CCIndex: a Complemental Clustering Index on Distributed Ordered Tables for Multi-dimensional Range Queries

Yongqiang Zou, Jia Liu, Shicai Wang, Li Zha, Zhiwei Xu

CCIndex: a Complemental Clustering Index on Distributed Ordered Tables for Multi-dimensional Range Queries

Yongqiang Zou, Jia Liu, Shicai Wang, Li Zha, Zhiwei Xu

Institute of Computing Technology, Chinese Academy of Sciences
Beijing, 100190, China
{zouyongqiang, liujia09, wangshicai}@software.ict.ac.cn, {char, zxu}@ict.ac.cn

Abstract. Massive scale distributed database like Google's BigTable and Yahoo!'s PNUTS can be modeled as Distributed Ordered Table, or DOT, which partitions data regions and supports range queries on key. Multi-dimensional range queries on DOTs are fundamental requirements; however, none of existing schemes work well while considering three critical issues: high performance, low space overhead, and high reliability. This paper introduces CCIndex scheme, short for Complemental Clustering Index, to solve all three issues. CCIndex creates several Complemental Clustering Index Tables for performance, leverages region-to-server information to estimate result size, and supports incremental data recovery. This paper builds a prototype on Apache HBase. Theoretical analysis and micro-benchmarks show that CCIndex consumes 5.3% ~ 29.3% more space, has the same reliability, and gains 11.4 times range queries throughput of secondary index scheme. Synthetic application benchmark shows that CCIndex query throughput is 1.9 ~ 2.1 times of MySQL Cluster.

Keywords: Clustering, index, range queries, multi-dimensional

1 Introduction

Massive scale distributed databases like Google's BigTable [1] and Yahoo!'s PNUTS [2] gain more and more attention to store data for Internet scale applications. These systems can be modeled as Distributed Ordered Table, short as DOT, which partitions continuous keys to regions, replicates regions for performance and reliability, distributes regions to shared-nothing region servers for scalability, serves as tables and columns, and supports range queries on keys. Multi-dimensional range queries on DOT systems are natural requirements. For example, a query needs to find out nearby restaurants through "latitude > 48.5 and latitude < 48.6 and longitude > 112.5 and longitude < 112.8 and type = restaurants". Another example is finding out hottest pictures in this week in a photo-sharing application, such as Flickr, with a query like "timestamp > 1267660008 and rank > 1000". With only DOT's range queries over key, a multi-dimensional range query is a table scan over key with predicates on non-key columns to filter results, which is very ineffective for a low selectivity query on

non-key columns, and the latency is unacceptable in large scale data sets, such as TB or PB level. This paper refers to selectivity by the percentage of records passing the predicate. The inefficiency requires index on non-key columns to accelerate multi-dimensional range queries. However, multi-dimensional range queries over DOT are big challenges if we considering the three critical issues: high performance, low space overhead, and high reliability.

None of existing schemes can work well considering all the three issues. Building secondary indexes for non-key columns through creating ordered tables to store indexes is common. However, the range query over secondary index is significantly slow, because random read is slower than scan (eg. In BigTable is 13.7 times). Other "better" index schemes without clustering data will encounter the same problem as slow random read. Clustering index reduces the random reads but needs several times storage, and data recovery is a big issue if the underlying replica mechanism is disabled. Lacking of statistics on DOTs imposes more difficulties on optimizing multi-dimensional range queries.

This paper introduces a new scheme CCIndex, short for Complemental Clustering Index, to support multi-dimensional range queries over DOT while achieving high performance, low space overhead, and high reliability. CCIndex creates several Complemental Clustering Index Tables, or CCIT, each for a search column with the full row data, which makes range query over this column a range scan. CCIndex leverages the region-to-server mapping information to estimate the result size of each queries. CCIndex disables the underlying data replica mechanisms to avoid too much storage overhead, and creates a replicated Complemental Check Table, or CCT, for each search column to support incremental data recovery.

CCIndex prototype has been built on Apache HBase, a subproject of Hadoop. Theoretical analysis and experimental evaluations have been given.

The rest of this paper is structured as follows. Section 2 presents related work. Section 3 describes the CCIndex architecture design, including the construction of the index. Section 4 presents query processing and optimization. Section 5 gives the fault tolerant mechanisms. Section 6 gives detailed evaluations. Section 7 concludes the paper.

