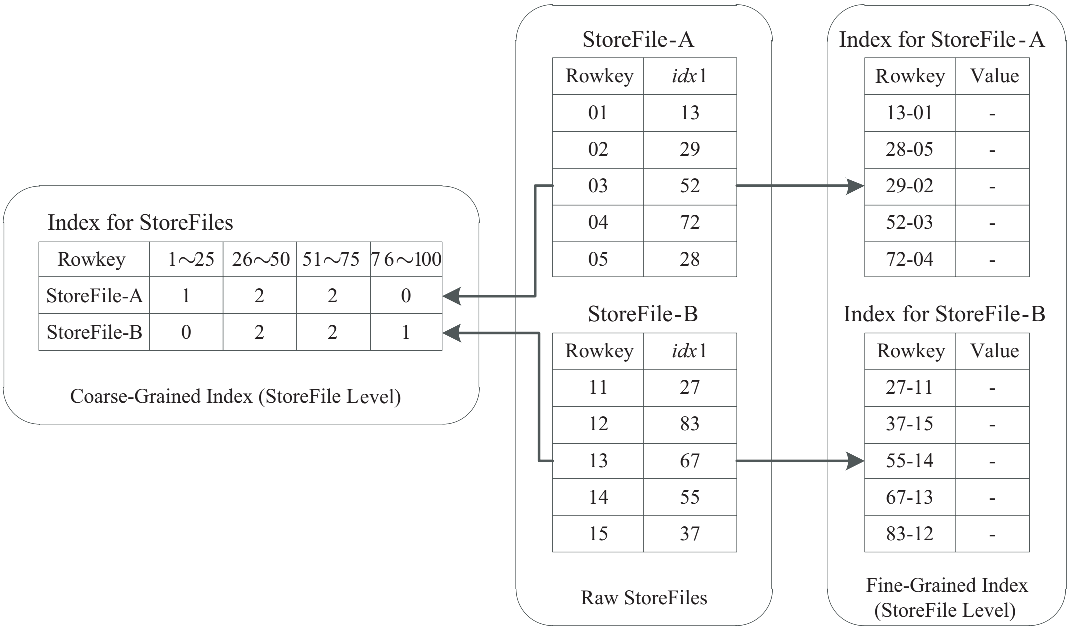
In summary, fine-grained indexes can locate records precisely and avoid unnecessary searches at the cost of high storage overhead. Coarse-grained indexes have lower storage cost and higher performance on pre-defined aggregation queries.

2.4 Index Placement

In DOTs, the operation of each record is distributed to the region managing it. According to the location of the region managing the raw record and the location holding its index record, an indexing technique can be either global or local.

2.4.1 *Global*

Many indexing techniques build index tables by utilizing APIs of DOTs directly and leave the responsibility of managing regions of index tables to DOTs, where the regions are automatically distributed over


Coarse-Grained Index (StoreFile Level)
Raw StoreFiles
Fine-Grained Index (StoreFile Level)

Index for StoreFiles

Rowkey	1∼25	26∼50	51∼75	76∼100
StoreFile-A	1	2	2	0
StoreFile-B	0	2	2	1

StoreFile-A

Rowkey	*idx*1
01	13
02	29
03	52
04	72
05	28

StoreFile-B

Rowkey	*idx*1
11	27
12	83
13	67
14	55
15	37

Index for StoreFile-A

Rowkey	Value
13-01	-
28-05	-
29-02	-
52-03	-
72-04	-

Index for StoreFile-B

Rowkey	Value
27-11	-
37-15	-
55-14	-
67-13	-
83-12	-

Fig.4. Fine-grained index and coarse-grained index.

all region servers. Therefore, index records and corresponding raw records can be stored on different region servers, which is called global. Global indexes are simple to implement for both insert and query, but they involve cross-node network traffic in building indexes and querying on secondary indexes.

2.4.2 *Local*

Local indexes co-locate index records with the raw records on the same region server. Therefore, the network traffic of local indexes is reduced in building indexes and eliminated in querying on secondary indexes. Two implicated premises for the effects of local indexes are as follows. 1) The underlying file system like HDFS tries to write and read data on the same node where the operations are proposed. 2) DataNodes of HDFS and region servers of DOTs are co-located. A disadvantage of local indexes is that queries must be split to sub-queries on every single region, which increases the complexity. However, from another point of view, the query performance can be increased by executing sub-queries in parallel across region servers.

Fig.5 presents an example of a global index and a local index. It is clear that index records of the local index are always on the same region server with the raw records. For the global index, record 03 is stored on region server 2 while its index record is on region server 1.

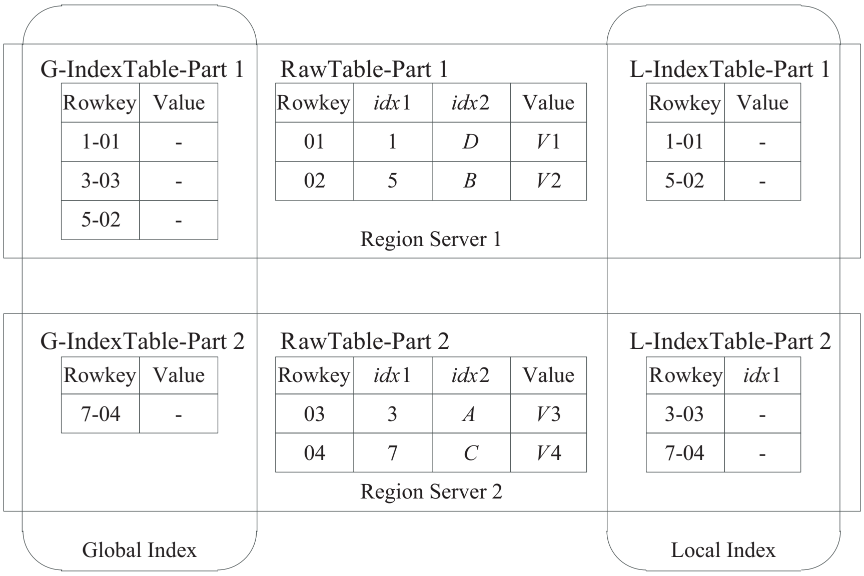
In summary, global indexes are simpler to implement; local indexes have lower network traffic and can be executed in parallel to improve the query performance, but they are more complex in implementation.

2.5 Time to Build Indexes

DOTs usually use LSM-tree on regions to reduce insert latency, namely, a record will be firstly inserted into the MemStore. The MemStore will be flushed to disk when meeting some conditions, like reaching the threshold. Therefore there are two kinds of indexes, on insert or on flush.

2.5.1 *On-Insert Index*

On insert indexes build indexes when records are inserted, and they can be on either client-side or server-side. Client-side means indexes are built when records are inserted into the raw table. It leaves the responsibility of building and maintaining indexes to users. On the other hand, server-side indicates indexes are built when records are inserted into a region server, which eliminates the network traffic of sending index records from the client to the server. Since indexes are ready along with raw records, queries can be processed straightforwardly once records are inserted. A considerable disadvantage of on insert indexes is that they involve index operations for each insert and decrease the insert performance. It is noteworthy that on insert indexes usually store index data on the default table structure of DOTs, in other words, the index data is flushed and synchronized to log files for data recovery, which increases the insert latency and disk operations.



G-IndexTable-Part 1

Rowkey	Value
1-01	-
3-03	-
5-02	-

RawTable-Part 1

Rowkey	*idx*1	*idx*2	Value
01	1	*D*	*V*1
02	5	*B*	*V*2

L-IndexTable-Part 1

Rowkey	Value
1-01	-
5-02	-

G-IndexTable-Part 2

Rowkey	Value
7-04	-

RawTable-Part 2

Rowkey	*idx*1	*idx*2	Value
03	3	*A*	*V*3
04	7	*C*	*V*4

L-IndexTable-Part 2

Rowkey	*idx*1
3-03	-
7-04	-

Fig.5. Example of global index and local index. G: global; L: local.

2.5.2 *On-Flush Index*

On flush indexes will not build indexes for in-memory records until they are flushed to the disk. Inserts return directly without building indexes, which significantly reduces the insert latency. However, a query must be processed on both indexed data on disk and non-indexed data in MemStore. The time used in querying the MemStore has an upper limit and can be negligible because the size of the MemStore is limited and scanning the MemStore in memory is much faster than that of the disk. Since code modification is inevitable to scan the MemStore during query processing, on flush indexes are naturally to be implemented on the server side. As a result, on flush indexes are usually set to local to save network traffic and improve flush performance. Local index tables of on flush indexes are either region level or StoreFile level. The region level means there is an index table for the whole region. The StoreFile level indicates each StoreFile has a corresponding IndexStoreFile. Such IndexStoreFile requires additional operation in maintaining index data when a StoreFile is created or deleted due to flush, merge or split. It takes a long time and costs lots of resources to build indexes for lots of records at the same time, which may decrease the overall performance when flushing. Building indexes asynchronously is helpful, but it involves more complex modification to handle the inconsistency between the raw data and the index data before indexes are flushed to disk.