2 Related work

Recently, some research focuses on index mechanisms over DOT. Yahoo! focuses on optimizing range queries on DOT through adaptive parallelizing [3], and multi-dimensional range queries are done through range scan over primary key with predicates. This approach is very ineffective with low selectivity queries. Google and Yahoo! claim the future work on secondary index over DOTs [4, 2]. A currently available secondary index over DOT is the IndexedTable mechanism in Apache Hbase [5]. IndexedTable creates a new table for each index column, saves it in the DOTs in the order of the index value. IndexedTable is more effective for low selectivity queries than table scan, and has acceptable space overhead and fault tolerance ability. However, index scan needs random reads on original table which is very slow. Traverse [6] builds B-tree [7, 8] index for the map-reduce-merge system.

Traverse has the same performance problem as IndexedTable, and is lack of reliability due to the non-replica B-tree indexes. CCIndex is better than these approaches in space overhead, reliability, and index scan performance.

Multi-dimensional range queries in databases are topics gaining attentions for more than 20 years. R-tree [9], R^+-tree [10], and their successors extend B-tree, divide the multi-dimensional space, and store the recursively divided spaces as tree nodes. Queries walk through the tree to find out the data block. These schemes does not consider the reliability problem over DOT, unless they are implemented in a scalable and reliable way, just like the distributed B-tree [11]. Even though, the performance degradation also exists due to missing clustered data.

DB2 introduces multi-dimensional clustering [12] to form every unique combination of dimension values as a logical 'cell', which is physically organized as block of pages. Multiple B-tree indexes are built for every dimension and the B-tree leaves point to the block. This scheme can avoid random read only when the values in block are dense, and the reliability of the B-tree index is not considered.

Parallel databases [13, 14] support multi-dimensional queries and have good reliability, such as the MySQL cluster [15]. CCIndex is designed for more scalable DOTs to get good performance with large dataset and many machines.

DHTs [16, 17, 18, 19] are scalable and reliable for key-value pair storage. Because the data is partitioned by hashing functions, DHT systems do not support range queries naturally. MAAN [20] and SWORD [21] use locality preserving hashing and store attributes in DHT as index to support range queries. However, the logN hop latency is not good for user-interactive applications.

3 Data layout and management

This section introduces the CCIndex ideas and the underlying data layout.

3.1 Basic idea

The CCIndex is inspired through these observations: (1) There are usually 3 to 5 replica in the DOT systems to assure reliability and improve performance. (2) The indexes number is usually less than 5. (3) The random reads is significantly slower than scan. The trick of CCIndex is reorganizing the data to a new layout to accelerate multi-dimensional range queries. CCIndex introduces several Complemental Clustering Index Tables, each for a search column with the full row data, to convert the slow random reads to fast range scan. With multiple tables, a key decision is determining which table is chosen to scan. CCIndex leverages the region-to-server mapping information to estimate the result size of each sub queries. CCIndex disables the underlying data replica mechanisms to get an acceptable storage overhead, and uses these Complemental Clustering Index Tables to recovery each other to assure reliability. CCIndex creates a replicated Complemental Check Table for each search column to support incremental data recovery.

3.2 Data Layout

In DOT systems, tables are very tall and logically ordered by row keys. Physically, each table is partitioned to regions containing continuous ranges, and each region has several replicas identical to each other for fault tolerance. CCIndex reorganizes the underlying data layout as in Fig. 1.

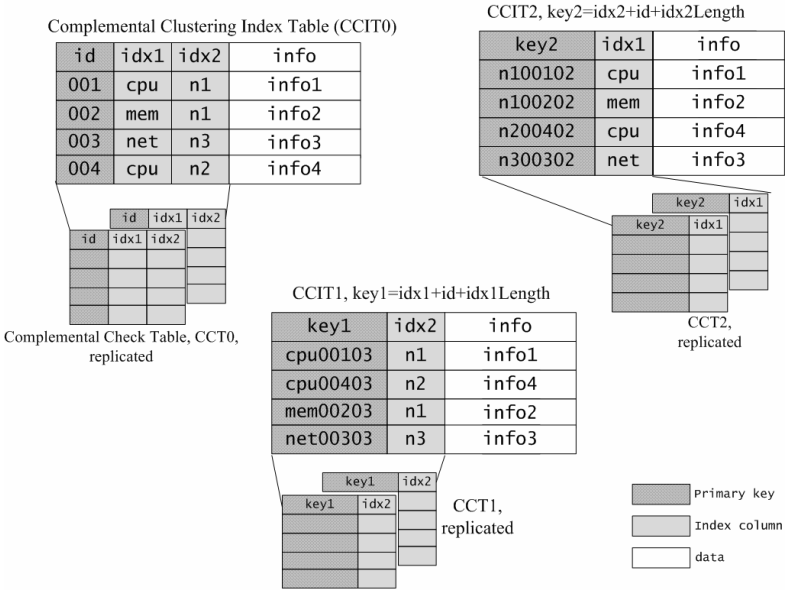

Fig. 1. Data layout of CCIndex. For a logical table has *id* as the primary key with two index columns *idx1* and *idx2*, CCIndex creates another two CCIT tables each for a index column and reorganizes the rows in the order of this column. CCIndex creates replicated CCTs for fast data recovery.

In the data layout, each logical table has several complemental tables. In Fig. 1, there is a table to support range queries over *id* and two index columns *idx1* and *idx2*. The table CCIT0 is the original table ordered by unique *id*. CCIT1 and CCIT2 are ordered by *key1* and *key2*, which are generated by concatenating index column value, the original id, and the index column value length. The construction of the new key makes sure the new CCITs are ordered by index column values, and makes the duplicated values of index columns be unique keys. The index value length field makes it easy to split the index value and id. With these CCITs, range queries over id or index columns can be a scan on the corresponding CCITs.

Each CCIT has a corresponding replicated CCT, which contains the primary key and index columns of the CCIT. CCTs are necessary to help incremental data recovery of CCITs. The CCITs have no replica but the CCTs have replicas.

Fig. 1 shows the logical view of CCIndex, and these tables are physically stored in DOT system. Storing CCITs and CCTs in DOT leverages the primary key ordering, data partition, and various operation optimizations to simplify CCIndex implementation.

3.3 Index create, update, and delete

The index maintenance is done along with the record insert and delete operations.

When CCIndex creates a table with specified index columns, all CCITs are created. When a record is written to DOT, the CCIndex first reads the original table to check whether the index column values are changed, and delete the corresponding records in CCITs when necessary. After that, the CCIndex writes the records to all CCITs in a parallel way. The delete operation also involves all the CCITs.

4 Query processing and optimization

The DOT read and scan operation are simply redirecting to the original CCIT. The index scan is processed by CCIndex.

4.1 Query plan generation and execution

CCIndex introduces a SQL-like syntax to expression multi-dimensional range queries. The query string is like this:

```
select rowkey, host, service, time, status from MonitoringData where host='node 216' and service='CPU Load' and (time > 1260610511 and time < 1260610521)
```

CCIndex translates the SQL expression to a query plan tree, optimizes the tree, and translates the tree to disjunctive form. Then CCIndex executes the logic OR part in parallel, and executes each internal AND part by the estimated optimal one query with predicates of other columns to filter rows.

4.2 Query plan optimization

CCIndex first does simple optimization of query plan tree to eliminate redundant range queries. For example, the *time > 123 and time > 135* could be merged into *time > 135*. Furthermore, the important optimization is estimating result size of multiple AND queries and choosing the minimal one.

In databases, query optimization is based on statistics of tables. However, DOT systems are lack of statistics, because the statistics are very difficult to gather and maintain in massive scale tables maintained by thousands of region servers. For example, there is not any statistics in HBase, and an additional tool must be written to count table rows.

CCIndex introduces a way to estimate the query result size in the absence of statistics. CCIndex's estimation method relies on the region-to-server mapping information of DOTs. The mapping information is necessary for DOT systems to record the responsible region server for each region. DHTs have no such information, because the mapping relationship is deduced by the overlay topology, object ids, and node ids.

The mapping information can be abstracted in the form of <regionStartKey, RegionServerInfo>. The regionStartKey is the minimal key in this region and serves as the region id. The mapping information is gathered together and ordered by the regionStartKey. CCIndex scans all this mapping information using a binary search, and finds out the number of covered regions for each range query. CCIndex claims that the region number determines the result size, because for a DOT containing more than one region and having 64 MB default region size, each region size must be between 32 MB and 64 MB. For the first or last region not fully covered in a range, the coverage ratio is calculated to estimate the result size for the regions.

This policy is more accurate for large query result size, because the average region size is more accurate when there are lots of regions. For query covering few of regions, the detailed size is not important, because this result size is small and the estimation objective is determining a query with small result size to execute.

5 Fault tolerance

In CCIndex, CCITs have no replica to avoid huge storage overhead, and cause the problem of fault tolerance. The basic idea is that CCITs replicate and recovery each other in record level. However, when a region of a CCIT is damaged, we can only reconstruct the whole CCIT if there is lack of ways to gather necessary records to recovery the region.

CCIndex introduces the CCTs to help recoverying the damaged region. CCIndex only checks the corresponding CCTs to get the proper keys for CCITs and get the record data to rebuild the region.

The CCTs imposes additional overhead on inserting or deleting records. CCIndex maintains the CCTs in an asynchronous way to minimize the overhead. CCIndex leverages the log of DOTs to update CCTs in batch mode by the background threads.