Building indexes on bulk load is feasible[17] but is not discussed here because it usually builds indexes for StoreFiles, which is similar to on flush.

Fig.6 presents an example of an on insert index and an on flush index, where IndexBuilder parses raw record to records on index tables. The IndexBuilder of the on insert index can be either client-side or server-side, and it builds indexes for every record inserted. The IndexBuilder of the on flush index is always on the server side, and it is called when the MemStore is flushed.

In summary, on insert indexes are simpler in building indexes, maintaining indexes, and query processing, but the insert performance is lower; on flush indexes have higher insert performance, but are more complex in building indexes, maintaining indexes, and processing queries.

2.6 Index Persistence

As the name indicates, DOTs are distributed systems. No matter where the index data is stored, its reliability should always be considered carefully. There are four kinds of persistence for index data, namely

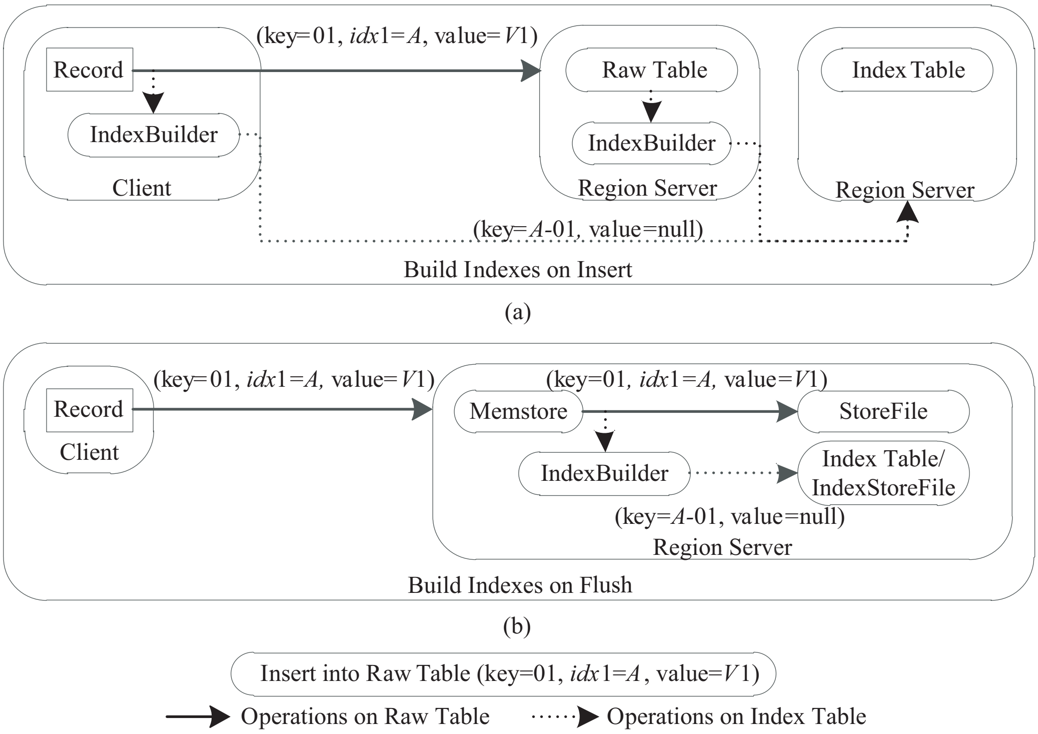


Fig.6. (a) On-insert index. (b) On-flush index.

persistent on disk, nonpersistent on disk, memory only, and memory as cache.

2.6.1 Persistent on Disk

It is not necessary for users to worry about reliability when the index data is persistent on disk because the underlying distributed file systems like GFS and HDFS naturally provide transparent persistence. The disadvantage of persistent on disk index is that the storage cost is high. Using erasure codes to save index data is also considerable, but it leaves the risk of lower query performance when recovering data.

2.6.2 Nonpersistent on Disk

To reduce the storage overhead, some techniques like CCIndex store only one copy of index data, which is called nonpersistent on disk in this paper. However, extra mechanisms must be involved to handle two challenges at data loss: how to recover index data and how to process queries with incomplete index data, which increases the complexity. No matter how the mechanisms are designed, performance degradation and resource cost are inevitable when index data loss occurs.

2.6.3 Memory Only

Memory only indexes store index in memory only for high performance of both insert and query. Indexes will be rebuilt when a region is migrated or the node is restarted. The memory will be easily exhausted even if there exists only one replica in memory. Using coarse-grained indexes instead of fine-grained indexes can reduce memory cost; however the query performance is reduced as well.

2.6.4 Memory as Cache

Memory as cache indexes adapt a hybrid strategy by saving index data persistent on disk and keep some of them in memory as cache to improve the query performance. They cost lots of memory and the query performance relies on a proper cache replacement algorithm for index data.

In summary, persistent on disk indexes are simple for implementation, but the storage cost is high; nonpersistent on disk indexes save storage cost but require additional mechanisms to handle data losses; memory only indexes have high performance in both insert and query but cost lots of memory and are complex in data recovery; memory as cache indexes have high query performance and are persistent for failures, but they are complex in implementation and cost lots of storage space.

2.7 Summary

The pros and cons of different indexing issues are summarized in Table 1.

3 Comparison of Existing Indexing Techniques

Based on the taxonomy presented in Section 2, we discuss indexing techniques from both academia and industry based on the order of their relationships and the time they were proposed, and analyze their appropriate workloads. A summary of these techniques is presented in Table 2, ordered by the indexing issues. Note the bold techniques (CMIndex, CCIndex, LCIndex and MD-HBase) are used in the model discussed in Section 4.

3.1 Single Column to Single Dimensional Indexes

CMIndex is the basic prototype of indexes, whose initial purpose is for rapid application with simple implementation complexity; thus building on insert and secondary indexes at the client side is the consequent choice. Since index data is managed and replicated by DOTs, CMIndex is global and persistent on disk. CMIndex has small storage cost, but its performance in insert and query is not high. ITHBase[9] can be regarded as an enhancement of CMIndex, and it supports transaction in building indexes. Inevitably, it increases the insert latency dramatically.

Building secondary indexes by the coprocessor at the server side is convenient to execute intrusive operations without modifying source code. HIndex is the first well known technique that builds indexes in this way. It is a secondary and local index by modifying the load balancer to co-located regions of index tables and corresponding regions of raw tables. HIndex has higher insert performance than CMIndex and is more complex in implementation, and it is suitable for workloads with some constraints on the performance of insert and query. HiBase[18] pays more attention to the query performance. It builds memory as cache indexes on HBase, and stores permanent indexes on HBase tables and in-memory indexes based on distributed consistency hash.

[9] https://github.com/hbase-trx/hbase-transactional-tableindexed, Nov. 2017.

Table 1. Summary of Pros and Cons of Different Indexing Issues

Issue	Item	Pros	Cons
Mapping	S2S	• Can take advantage of the default structure of DOTs to store index data	• High storage overhead • Only one column is filtered by the rowkey
	M2M	• Filter multiple columns at the same time	• Require additional spatial data structure
	M2S	• Filter multiple columns at the same time • Use table of DOTs to store index data	• Meet challenges in trading off between the performance of insert and query
Index structure	Secondary	• Low storage overhead • Simple procedure in update and deletion • High insert performance	• Low query performance due to random get operation on raw table
	Clustering	• High query performance	• High storage overhead • Complex procedure in update and deletion • Low insert performance
Granularity	Fine-grained	• Locate data precisely, no unnecessary searches	• High storage cost
	Coarse-grained	• Low storage cost • High performance for pre-defined aggregation queries	• Low performance of undefined queries
Placement	Global	• Take advantage of the default structure of DOTs	• Additional cross-node communication in insert and query
	Local	• Low network traffic • Executing sub-queries in parallel can improve the query performance	• Additional mechanism is needed to place index data • Costly for highly selective queries
Time to build indexes	On insert	• Simple in building indexes, maintaining indexes, and processing queries	• Low insert performance
	On flush	• High insert performance	• Complex in building indexes, maintaining indexes, and processing queries
Persistence	Persistent on disk	• Simple in implementation	• High storage cost
	Nonpersistent on disk	• Low storage cost	• Performance degradation at failure • Additional mechanism to recovery indexes and process query at failure
	Memory only	• High performance in both insert and query	• Cost lots of memory • Consume lots of time in rebuilding indexes from disk at failure
	Memory as cache	• High query performance	• High storage cost and memory cost • Complex in implementation

To leverage the limited memory capacity, HiBase proposes a new hotscore algorithm cache replacement policy. HSQL[21] builds indexes inside a region using distributed B-tree and executes sub-queries in parallel by coprocessors to improve the query performance.

Both IHBase and IRIndex build secondary indexes on flush and their insert performance is high. IHBase builds indexes for the entire region and updates indexes in memory; therefore indexes must be rebuilt when a region is opened for the first time. Since recovering indexes consumes lots of resource and time, IHBase only works for workloads with limited data size. IRIndex builds indexes for each StoreFile and stores index data with only one replica to save storage cost. If IRIndex finds index data missing during query processing, the query will be degraded to a full scan on corresponding region. IRIndex is suitable for workloads requiring high insert performance and decent query performance.

Compared with secondary indexes, clustering indexes have higher MDRQ performance but are less attractive mainly because of the high storage cost. CCIndex is a representative of clustering indexing technique. To reduce storage overhead, CCIndex views the raw table and all index tables as complemental clustering index tables (CCITs), and sets the replica factor of CCITs to 1. CCIndex introduces complemental check tables (CCTs) to recover data via a complex mechanism. However, maintaining CCTs decreases the insert performance of CCIndex. CCIndex counts the number of regions covered by each indexed column and then selects the smallest one to execute the query. CGDM[19] is similar to CCIndex but specialized for molecular profiling workload. CGDM contains three index tables, CGCIT1, CGCIT2 and CGCIT3. It stores the index tables with one replica and discards the check tables. This is because CGCIT1 and CGCIT2 have the same prefix and can be recovered by each other. CGCIT3 can be recovered by scanning CGCIT2 or CGCIT1. Both CCIndex and CGDM can provide high query performance but require huge storage space. SQL-to-HBase[20] is

Table 2. Analysis and Comparison of Existing Indexing Techniques

Technique	Index Mapping	Index Structure	Index Granularity	Index Placement	Index Build Time	Index Persistence
CMIndex[5], ITHBase⑩	S2S	Secondary	Record	Global	On insert	PD
HiBase[18]	S2S	Secondary	Record	Global	On insert	MC
HIndex⑪	S2S	Secondary	Record	Local	On insert	PD
UQE-Index current[8]	S2S	Secondary	Time interval	Global	On insert	N/A
IHBase⑫	S2S	Secondary	Record	Local	On flush	MO
IRIndex⑬	S2S	Secondary	Record	Local	On flush	NP
CCIndex[9], CGDM[19]	S2S	Clustering	Record	Global	On insert	NP
SQL-to-HBase[20]	S2S	Clustering	Record	Global	On insert	N/A
LCIndex[6]	S2S	Clustering	Record	Local	On flush	NP
HSQL[21]	S2S	N/A	Local	On insert	N/A	N/A
Async Views LVT[22]	S2S	N/A	Region	Local	On insert	PD
Async Views RVT[22]	S2S	Either	Record	Global	On insert	PD
Phoenix⑭	S2S	Either	Record	Either	On insert	PD
UQE-Index history[8]	M2M	Secondary	Record	Local	On flush	PD
ABR-Tree[23]	M2M	N/A	Record	Local	On insert	PD
MD-HBase[7], SK-HBase[24]	M2S	Secondary	Record	Global	On insert	PD
HBaseSpatial[25]	M2S	Secondary	Grid	Global	On insert	PD
HST[10]	M2S	Secondary	Region	Global	On insert	PD
HGrid[26]	M2S	Secondary	Grid	Global	N/A	PD
HDKV[27]	M2S	Clustering	Record	Global	On insert	PD
Index for spatial	M2S	Secondary	Coarse-grained	Global	N/A	N/A

Note: PD: persistent on disk; NP: nonpersistent on disk; MO: memory only; MD: memory as cache; index for spatial: a series of indexing techniques for multi-dimensional rowkey.

another study of clustering index on DOTs. But its persistence is not mentioned in the literature. In its experiment, HBase is not replicated, but no additional mechanism to reduce storage overhead or recovery data is introduced. LCIndex is short for local and clustering index. It builds StoreFile level index on flush, and index data is stored on local disk with one replica. LCIndex runs background jobs to build indexes to reduce overall performance interference. A query will be degraded to a full scan when corresponding index files are missing. Moreover, the way CCIndex selects the index table is based on the characteristics of global index, which is not suitable for LCIndex. Thus, LCIndex requires users to describe the value ranges and the number of spans of indexed columns. Based on the description, LCIndex creates statistic files that count the number of index records on different spans. LCIndex is suitable for workloads requiring high performance on both insert and query, but has low requirement on storage space.

Asynchronous views[22] contain two kinds of indexes, remote view tables (RVT) and local view tables (LVT). RVT uses PNUTS table for record level index and therefore it is global and persistent on disk. RVT is used for record level queries, and it can be secondary or clustering depending on the configuration. LVT is a region level index and co-located with raw region, and it is used for queries on aggregation information. Both RVT and LVT build indexes on insert, and the difference is that the former is maintained via an asynchronous log manager, while the latter stores aggregation data synchronously. Apache Phoenix builds indexes on insert and supports both secondary and clustering indexes like RVT. Local indexes of Phoenix are stored in a shadow column family of the raw table, which is significantly different from other techniques⑮.

⑩ https://github.com/hbase-trx/hbase-transactional-tableindexed, Nov. 2017.

⑪ https://github.com/Huawei-Hadoop/hindex, Nov. 2017.

⑫ https://github.com/ykulbak/ihbase, Nov. 2017.

⑬ https://github.com/wanhao/IRIndex, Nov. 2017.

⑭ http://phoenix.apache.org, Nov. 2017.

⑮https://issues.apache.org/jira/browse/PHOENIX-1734, Nov. 2017.

3.2 Multi-Column to Multi-Dimensional Indexes

In many workloads, records of DOTs will be updated after they are inserted, which leads to update for index data. For R-tree and its derived spatial data structures, continuously updating decreases the insert performance. However, in some workloads, records are identified by increasing time value and historical data will never change, which makes it possible to build M2M indexes for DOTs.

UQE-Index is a combination index highly customized for Internet of Things. UQE-Index divides time dimension into time intervals and builds three layers of indexes based on it. In the first layer, a coarse-grained B$^+$-tree index is built based on the time intervals. The second layer is a k-d tree index that divides data inside current time interval into subspaces. Once a time interval ends, corresponding data is used to build the third layer index via background jobs. The index is local and secondary and is fine-grained in R-tree. The first two layers are used for querying current data, and they are coarse-grained and built on insert for low insert latency. The first and the third layers are used for querying historical data, where R-tree based local indexes lead to high query performance. ABR-Tree[23] firstly builds a global AB-Tree (Append-efficient B-Tree) based on time intervals, each of which is mapped to a local R*-tree based on other indexed columns. Both AB-Tree and all R*-trees are stored in different HBase tables. Queries search AB-Tree for candidate intervals and then search local R*-trees for final results.

3.3 Multi-Column to Single Dimensional Indexes

In many temporal cases, time is always included in query constraints; therefore many specific indexing techniques emerge.

HBaseSpatial[25] is a trivial M2S index by concatenating indexed columns. It divides the space by levels of grids and stores grid level index in HBase tables. A record in the index table is mapped to a grid. Its rowkey is concatenated by the longitude and latitude of the grid and the grid levels and its values are the rowkeys of rows in the raw table within the grid. During queries, HBaseSpatial traverses possible grids from the index table to get candidate rowkeys, and then get results from the raw table.

Different from HBaseSpatial, a series of indexing techniques on spatial data apply space filling curve to divide the space into subspaces and map the coordinates to single dimension as (part of) the rowkey for index tables. They are usually secondary and global indexes. For record level index, queries firstly get candidate subspaces (ranges of rowkey or regions), then scan subspaces for candidate rowkeys, and finally query the raw table for final results. For coarse-grained index like region or grid level, the raw table is directly scanned because each subspace is directly mapped to a region or a grid rather than rows of the raw table. Another main difference between these indexes is how to prune false positive subspaces to improve the query performance. As the representative of these indexing techniques, MD-HBase makes use of k-d tree or Quad tree to prune false positive subspaces of Z-order curve. HGrid[26] uses Z-order curve and manages the space by Quad tree whose leaf node is a grid. In HST, the space is partitioned into grids to apply Hilbert curve, and the meta structure is divided into two segments, spatial segment (SS) and temporal segment (TS). SS stores mapping from the Hilbert value of a grid to entries in TS. The time in TS is marked by the modulus on a period T and mapped to a region ID holding corresponding data. SK-HBase[24] is used for queries on spatial keyword, and it maintains an index table whose rowkeys are concatenated by the term and the Z-order value, and uses k-d tree to reduce false positive. As an extension of SK-HBase, ST-HBase[28] proposes term cluster based inverted spatial index (TCbISI) based on the observation that some terms are often used together in the same query. By involving term cluster index for terms, TCbISI can improve the query performance on several terms. HDKV[27] firstly maps objects to buckets by local sensitive hashing (LSH), and then converts the buckets to single dimension ranges by an order-preserving hash function. HDKV replicates the raw table based on the order of hash tables in LSH for better query performance.