6 Implementation and Evaluations

This paper implements a CCIndex prototype and evaluates CCIndex through theoretical analysis, micro benchmarks, and synthetic application benchmarks.

6.1 Implementation

This paper builds a CCIndex prototype based on Apache Hbase, an open-source implementation of BigTable. Hbase is a sub-project of Apache Hadoop [22], which has HDFS as the distributed file system and MapReduce as the parallel computing model. HBase builds on top of HDFS, has one master process called HMaster and many slave processes called HRegionServer to manage data regions.

The CCIndex prototype uses HBase v0.20.1 as code base, adds clustering index table package, and implements CCIndex in Java. CCIndex disables the replica of HDFS by setting the replica factor to one, and creates one CCIT for each search

column. CCIndex builds several CCTs on replicated HFDS files to achieve reliability. HBase has multiple META regions, and each META region contains mapping of a number of user regions comprising the tables to HRegionServers. HBase has a ROOT region to locate all the META regions. CCIndex scans the ROOT and META regions to get the region-to-server mapping information and estimates the query result size.

The comparable IndexedTable is a built-in index mechanism provided in HBase. IndexedTable creates a replicated ordered table for each index column and is an implementation of secondary index scheme. IndexedTable does not provide multi-dimensional range queries interface or optimization to estimate query result size for multi-dimensional range queries.

6.2 Theoretical analysis

For the three metrics performance, space overhead, and reliability, the first one is easy to evaluate through experiments, while the other two are more suitable to do theoretical analysis to get more insight.

Theorem 1. The space overhead ratio of CCIndex to IndexedTable is

$$(N*N+1)/(2*N+(N+1)*L/L_n)$$

Where N is the number of index columns without primary key, and L/L_n is factor that total record length divided by the sum of index column lengths and key length, with the suppose that the replica factor for record data is $N+1$, and index column and primary key has the same length.

Proof. In IndexTable, the space for each record is the original table plus index:

$$S_{ii}=(L_k+L_i)*N*F+L*F \quad (1)$$

Where N is the number of index columns, F is the replica number, L_k is the length of key, L_i is the average length of index columns, and L is the total length of a record.

In CCIndex, the space for each record is the CCITs plus CCTs. The space for CCTs is:

$$S_c = (L_k+N*L_i)*N*F + (L_k+N*L_i)*F = (L_k+N*L_i)*(N+1)*F \quad (2)$$

The total space for CCIndex is:

$$S_{cc}=S_c+L*(N+1) \quad (3)$$

If $L_k = L_i$, $F = N + 1$, the space overhead ratio of CCIndex to IndexedTable is:

$$(S_{cc}-S_{ii})/S_{ii} = (N*N+1) / (2*N+L/L_k) \quad (4)$$

Let $L_n=L_k + N*L_i$, then the formula (4) is:

$$(S_{cc}-S_{ii})/S_{ii} = (N*N+1)/(2*N+(N+1)*L/L_n) \quad (5)$$

□

The equation (5) in theorem 1 can be plotted as Fig. 2.

From Fig. 2, the overhead ratio drop significantly as the L/L_n increases and the N decreases, which indicates that CCIndex should have less columns to index and all index columns should have small length to avoid big space overhead. If N changes from 2 to 4 and the L/L_n changes from 10 to 30, then the overhead changes from 5.3% to 29.3%.

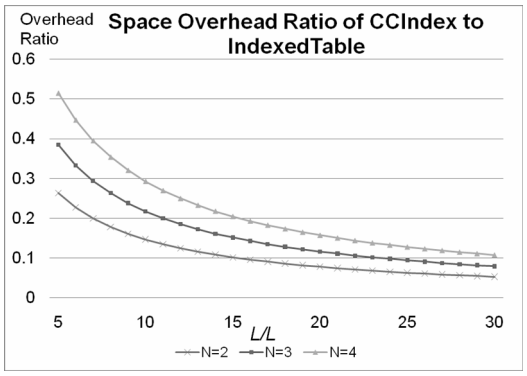

Fig. 2. The space overhead ratio of CCIndex to IndexedTable. The overhead ratio drops significantly as the L/L_n increases and the N decreases. If N changes from 2 to 4 and the L/L_n changes from 10 to 30, then the overhead changes from 5.3% to 29.3%.

Theorem 2. In CCIndex, the probability of being able to recovery a damaged record is

$$(1 - f^{(N+1)})^2$$

Where f is the probability of a record damages, N is the indexed column number. The probability is the same as that of IndexedTable.

Proof.

CCIndex recoveries the data through CCTs and CCITs.

CCTs have N replicas plus another copy in the corresponding columns in CCIT and the probability of failing to read from all CCTs is $f^{(N+1)}$. The probability of replicas for a given record in all CCITs are damaged is $f^{(N+1)}$.

So, the probability of being able to recovery a damaged record is $(1 - f^{(N+1)})^2$.

For IndexedTable, data access relies on replicated index and the original table. The probability is obviously the same as CCIndex.

□

6.3 Micro benchmarks

BigTable introduces a micro benchmark to evaluate the basic operations throughput, including random *read/write*, sequential *read/write*, and *scan*. The workload is comprised of a table with 1KB rows, and each row has an additional 10 bytes rowkey. The throughput is defined as rows per seconds for all clients. HBase implements this micro benchmark and has single thread client, multi-threads clients, or MapReduce clients to evaluate the throughput. CCIndex extends the micro benchmark by adding an "index" column family to contain three columns and each is 10 bytes, and building three indexes using these columns. CCIndex adds an *IndexScan* operation to scan through the first column index.

We setup an experimental environment having two clusters. The small cluster has 3 nodes for micro benchmarks, and the big one has 16 nodes for synthetic application benchmark. Each node has two 1.8 GHz dual-cores AMD Opteron (tm) Processor 270, 6 GB memory. Each node in the small cluster has 321 GB RAID5 SCSI disks, and

each node in the big cluster has 186GB RAID1 SCSI disks. All nodes in each cluster are connected by Gigabits Ethernet. Each node uses Red Hat CentOS release 5.3 (kernel 2.6.18), ext3 file system, Sun JDK1.6.0_14, Hadoop v0.20.1, and HBase 0.20.1. The HBase itself uses 3 GB heap memory.

In our experiments, we choose the workloads which have 1 million rows, and run each tests three times to report the average value. The client uses one of the 3 machines with three concurrent threads. The micro benchmarks use 3 machines.

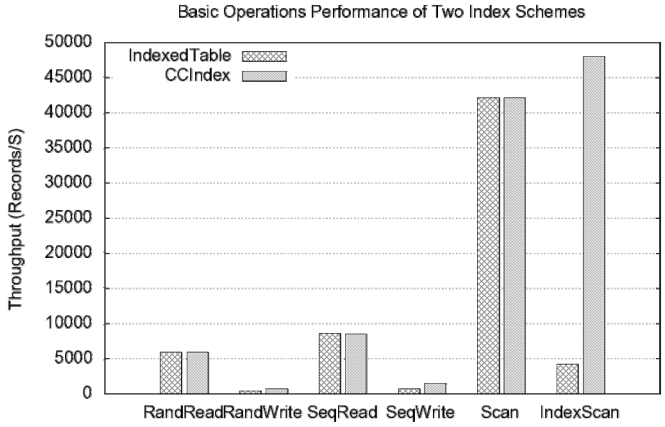

Fig. 3. Basic Operation Performance of Two Index Schemes. CCIndex throughput is 11.4 times of IndexTable's in *IndexScan* operation. CCIndex random *write* and sequential *write* operations is 54.9% and 121.4% better than that of IndexTable.

We compare CCIndex with IndexedTable in HBase and show results in Fig. 3. CCIndex's *IndexScan* operation throughput is 11.4 times of IndexTable's, which shows the benefits of CCIndex through avoiding random reads in primary key. CCIndex *random write* and *sequential write* operations are 54.9% and 121.4% better than those of IndexTable, which is due to the parallel index updating. The *scan*, *random read*, and *sequential read* of these two schemes are nearly identical due to the same logic path.

We compare the throughput of CCIndex with the original table, and the result is in the Fig. 4. The *IndexScan* is unavailable for origin table without index. CCIndex *IndexScan* throughput is 10.9% more than origin table, which is due to the first column in the "index" family is moved to the row key, so the data length of CCIndex table is smaller than original table's. Fig. 4 further interprets why CCIndex can gain an order of magnitude improvement over IndexedTable. For IndexedTable, a range query over an index column should first scan the index table, and then issue multiple random reads in the original table to get the row data. In IndexedTable, throughput of *scan* over index table is nearly the same as scanning the original table. While in CCIndex, range query over an index column is done by *IndexScan*, which scans over corresponding CCIT. The *IndexScan* throughput for CCIndex is 8.2 times of *random read* in original table, and 1.1 times of *scan* in original table, so the throughput is at least 9.3 times over IndexedTable, because IndexedTable needs additional time to parse and wrap intermediate results.