3.4 Indexing Techniques for Multi-Dimensional Rowkey

Some techniques directly use the values of space filling curve as the rowkey of the raw table, and build secondary, M2M, global, coarse-grained index to convert geographic coordinates based queries to sub-queries of scans over rowkey ranges. Strictly, they can be reduced to “indexing techniques for rowkey”, which does not match our scope. However, they can be easily adapted to “index for non-key columns” by storing their “raw table” as index tables for raw data, similar to MD-HBase.

Hence, we summarize these techniques and call them as "index for spatial" and fill them in the same line of Table 2. In detail, the main differences between them are 1) the data structures used to build the M2M indexes, and 2) the persistence of the index data.

Tang *et al.*[29] divided the space into grids and each subspace is further identified by pyramids and heights. Ma *et al.*[30] built a global region split tree to manage grids. The index data can be stored in either memory or HBase table. The local region level R-tree index is stored as a file on HDFS with additional statistic information involved for query processing. In EDMI[12], the space is partitioned by a global k-d tree whose leaf node maps to a Z-order prefix R-tree (ZPR-tree) built by MapReduce. Wang *et al.*[17] used grids for Hilbert values. The local R-trees are built by MapReduce jobs and used as subtrees of the global R-tree. STEHIX[31] saves the mapping from Hilbert curve values to regions and builds region level indexes. In each region, two kinds of indexes, s-index and t-index, are built on flush and saved in memory only. Values of these indexes are the addresses of key-value data in StoreFiles. The s-index is used for spatial data and the space in it is further partitioned by Quad tree. The t-index is an array based on a period T, and is used for temporal data. Geohash is used to generate the rowkey of index table[32]. The longer the geohash code is, the finer the rectangles will be, and hence more index records will be stored. TNBGR[33] builds a multiple layer grid tree where each cell in the tree is mapped to a region. Queries will be translated to MapReduce jobs for the final results where each candidate region in the query involves a map task. KR$^+$-index builds secondary index on insert. An R$^+$-tree and a grid are both used to divide the space separately. The R$^+$-tree is stored in an index table, whose rowkey is the Hilbert value of the rectangle and the values are records belonging to this rectangle. KR$^+$-index also stores the mapping from the grid to the Hilbert values of rectangles overlapping the grid. R-HBase[34] divides the space by grids, manages the grids by a global R-tree, and uses the code of space filling curve as the rowkey. Li *et al.*[35] adopted Hilbert curve and ran a MapReduce job on each candidate region for final results. Pyro[36] uses Moore encoding for linearization and optimizes multi-scan for better performance. Du *et al.*[37] built R-tree based on Hilbert curve. Historical statistics is used to determine the distribution and size of MBR.

3.5 Potential Combinations of Indexing Issues

Table 2 presents the classifications of known indexing techniques. By viewing the pros and cons shown in Table 1, we get some potential combinations of indexing issues.

First of all, the proposition of LCIndex is inspired by the classifications of indexing issues. In [6], the authors found the potential value of building local clustering indexes on flush, namely gaining high insert performance and MDRQ performance at the same time. The prototype and experiments confirm the value of LCIndex: its insert throughput is 72.5% compared with HBase without indexes and is 4.2x compared with CCIndex, and its MDRQ throughput is 43.6%~54.6% of CCIndex, 1.83x~4.07x of IRIndex, and 3.15x~7.46x of HBase without indexes.

In this paper, we notice that LMSIndex, namely, the M2S, secondary, local, on flush, find-grained index, has not been proposed. Based on the theoretical analysis, LMSIndex can achieve high insert performance and low storage overhead easily. The query performance of LMSIndex may be lower than that of clustering indexes, but it is considered to be higher than S2S secondary indexes like CMIndex or IRIndex. Since LMSIndex is M2S and secondary, its storage overhead will be lower than any other S2S indexes (for more than two indexed columns) or clustering indexes; therefore LMSIndex can directly store persistent on disk indexes without worrying about the storage cost.

Considering the detail implementations of local indexes like HIndex, IRIndex, and LCIndex, we find that the maintenance of StoreFile level index is complex and costly, especially when a compaction generates a new large StoreFile. Therefore following the way HIndex used to co-locate regions of index tables, IRIndex and LCIndex can build index records and bulk load the new built index StoreFiles into local regions of index tables rather than write local index files manually. In this way, minor compactions on StoreFiles of the raw table can be processed without considering indexes because the index regions will compact the StoreFiles of the index tables appropriately.

After all, Table 1 and Table 2 also indicate the key issues when selecting or designing an indexing technique under some resource or performance constraints. For example, for a workload not caring about the insert performance, to achieve the best query performance under limited storage cost, users should firstly consider the storage costs of different kinds of clustering indexes.

CCIndex and LCIndex persistent on disk should be the first options, and then are the erasure coded CCIndex and LCIndex. If no clustering index can meet the storage constraint, then users should start considering the secondary indexes.

Another trivial observation is that no existing indexing technique can acquire high insert performance, high query performance and low storage overhead at the same time for general workloads. By analyzing the issues, we find that the insert performance is independent with the other two metrics and the root of the problem is the trade-off between the query performance and the storage overhead. A trivial explanation is that the high query performance requires getting the final results with few disk reads. Such demands usually rely on new designed data layout, which requires lots of storage overhead.

4 QSModel: Modeling the Query Time and Storage Cost for Indexing Techniques of DOTs

When applying an indexing technique to the production environment, it is general to execute a sampled workload on a test environment. However, it is difficult to use the test results directly because the hardware and the query conditions are not exactly same between the test and the production environment.

4.1 Concerns of QSModel

In Section 4, we want to propose a performance model for indexing techniques of DOTs that can estimate the performance in production environment based on theoretical analysis and key performance parameters from some simple experiments. Ideally, such a model should estimate the insert performance, the query performance, and the storage cost. However, the insert performance of an indexing technique is very difficult to predict mainly because of three reasons.

- *Large Number of Regions.* Generally, the raw table is pre-split into tens to thousands of regions to fully take advantage of the scalability of DOTs. However, a global indexing technique can easily meet the hotspot problem when the values of indexed columns are skewed, which leads to region splits at runtime. Worse still, the selection of CCIT to scan in CCIndex relies on the assumption that regions of index tables have similar size. Therefore improper pre-split regions can decrease the query performance.
- *Large Complexity of Workloads.* Each instance of DOTs in practical may contain serveral workloads and they can influence each other.
- *Parallelism of Inserts.* Various insert pressures (usually measured by query per second, QPS) involve different pressures on the whole cluster, which leads to different insert performance on index tables.

It may be possible to build insert performance model under a low degree of parallelism when the overall pressure is low and the region distribution problem can be ignored. However, such a model is not important since DOTs are mostly used in two cases: 1) continuous insert, which has high QPS; 2) load data via bulk load, which has zero QPS.

Therefore, we focus on the query performance model by estimating query time cost under a low degree of parallelism because query usually has small QPS compared with insert. Besides, since the storage cost is irrelevant to parallelism, we summarize the total storage cost of an indexing technique and get the ratio by comparing the storage cost of an indexing technique with the DOTs without indexes. We name it QSModel since it models the query time and storage cost ratio.

There are also many metrics other than the mentioned three ones, such as update performance, availability, mean time to failure (MTTF), mean time to repair (MTTR) and implementation cost. They are all discarded due to different reasons. Update is discarded because it is not the key concern of indexing techniques. For example, CCIndex has very low update performance, and MD-HBase and LCIndex do not support update at all. The implementation cost is discarded because it is a concept in programming and cannot be quantified. Other metrics, such as availability, MTTF, and MTTR, are related to the failure rate and recovery performance of DOTs, which are difficult to quantify and whose effects are not so important as the query performance and storage overhead.

4.2 Scope of Application

According to Table 1, there are hundreds of theoretical possible combinations of indexing issues, and some of them are not considered in QSModel.