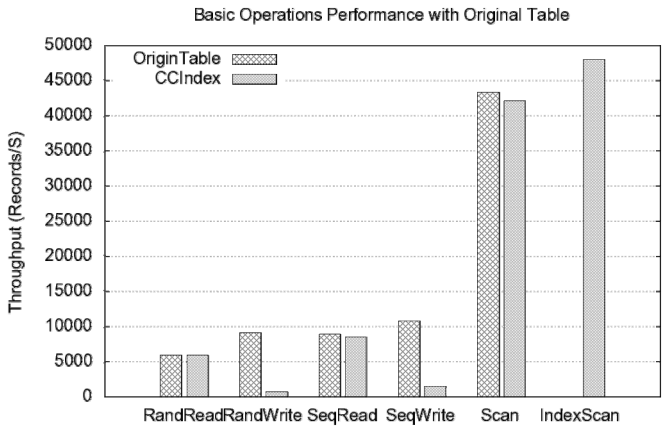

Fig. 4. Basic Operation Performance with Original Table. CCIndex *IndexScan* throughput is 10.9% more than origin table. The *random write* and *sequential write* is significantly lower than the origin table due to the overhead to maintain index, which is a common issues for both index schemes.

The throughput of *random write* and *sequential write* for CCIndex is significantly lower than the origin table, because maintaining index needs another *random read* to get row data for checking whether to change index column value, and a further *delete* and *write* to update index if is necessary.

Because scanning index to get the matching row data is the most important functions of building index, we claim that CCIndex significantly outperforms IndexedTable and is suitable for range queries over indexes. However, we should carefully choose the workloads having less *write* operations and choose more stable index columns to avoid the performance degradation of *write*, and these are general guidelines for all two index schemes.

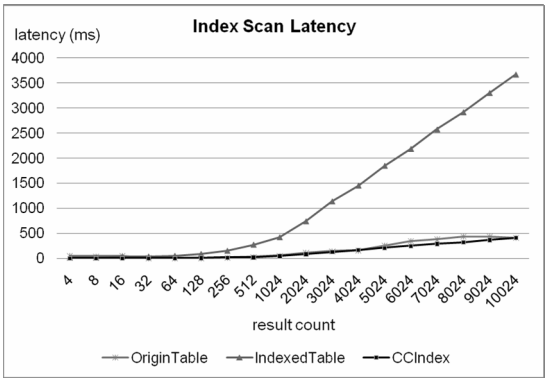

Fig. 5. *Index scan* latency of all three schemes. CCIndex is 9.2 times faster than IndexedTable when the result count is larger than 1024.

The following experiments show the *index scan* latency of all the three schemes in different result count. We use the *scan* over primary key to represents the unavailable *index scan* of the origin table. The results are illustrated in Fig. 5.

From Fig. 5, the CCIndex latency is significantly smaller than IndexedTable, and the ratio is stable at 9.2 when the result count is larger than 1024. Another interesting thing is that the absolute latency of CCIndex is low, and the round-trip latency to get 1024 1KB continuous rows is 42 micro-seconds. The low latency of CCIndex over HBase shows the ability of serving high user-interactivity applications, such as blog, wiki, and twitter.

6.4 Synthetic Application benchmarks

Multi-dimensional range queries are not directly supported by IndexedTable, so we designed a suite of experiments to compare the performance with the memory-based parallel database MySQL cluster.

There are no common accepted benchmarks for multi-dimensional range queries over DOTs yet. The well-known benchmarks for database, such as TPC-C [23] and TPC-H [24], have a majority of operations not supported by DOT, such as transactions over multiple records and complex queries with joins and aggregations.

This paper designs a synthetic application benchmark by analyzing a well-known cluster monitoring application Nagios [25]. Nagios supports comprehensive monitoring of operating systems, applications, network protocols, system metrics, and network infrastructure through user-configured monitoring items, called "service", in a fixed interval on all hosts in a cluster. Nagios records the information about launching a monitoring item on a host into the log, including timestamp, host, service, execution time, and the response message for this monitoring item, etc. Nagios provides a web portal contacting backend CGI programs to read monitoring data and show various aspects of the cluster. The log information volume is exposing if we have more monitor items, more hosts, shorter interval, and a longer period of information to store.

Through analyzing the application logic of the Nagios web portal, we construct a table *ServiceTime* using *host* concatenating *service* and *time* as the primary key, and with *service* and *time* as the record. We design two queries for our tests.

- **AndQuery**: Multi-dimensional range queries with AND operations results in a big result count. The query likes "`select * from ServiceTime where (primaryKey > K1 and primaryKey < K2) and (time > k3 and time < k4) and (service = 'CPU Load')`". The query runs with multiple clients concurrently, each with different ranges to get load balance. The result count for each client is about 5 million.
- **OrQuery**: Multi-dimensional range queries with OR operations results in a big result count. The query is similar as AndQuery, but uses OR to connect different dimensions. The result count for each client is about 10 million.

These queries should be run in multiple clients to get the total throughput of all clients.

We use the 16 node cluster described in the micro benchmarks, and there is totally 64 cores and 96 GB memory. In our experiments, we collect more than 120 million monitoring records with average record length 118 bytes.

The MySQL cluster is version 7.09, which is configured with 1 management node, 2 SQL nodes, and 14 data nodes. In this test the maximum data node number is 14 because data nodes must not co-located with management node and must be even number. The HBase regionserver in each node has 3GB heap memory.

In the following tests, we use at most 90 million records because it reaches the capacity limits of our configured MySQL cluster. We allocate 3 GB as the data memory for each MySQL data node. MySQL cluster stores all records in data memory and cannot accept new records when the memory is all consumed.

In the tests, each node runs an instance of client. Fig. 6 shows the results.

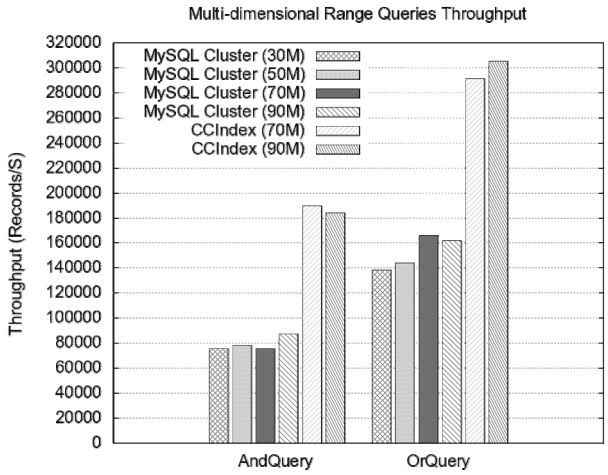

Fig. 6. Multi-dimensional range queries throughput for multiple clients. MySQL Cluster performance is stable when the data sets increases from 30 million to 90 million records. With the 90 million records, CCIndex AndQuery and OrQuery throughput is 2.1 and 1.9 times of the memory-based parallel database MySQL Cluster.

CCIndex AndQuery and OrQuery throughput is 2.1 and 1.9 times over MySQL Cluster with 90 million records dataset, which shows CCIndex performance is significant better than MySQL Cluster.

In Fig. 6, the MySQL Cluster performance is stable for AND and OR queries when the data sets increase from 30 million to 90 million records. However, the MySQL Cluster scalability problem is that the capacity is determined by the total memory for data, because all the data has a copy in memory, which improves the performance but limits the capacity.

6.5 Discussion

The CCIndex can be applied to DOT systems with few of columns to index, which has great impact on the storage overhead. For a table more than 5 columns having

query requirements, the practical solution is identifying the most frequently used columns to build index with CCIndex, or combines some columns to reduce the column number.

CCIndex practically does not support adding or removing index after the table is created, for the reason that creation or deleting of CCIT costs unaffordable time for massive scale data. Another problem is that CCIndex *write* operation is slower than the original table. These two are common problems for many index schemes.

In CCIndex, the probability of being able to recovery a damaged record is fairly good; however, the data recovery time is longer than IndexedTable scheme. Because in CCIndex, recovery a region needs gathering all records by *random read* in other CCITs, which is slower than copying a 64 MB region data file.

7 Conclusions and future work

This paper models the massive scale databases as Distributed Ordered Table, or DOT, which partitions continuous keys to regions, replicates regions for performance and reliability, distributes regions to shared-nothing region servers for scalability, serves as tables and columns, and supports range queries on keys. This paper formulates the problem as supporting multi-dimensional range queries over DOT while considering the three metrics: high performance, low space overhead, and high reliability.

This paper proposes a scheme called CCIndex, short for Complemental Clustering Index, to tackle this problem. CCIndex introduces Complemental Clustering Index Tables each for a search column with the full row data to reorganize data and improve query performance. CCIndex leverages the region-to-server mapping information to estimate the result size of each query without statistics. CCIndex disables the underlying data replica mechanisms to avoid too much storage overhead, and introduces replicated Complemental Check Table to support incremental data recovery.

CCIndex prototype has been built on Apache HBase. Theoretical analysis shows that CCIndex consumes 5.3 ~ 29.3% storage more than secondary index scheme in HBase for typical situations and the probability of failing to recovery bad rows is the same as secondary index scheme. Micro benchmarks show that CCIndex throughput of range queries on non-key column is about 11.4 times of secondary index. The synthetic monitoring application range queries in a 16-node cluster shows that CCIndex AndQuery and OrQuery throughput is 2.1 and 1.9 times over MySQL Cluster with 90 million records dataset.