- *M2M Index.* M2M indexes are mainly used for querying cold data and not applicable for online workloads. Besides, the query time and storage cost of different space structures vary, which makes it difficult for estimating the query time and storage cost.
- *Coarse-Grained Index.* Queries in coarse-grained

indexes are difficult to estimate because the size of unnecessary data searched by queries is not predictable.

- *Memory as Cache Index.* Hit rate of caches is highly correlated with the running workload, the size of memory, and the cache replacement algorithm. Therefore the query time is difficult to estimate.

Table 3 defines three kinds of parameters used in QSModel, and the parameters of configuration and workload are confirmed before running, and parameters of basic performance can be calculated from some basic tests.

Table 3. Parameters for QSModel

Parameter Source	Parameter Name	Description
Configuration	r	Number of replicas
Workload	n	Number of records
	k	Number of indexed columns
	l_r	Average length of records
	l_s	Average length of records in secondary indexes
	l_c	Average length of records in clustering indexes
	R	Number of final result records
	R_i	Number of records in an index table covered by a query constraint
	R_s	Number of buckets in the bucket table covered by query constraints
Basic performance	q_{r-r}	Average latency of random reads on the raw table per record
	q_{r-s}	Average latency of getting a record in scanning the raw table
	q_{r-k}	Average latency of skipping a record in scanning the raw table
	q_{q-s}	Average latency of getting a record in scanning a secondary table
	q_{c-s}	Average latency of getting a record in scanning a clustering table
	q_{c-k}	Average latency of skipping a record in scanning a clustering table

4.3 Modeling the Query Time

A full scan on the raw table skips the whole record when one column does not meet the condition. Therefore a scanned record will be either returned or skipped. The total time in querying on DOTs without indexes $T_{\text{NoIndex}} = R \times q_{r-k} + (n - R) \times q_{r-s}$.

As discussed, the index structure decides the procedure of the query, so that we analyze secondary indexes and clustering indexes separately. For secondary indexes, the total query time contains three parts, time for candidate rowkeys T_{Can}, time used to retrieve the final rowkeys T_{Merge}, and time in searching the raw table T_{Final}. Therefore we have the time cost in querying the secondary index $T_{\text{Sec}} = T_{\text{Can}} + T_{\text{Merge}} + T_{\text{Final}} \approx T_{\text{Can}} + R \times q_{r-r}$. Here we neglect T_{Merge} because it is usually much smaller than the other two parts. T_{Final} is processed by random gets on R final rowkeys. The value of T_{Can} is decided by other indexing issues, as shown in Table 4.

Table 4. Impacts of Indexing Issues on Query Parameters Except Index Structure

Issue	Value	Impact
Index mapping	S2S	$T_{\text{Can}} = \sum T_{\text{IndexTable}} = \sum R_i \times q_{s-s}$
	M2S	$T_{\text{Can}} = T_{\text{IndexTable}} = R_i \times q_{s-s}$
Index granularity	-	No impact since coarse-grained is not considered
Index placement	-	No impact since no parallelism is considered
Time to build index	-	No impact since time in scanning MemStore is neglectable
Index persistence	-	Impact on basic performance, no impact on query procedures since memory as cache is not considered

The query procedure of clustering indexes is simpler. It selects the executed columns, scans the executed table, and filters records by other conditions. Hence we have the time cost in querying the clustering index $T_{\text{Clu}} = T_{\text{SelectExeCol}} + T_{\text{ScanExeTbl}} \approx R \times q_{c-s} + (R_s - R) \times q_{c-k}$. The time to select the executed columns $T_{\text{SelectExeCol}}$ is neglected because it is much smaller than the time to scan the executed table $T_{\text{ScanExeTbl}}$.

Table 4 discusses impacts of indexing issues on query procedures except index structure.

4.4 Modeling the Storage Cost Ratio

For a DOT without index, there is no additional storage overhead. Its total storage cost is the cost of the raw table, saying $S_{\text{NoIndex}} = S_{\text{RawTable}} = r \times n \times l_r$.

The total storage cost of indexing techniques (S_{Index}) can be calculated as the sum of the cost of the raw table S_{RawTable} and index tables $S_{\text{IndexTables}}$: $S_{\text{Index}} = S_{\text{RawTable}} + S_{\text{IndexTables}} = r \times n \times l_r + r_{\text{issue}} \times k_{\text{issue}} \times n/N_{\text{issue}} \times l_{\text{issue}}$, where r_{issue}, k_{issue}, N_{issue} and l_{issue} are all determined by indexing issues, as shown in Table 5.

It is noteworthy that if an indexing technique introduces new structures for data recovery using nonpersistent on disk (like CCTs in CCIndex), its storage cost ratio should be calculated separately.

Table 5. Impacts of Indexing Issues on Storage Parameters

Issue	Value	Impact
Index mapping	S2S	$k_{\text{issue}} = k$
	M2S	$k_{\text{issue}} = 1$
Index structure	Secondary	$l_{\text{issue}} = l_s$
	Clustering	$l_{\text{issue}} = l_r$
Index granularity	Secondary	$N_{\text{issue}} = 1$
	Coarse-grained	N_{issue} = the number of records in each unit
Index placement	-	No impact at all
Index building time	-	No impact since the size of MemStore of index tables is neglectable
Index persistence	Persistent on disk	$r_{\text{issue}} = 3$ for default replication, specified for erasure code
	Nonpersistent on disk	$r_{\text{issue}} = 1$
	Memory only	$N_{\text{issue}} = 1$

4.5 Model Instantiation

QSModel is presented for possible combinations of indexing issues. To check the correctness and the precision of QSModel, we should use it to describe some instances of indexing techniques and run experiments. We select CMIndex, CCIndex, LCIndex, and MD-HBase, where all issues considered in QSModel are covered, as shown in Table 2. To describe the query time and storage cost of CCIndex and MD-HBase precisely, we introduce some new parameters for them, as listed in Table 6.

When executing MDRQ, CCIndex and LCIndex will select the column with the minimal number of candidate records; therefore $R_s = \min(R_i)$. CCIndex sets the replica factor of the raw table to 1 and introduces CCTs, and thus its total storage cost is $k \times n \times l_c + r \times k \times n \times l_{ct} + n \times l_r$.

In MD-HBase, the final rowkeys are retrieved by scanning indexes of R_b buckets, and the time is $T_{\text{Can}} = R_b \times B_s \times q_{mb-s}$. The storage cost of bucket table should be considered as well, and it is $r \times n / B_s \times l_{mb}$.

Table 6. Parameters for Analysis

Parameter	Description
l_{ct}	Average length of records in CCTs
l_{mb}	Average length of records in the bucket table of MD-HBase
l_{ms}	Average length of records in the secondary table of MD-HBase
q_{mb-r}	Average latency of random reads on the bucket table of MD-HBase per record
q_{mb-s}	Average latency of getting a record in scanning the bucket table of MD-HBase
q_{ms-s}	Average latency of getting a record in scanning the secondary table of MD-HBase
B_t	Bucket size threshold of MD-HBase
B_s	Average bucket size of MD-HBase
R_b	Number of buckets in the bucket table covered by query constraints

The storage overhead and the query time are summarized in Table 7. As shown in the table there are too many parameters so that it is difficult to use them for prediction. Based on some reasonable assumptions, we give some approximations and the final results are also presented in Table 7.

- In clustering indexes, l_c is longer than l_r, but they have one column less than the raw table. When the number of columns is large, we assume $l_c = l_r$. However, it is noteworthy that the performance of CCIT is different compared with that of the raw table because of the different number of regions.
- In MD-HBase, we suppose the size of buckets is uniformly distributed, namely $B_s = B_t/2$. Meanwhile, B_t is usually over 1 000 in practical; hence we ignore items using B_t as denominator.
- Since the record length of bucket table and secondary table in MD-HBase is similar to that of secondary table in CMIndex, we use the same parameter l_s to present them.
- We use storage cost ratio to evaluate the storage cost of indexing techniques, where the cost without index $r \times k \times n \times l_r$ is used as the baseline.

Table 7. Quantitative Analysis for Representative Indexing Techniques

Indexing Technique	Result		Result after Approximation	
	Query Time	Storage Cost	Query Time	Storage Cost Ratio
NoIndex	$R \times q_{r-k} + (n-R) \times q_{r-s}$	$r \times n \times l_r$	$R \times q_{r-k} + (n-R) \times q_{r-s}$	1
CMIndex[5]	$\sum R_i \times q_{s-s} + R \times q_{r-r}$	$r \times n \times (k \times l_s + l_r)$	$\sum R_i \times q_{s-s} + R \times q_{r-r}$	$1 + k \times l_s/l_r$
CCIndex[9]	$R \times q_{c-s} + (\min(R_i) - R) \times q_{c-k}$	$k \times n \times l_c + r \times k \times n \times l_{ct} + n \times l_r$	$R \times q_{c-s} + (\min(R_i) - R) \times q_{c-k}$	$(k+1)/r + k \times l_{ct}/l_r$
LCIndex[6]	$R \times q_{c-s} + (\min(R_i) - R) \times q_{c-k}$	$n \times (k \times l_c + r \times l_r)$	$R \times q_{c-s} + (\min(R_i) - R) \times q_{c-k}$	$1 + k/r$
MD-HBase[7]	$R_b \times B_s \times q_{mb-s} + R \times q_{r-r}$	$r \times n / B_s \times l_{mb} + r \times n \times (l_{ms} + l_r)$	$R_b \times B_t/2 \times q_{s-s} + R \times q_{r-r}$	$1 + l_s/l$

5 Experiments

The taxonomy and the analysis in above sections present a theoretical understanding of the indexing issues and techniques. As mentioned, we implement CMIndex, CCIndex, LCIndex and MD-HBase to check the results of QSModel. At the same time, we want to solve another problem in evaluating indexing techniques in practice.

As mentioned before, some comparisons of some existing techniques have been presented in literature [6, 10]. However, the existing theoretical understanding and comparisons are still not enough for users to select the best indexing technique because of three reasons.

• Existing experiments are incomplete. For example, LCIndex[6] only compares S2S indexing techniques, and some literature[31] only considers M2S techniques.

• Existing experiments are executed on different workloads. As shown in Table 3, many workload characters parameters impact on the final behavior, like the length of record and the query patterns.

• Existing implementations of these indexing techniques are isolated. Though most of them are implemented on HBase, the underlying versions are different.

In this paper, we propose and implement IndexComparator based on HBase. IndexComparator contains implementations of some representative indexing techniques and provides flexible interfaces to run different workloads. We believe IndexComparator can solve these challenges.

• Though the indexing techniques in IndexComparator are not exhaustive, users can regard some techniques like MD-HBase as "bridges" to compare their target indexing techniques.

• Users can run their own workloads via IndexComparator for practical comparisons.

• IndexComparator is open source[16], adding new implementations of indexing techniques in it for further tests are convenient.

In this section, we run IndexComparator based on a practical workload from Tencent to evaluate QSModel.

5.1 Experimental Environment

We use a cluster with 11 nodes with Xeon E5-2420 v2 CPU, 64 G memory, 1 TB hard disk. The nodes are connected with an Intel® Corporation 82576 Gigabit Network Connections. The servers are all running Linux 2.6.32. Among the cluster, we use one node as NameNode and HMaster, five nodes as DataNode and HRegionServer, and the rest five nodes as clients to input data and execute queries. IndexComparator is implemented on HBase-1.2.1 and we use Hadoop-2.5.1 to provide HDFS. CMIndex, CCIndex and LCIndex all are implemented based on the literature. MD-HBase is extended from project Tiny-MD-HBase[17] and its quad tree is stored in a separate HBase table. HBase without index (NoIndex) is executed as the baseline.

5.2 Workload Analysis

The workload is a statistic summary of the daily advertisement display from Tencent. It contains one table with tens of columns under the same column family. A daily Hive job at midnight generates the table records for the previous day. Since the table is mainly used for advertisers to check their own advertisement effect, the rowkey of the table is concatenated by columns of advertiser ID, advertisement ID, and date to aggregate records belonging to the same advertiser together to improve performance.

However, analysts want to see statistics like the number of views and clicks within a time period. We build indexes on three columns of date, view and click, and use the query like "select * from table where date ∈ [startDate, endDate] and view ∈ [minView, maxView] and click ∈ [minClick, maxClick]" as our query patterns. We dump the data of one week from online servers as the data source, which is tens of gigabytes. The queries and the numbers of final results are shown in Table 8. Totally 10 queries are processed, six queries (Data-A∼Click-B) are on single indexed column, and the rest four (3-A∼3-D) are multi-dimensional range

Table 8. Queries and Number of Results

Index Type	Constraints of Indexed Columns			Number of Results
	Date	View	Click	
Data-A	$[A, C]$	-	-	1 339 731
Data-B	$[A, B]$	-	-	652 256
View-A	-	$[M, O]$	-	1 756 638
View-B	-	$[M, N]$	-	1 216 659
Click-A	-	-	$[X, Z]$	565 644
Click-B	-	-	$[Y, Z]$	329 343
3-A	Data-A	View-A	Click-A	142 474
3-B	Data-B	View-A	Click-A	74 919
3-C	Data-B	View-B	Click-A	35 283
3-D	Data-B	View-B	Click-B	10 821

[16] https://github.com/fengchen8086/ditb, Nov. 2017.

[17] https://github.com/shojinishimura/Tiny-MD-HBase, Nov. 2017.

queries combined by the constraints of single indexed columns queries. Here we use symbols $A < B < C, M < N < O$ and $X < Y < Z$ to replace the practical number used for privacy.

5.3 Experimental Procedures

For each indexing technique in IndexComparator, the experiment is processed as the following steps:

1) delete all files that belong to HBase and HDFS on hard disks,

2) format HDFS, start HDFS and HBase,

3) create tables and insert data from the five clients, each of which has 10 processes and 20 threads,

4) execute queries (BlockCache is disabled),

5) close trash of HDFS, delete /hbase/WALs, /hbase/archive, /hbase/oldWALs directories, and use "du -s HDFS_DATA_ROOT" on all DataNodes for storage cost,

We calculate the insert time of step 3, the query throughput of step 4, and the storage cost of step 5, respectively.

5.4 Results and Discussion

In this workload, $k = 3$ and $r = 3$. To show the effect more clear, we extend the number of total columns to 80; therefore $l_s/l_r = 1/80$, $l_{ct}/l_r = 3/80$. Fig.7 presents the storage cost ratio of experimental and theoretical estimation. The ratios of CMIndex, CCIndex, LCIndex and MD-HBase compared with NoIndex are 1.039, 1.587, 2.125 and 1.015 respectively. Accroding to formulas in Table 7, the estimated ratios are 1.037 5, 1.446, 2.0, and 1.012 5 respectively, which well match the experimental results. The estimated ratios are always larger than the experimental ones because the rowkeys of index tables are always slightly longer than the summary of lengths of indexed column and the raw rowkey. Due to the KeyValue unit structure in HBase, the rowkey is used in every column of clustering index tables; therefore the deviation of experimental and estimated ratios of CCIndex and LCIndex is larger with the columns increasing. Comparing the estimated ratio and the experimental one, we see the maximum error rate of storage cost ratio is 9.8%.

The time cost of different indexing techniques in different queries is shown in Fig.8. Based on the results, we can calculate the values of the parameters of basic performance, as listed in Table 9.

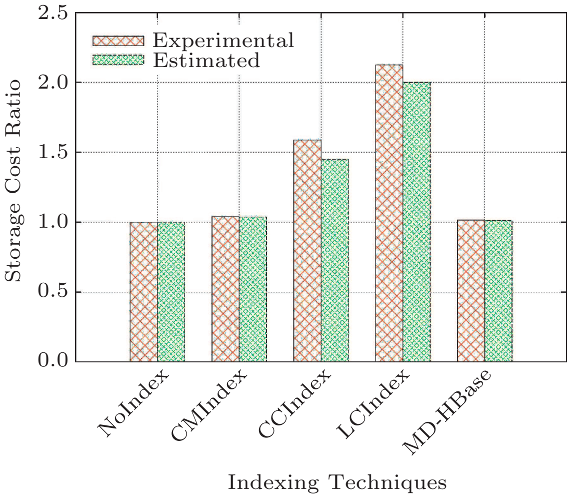


Fig.7. Storage cost ratios.

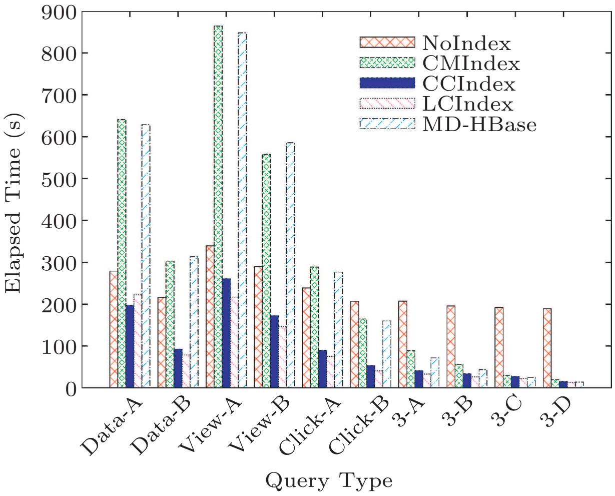


Fig.8. Query time.