The future work includes further optimization and evaluation the space overhead and reliability in terms of recovery time. Additional work should be done to optimize the index updating performance. Some real world application benchmarks should be involved to evaluate the query performance in real world scenarios. More practical experiences and lessons should be given.

Acknowledgment

We would like to thank Vega GOS R&D team members and GOS users, especially Liang Li to help optimizing the query plan. This work is supported in part by the Hi-Tech Research and Development (863) Program of China (Grant No. 2006AA01A106, 2006AA01Z121, 2009AA01A130), and the National Basic Research (973) Program of China (Grant No. 2005CB321807).

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. In: 7th USENIX Symposium on Operating Systems Design and Implementation. vol. 7, pp. 205--218. USENIX Association, Berkeley (2006)
2. 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. In: Proc. VLDB Endow. vol. 1, pp. 1277--1288. VLDB Endowment (2008)
3. Vigfusson, Y., Silberstein, A., Cooper, B.F., Fonseca, R.: Adaptively parallelizing distributed range queries. In: Proc. VLDB Endow. vol. 2, pp. 682--693. VLDB Endowment (2009)
4. Cafarella, M., Chang, E., Fikes, A., Halevy, A., Hsieh, W., Lerner, A., Madhavan, J., Muthukrishnan, S.: Data management projects at Google. SIGMOD Rec. 37(1), 34--38 (2008)
5. Apache Hbase project, http://hadoop.apache.org/hbase
6. Yang, H.C., Parker, D.S.: Traverse: Simplified Indexing on Large Map-Reduce-Merge Clusters. In: DASFAA 2009. LNCS, vol. 5463, pp. 308--322. Springer, Heidelberg (2009)
7. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. Acta Informatica. 1(3), 173--189 (1972)
8. Comer, D.: Ubiquitous B-Tree. ACM Computing Surveys (CSUR). 11(2), 121--137 (1979)
9. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proceedings of the 1984 ACM SIGMOD international conference on Management of data. vol. 13, pp. 47--57. ACM, New York (1984)
10. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R^+-tree: A dynamic index for multi-dimensional objects. In: Proceedings of the 13th International Conference on Very Large Data Bases. vol. 13, pp. 507--518. Morgan Kaufmann, San Francisco (1987)
11. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A., Zhou, L.D.: Boxwood: abstractions as the foundation for storage infrastructure. In: Proceedings of the 6th USENIX on Symposium on Operating Systems Design and Implementation. vol. 6, pp. 105--120. USENIX Association, Berkeley (2004)
12. Padmandabhan, S.,Bhattacharjee, B., Malkemus, T., Cranston, L., Huras, M.: Multi-dimensional clustering: a new data layout scheme in DB2. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. vol. 32, pp. 637--641. ACM, New York (2003)
13. DeWitt, D.J., Gerber, R.H., Graefe, G., Heytens, M.L., Kumar, K.B., Muralikrishna, M.: GAMMA - A High Performance Dataflow Database Machine. In: Proceedings of the 12th International Conference on Very Large Data Bases. vol. 12, pp.228--237. Morgan Kaufmann, San Francisco (1986)
14. Fushimi, S., Kitsuregawa, M., Tanaka, H.: An Overview of The System Software of A Parallel Relational Database Machine GRACE. In: Proceedings of the 12th International

Conference on Very Large Data Bases. vol 12, pp. 209--219, Morgan Kaufmann, San Francisco (1986)
15. Ronström, M., Oreland, J.: Recovery principles of MySQL Cluster 5.1. In: Proceedings of the 31st international conference on Very large data bases. vol. 31, pp. 1108--1115. VLDB Endowment (2005)
16. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 149--160. ACM, New York (2001)
17. Sylvia, R., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. pp. 161--172. ACM, New York (2001)
18. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In: Guerraoui, R. (eds) Middleware 2001. LNCS, vol. 2218, pp. 329--350. Springer-Verlag, Heidelberg (2001)
19. Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California at Berkeley. (2001)
20. Cai, M., Frank, M., Chen, J., Szekely, P.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. Journal of Grid Computing. 2, 3--14 (2004)
21. Albrecht, J., Oppenheimer, D., Vahdat, A., Patterson, D.A.: Design and implementation trade-offs for wide-area resource discovery. ACM Trans. Interet Technol. 8(4), 1--44 (2008)
22. Apache Hadoop project, http://hadoop.apache.org
23. TPC Benchmark C, www.tpc.org/tpcc
24. TPC Benchmark H, www.tpc.org/tpch
25. Nagios project, http://www.nagios.org