Table 9. Parameter Values in Queries

Parameter	Value
q_{r-r}	0.000 472 4
q_{r-s}	0.000 129 1
q_{r-k}	0.000 035 9
q_{s-s}	0.000 005 9
q_{c-s}	0.000 158 8
q_{c-k}	0.000 043 4

Based on the formulas in Table 7, we can estimate the query time. The time of NoIndex, CMIndesx and clustering index is shown in Figs.9(a)∼9(c), respectively. The average error rates for NoIndex, CMIndex, CCIndex and LCIndex are 5.1%, 3.2%, 7.0% and 15.8%, respectively. The maximum error rates for these indexing techniques are 9.5%, 6.6%, 19.4% and 24.2%, respectively.

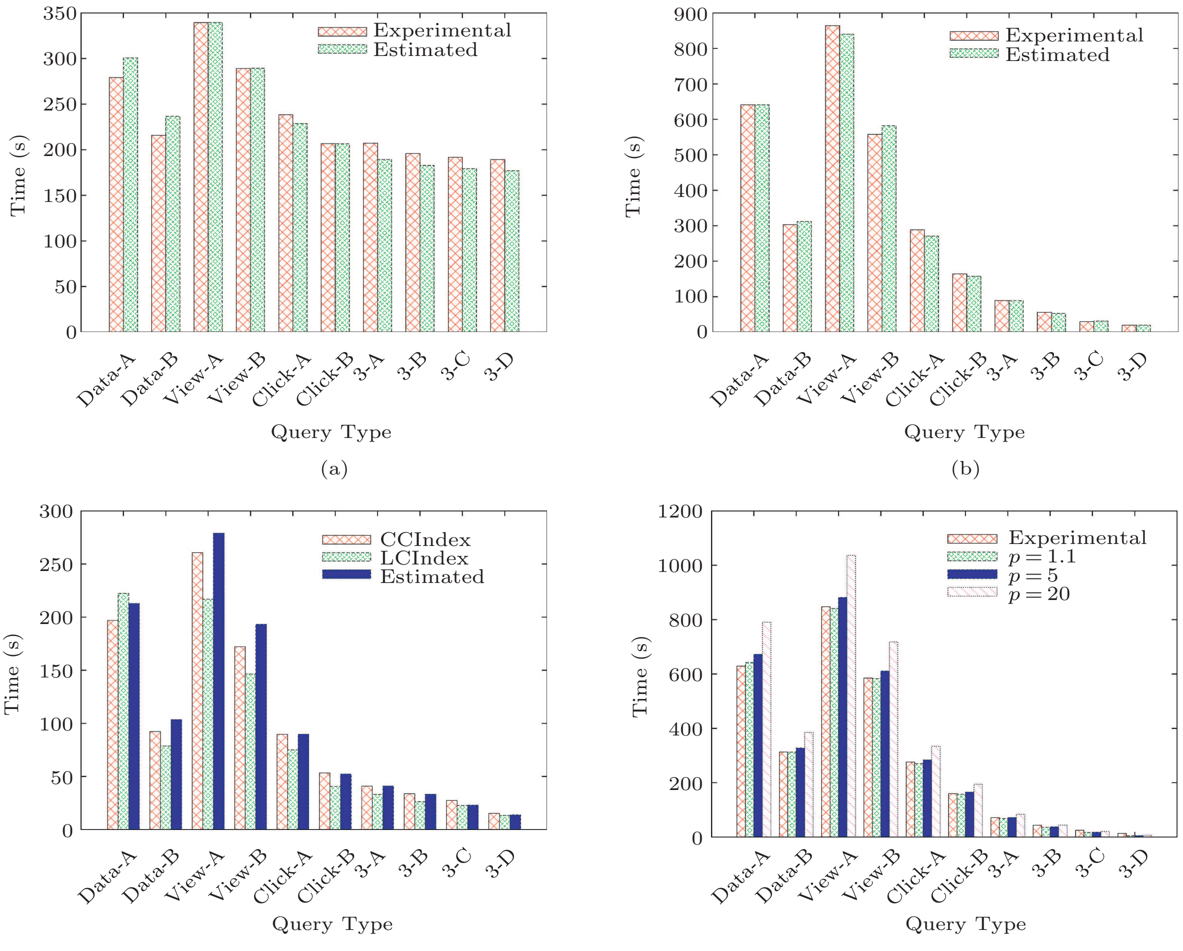


Fig.9. Query time and estimation. (a) NoIndex. (b) CMIndex. (c) Clustering index. (d) MD-HBase.

In MD-HBase, the number of buckets covered by query constraints is hard to estimate; therefore we introduce p to present the ratio between the number of secondary records covered by the buckets and the number of final results. We set $p = 1.1$, 5 and 20 for estimation and the results are shown in Fig.9(d). For $p = 1.1$, the maximum error rates for single column queries and query 3-A are always lower than 5%. However, for queries 3-B, 3-C and 3-D, the error rates are 21.5%, 48.2% and 170%, respectively. For $p = 20$, the maximum error rates for single column queries and query 3-A are always higher than 14%. But for queries 3-B, 3-C and 3-D, the error rates decrease to 1.3%, 20.2% and 119%, respectively. This is because for queries with lots of results, the number of buckets covered by query constraints is much larger than that of the intersecting ones. When the number of final results decreases, the false positive rows in intersecting buckets play a much more important role in the final query time.

As discussed above, the insert performance is difficult to estimate, but a qualitative analysis is feasible. Fig.10 presents the insert throughput of different indexing techniques. Except NoIndex, LCIndex has the highest throughput because indexes are built on flush. The third highest one is CMIndex because the secondary table is much smaller than the raw table. Both CCIndex and MD-HBase have low throughput. The throughput of CCIndex is limited by the huge size of index tables, and insert costs more resource than secondary tables and splits more often. As a secondary index, MD-HBase also has a low throughput. This is because MD-HBase must read the bucket table twice and increase the value of bucket table for each insert, which is processed in parallel with inserts as well, which decreases the insert throughput dramatically. From another point of view, the results for MD-HBase present the difficulty in estimating coarse-grained indexes.

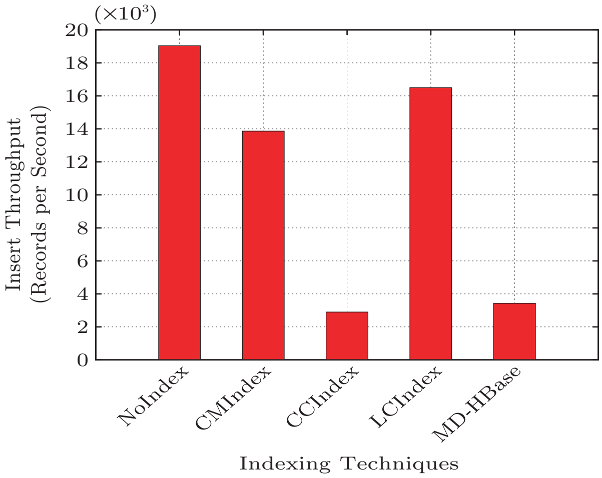


Fig.10. Insert throughput.

6 Conclusions

Numerous techniques have been proposed to build indexes on distributed ordered tables (e.g., HBase), aiming at improving the query performance on non-key columns. However, neither classification nor comparison of them has been presented, which makes it difficult for users to select a proper indexing technique for a certain workload.

In this paper, we analyzed and classified more than 20 existing representative indexing techniques based on the six indexing issues (mapping between indexed columns and the rowkey of index table, index structure, index granularity, index placement, index build time, and index persistence) to describe how these techniques are designed. The proposition of LCIndex in [6] has evidenced the effectiveness of such classification. LCIndex is expected to have high insert performance and MDRQ performance at the same time. The prototype experiment showed that the insert throughput and MDRQ throughput are 72.5% and 3.15x∼7.46x compared with HBase without indexes, respectively. A potential indexing technique named LMSIndex is discussed, and it is multi-dimensional to single dimensional, secondary, local, on flush, and find-grained. LMSIndex is expected to achieve very high insert performance, very low storage overhead, and moderate query performance.

We presented QSModel to describe the query time and storage cost of indexing techniques. Experiments on a practical workload from Tencent showed that the maximum error rate of storage cost rate is 9.8%, and the average error ratio for NoIndex, CMIndex, CCIndex, LCIndex is 5.1%, 3.2%, 7.0% and 15.8%, respectively.

At last, we implemented IndexComparator, an open source prototype of the four indexing techniques, i.e., CMIndex, CCIndex, LCIndex, MD-HBase. Users can use it directly or extend it with other techniques to test the performance on different workloads.

In future, we will implement LMSIndex to validate our prediction, and use QSModel to estimate its performance.

References

[1] Chang F, Dean J, Ghemawat S, Hsieh W C, Wallach D A, Burrows M, Chandra T, Fikes A, Gruber R E. BigTable: A distributed storage system for structured data. *ACM Trans. Computer Systems* (*TOCS*), 2008, 26(2): Article No. 4.

[2] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010, 44(2): 35-40.

[3] Cooper B F, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen H A, Puz N, Weaver D, Yerneni R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 2008, 1(2): 1277-1288.

[4] Harter T, Borthakur D, Dong S Y, Aiyer A, Tang L Y, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Analysis of HDFS under HBase: A Facebook messages case study. In *Proc. the 12th USENIX Conf. File and Storage Technologies*, February 2014, pp.199-212.

[5] George L. HBase: The Definitive Guide. O'Reilly Media, Inc., 2011.

[6] Feng C, Yang X, Liang F, Sun X H, Xu Z W. LCIndex: A local and clustering index on distributed ordered tables for flexible multi-dimensional range queries. In *Proc. the 44th Int. Conf. Parallel Processing*, September 2015, pp.719-728.

[7] Nishimura S, Das S, Agrawal D, Abbadi A E. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *Proc. the 12th IEEE Int. Conf. Mobile Data Management*, June 2011, pp.7-16.

[8] Ma Y Z, Rao J, Hu W S, Meng X F, Han X, Zhang Y, Chai Y P, Liu C Q. An efficient index for massive IOT data in cloud environment. In *Proc. the 21st ACM Int. Conf. Information and Knowledge Management*, October 2012, pp.2129-2133.

[9] Zou Y Q, Liu J, Wang S C, Zha L, Xu Z W. CCIndex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries. In *Proc. the 7th IFIP Int. Conf. Network and Parallel Computing*, September 2010, pp.247-261.

[10] Zhang C, Chen X Y, Shi Z L, Ge B. Algorithms for spatio-temporal queries in HBase. *Journal of Chinese Computer Systems*, 2016, 37(11): 2409-2415. (in Chinese)

[11] Hsu Y T, Pan Y C, Wei L Y, Peng W C, Lee W C. Key formulation schemes for spatial index in cloud data managements. In *Proc. the 13th IEEE Int. Conf. Mobile Data Management*, July 2012, pp.21-26.

[12] Zhou X, Zhang X, Wang Y H, Li R, Wang S. Efficient distributed multi-dimensional index for big data management. In *Proc. the 14th Int. Conf. Web-Age Information Management*, June 2013, pp.130-141.

[13] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996, 33(4): 351-385.

[14] Guttman A. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, June 1984, pp.47-57.

[15] Sellis T K, Roussopoulos N, Faloutsos C. The R$^+$-tree: A dynamic index for multi-dimensional objects. In *Proc. the 13th Int. Conf. Very Large Data Bases*, September 1987, pp.507-518.

[16] Silberschatz A, Korth H F, Sudarshan S. Database System Concepts (3rd edition). McGraw-Hill, 1997.

[17] Wang L, Chen B, Liu Y H. Distributed storage and index of vector spatial data based on HBase. In *Proc. the 21st Int. Conf. Geoinformatics*, June 2013.

[18] Ge W, Luo S M, Zhou W H, Zhao D, Tang Y, Zhou J, Qu W W, Yuan C F, Huang Y H. HiBase: A hierarchical indexing mechanism and system for efficient HBase query. *Chinese Journal of Computers*, 2016, 39(1): 140-153. (in Chinese)

[19] Wang S C, Mares M A, Guo Y K. CGDM: Collaborative genomic data model for molecular profiling data using NoSQL. *Bioinformatics*, 2016, 32(23): 3654-3660.

[20] Serrano D, Han D, Stroulia E. From relations to multi-dimensional maps: Towards an SQL-to-HBase transformation methodology. In *Proc. the 8th IEEE Int. Conf. Cloud Computing*, July 2015, pp.81-89.

[21] Chang C R, Hsieh M J, Wu J J, Wu P Y, Liu P F. HSQL: A highly scalable cloud database for multi-user query processing. In *Proc. the 5th IEEE Int. Conf. Cloud Computing*, June 2012, pp.943-944.

[22] Agrawal P, Silberstein A, Cooper B F, Srivastava U, Ramakrishnan R. Asynchronous view maintenance for VLSD databases. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, June 2009, pp.179-192.

[23] Zhou X, Li H, Zhang X, Wang S, Ma Y Y, Liu K Y, Zhu M, Huang M L. ABR-Tree: An efficient distributed multidimensional indexing approach for massive data. In *Proc. Int. Conf. Algorithms and Architectures for Parallel Processing*, November 2015, pp.781-790.

[24] Zhang Y, Ma Y Z, Meng X F. Efficient processing of spatial keyword queries on HBase. *Journal of Chinese Computer Systems*, 2012, 33(10): 2141-2146. (in Chinese)

[25] Zhang N Y, Zheng G Z, Chen H J, Chen J Y, Chen X. HBaseSpatial: A scalable spatial data storage based on HBase. In *Proc. the 13th IEEE Int. Conf. Trust Security and Privacy in Computing and Communications*, September 2014, pp.644-651.

[26] Han D, Stroulia E. HGrid: A data model for large geospatial data sets in HBase. In *Proc. the 6th IEEE Int. Conf. Cloud Computing*, June 28-July 3, 2013, pp.910-917.

[27] Zhou W, Han J Z, Zhang Z, Dai J, Xu Z Y. HDKV: Supporting efficient high-dimensional similarity search in key-value stores. *Concurrency and Computation: Practice and Experience*, 2013, 25(12): 1675-1698.

[28] Ma Y Z, Zhang Y, Meng X F. ST-HBase: A scalable data management system for massive geo-tagged objects. In *Proc. the 14th Int. Conf. Web-Age Information Management*, June 2013, pp.155-166.

[29] Tang X S, Han B D, Chen H. A hybrid index for multi-dimensional query in HBase. In *Proc. the 4th Int. Conf. Cloud Computing and Intelligence Systems*, August 2016, pp.332-336.

[30] Ma Y Z, Meng X F, Wang S Y, Hu W S, Han X, Zhang Y. An efficient index method for multi-dimensional query in cloud environment. In *Proc. the 2nd Int. Conf. Cloud Computing and Big Data in Asia*, June 2015, pp.307-318.

[31] Chen X Y, Zhang C, Ge B, Xiao W D. Efficient historical query in HBase for spatio-temporal decision support. *International Journal of Computers Communications & Control*, 2016, 11(5): 613-630.

[32] Lee K, Ganti R K, Srivatsa M, Liu L. Efficient spatial query processing for big data. In *Proc. the 22nd ACM SIGSPATIAL Int. Conf. Advances in Geographic Information Systems*, November 2014, pp.469-472.

[33] Wang H P, Ci X, Meng X F. Fast multi-fields query processing in bigtable based cloud systems. In *Proc. the 14th Int. Conf. Web-Age Information Management*, June 2013, pp.142-154.

[34] Huang S, Wang B T, Zhu J Y, Wang G R, Yu G. R-HBase: A multi-dimensional indexing framework for cloud computing environment. In *Proc. IEEE Int. Conf. Data Mining Workshop*, December 2014, pp.569-574.

[35] Li Q C, Lu Y, Gong X L, Zhang J. Optimizational method of HBase multi-dimensional data query based on Hilbert space-filling curve. In *Proc. the 9th Int. Conf. P2P Parallel Grid Cloud and Int. Conf. Computing*, November 2014, pp.469-474.

[36] Li S, Hu S H, Ganti R, Srivatsa M, Abdelzaher T. Pyro: A spatial-temporal big-data storage system. In *Proc. the USENIX Annual Technical Conf.*, July 2015, pp.97-109.

[37] Du N B, Zhan J F, Zhao M, Xiao D R, Xie Y C. Spatio-temporal data index model of moving objects on fixed networks using HBase. In *Proc. IEEE Int. Conf. Computational Intelligence & Communication Technology*, February 2015, pp.247-251.

Chen Feng is a Ph.D. candidate of Institute of Computing Technology, Chinese Academy of Sciences, Beijing. He received his B.S. degree in software engineering from Nankai University, Tianjin, in 2011. His current research interests include big data computing and distributed system. He is a student member of CCF.

Chun-Dian Li is a Ph.D. candidate of Institute of Computing Technology, Chinese Academy of Sciences, Beijing. He received his B.S. degree in software engineering from Wuhan University, Wuhan, in 2013. His research interests are modeling and analysis of distributed system. He is a student member of CCF.

Rui Li is a software engineer of Tencent Inc., Beijing. He received his Ph.D. degree in electronic engineering from Peking University, Beijing, in 2008. His current research interests include distributed storage and data processing.