Fast Multi-fields Query Processing in Bigtable Based Cloud Systems

Haiping Wang, Xiang Ci, and Xiaofeng Meng

School of Information, Renmin University, Beijing, China
{wanghaiping1022,cixiang,xfmeng}@ruc.edu.cn

Abstract. With the rapid increase of data sizes, enterprise applications are migrating their backend data management and analytic systems into cloud based data management systems.Bigtable is among one of the major data models used by cloud storage systems as their storage layer. Such systems provide high scalability and schema flexibility, and support efficient point and range based queries based on rowkeys. However, Bigtable based systems have limited support on non-rowkey based queries and multiple-fields based queries, due to much overhead on invoking extra scanning of data. In this paper, we develop a system *TNBGR*(Telecom Network Browsing Gateway Records) on managing and querying large scale telecommunication data. TNBGR is built on top of HBase and MapReduce, with a focus on optimizing multi-fields query processing. TNBGR provides a novel application and system resource aware data allocation strategy to minimize data access through multi-layer region partitioning, resource parameterization, and balanced region distribution.The query composition dynamically updates application parameters based on tracked system statistics and automatically translates queries for MapReduce. Through additional query optimization by improving region locality, TNBGR achieves high efficiency on supporting multi-field queries. The experimental results show that our solution improves the performance of the queries by about 5 and 18 times respectively.

Keywords: HBase, Bigtable, Multiple-Fields Query Processing.

1 Introduction

Enterprise companies traditionally rely on RDBMS or parallel DBMS based solutions to manage and query large datasets in their business applications. For example, telecommunication companies in China keep users' CDR(Called Detailed Records) data in data management systems.To facilitate illustration, in the rest of this paper, we use a table named R to represent the schema of the CDR data. The schema of table R is simplified as:

$$R(msisdn, url, ts, size, otherdata)$$

Here $msisdn$ is the user's phone number. url is the target web site and ts is a timestamp when the user start this accession. $size$ records the size of data

J. Wang et al. (Eds.): WAIM 2013, LNCS 7923, pp. 142–154, 2013.

traffic. All the other related data is stored in column *otherdata*. Based on the government's regulations, telecom companies only need to store latest access history data within a period of time, i.e., 90 days. Many applications can be built based on these CDR data. Following are two typical scenarios:

Q1: *For a given user, find the top N web sites the user has accessed and the corresponding network traffic during a period of time.*
Q2: *Return top N users who had accessed a web site during a period of time and the aggregated network traffic for each user.*

In previous solution, all the CDR data was stored in a distributed relational database and queries are executed in SQL. However, nowdays more and more users surf the Internet via their smart phones instead of their computers. The scale of access log data is growing rapidly into 100TB level and still keeps on increasing. Enterprise companies start to migrate their background data into cloud data management systems such as HBase[5] or others and use MapReduce[2] to accelerate query processing. Some of such systems use Bigtable[4] data model in their underlying storage layer, referred as *Bigtable based cloud systems* in this paper.

Bigtable based cloud systems such as HBase support row key based point query and range query efficiently. However, for queries based on multiple fields or on non-rowkey field, many redundant records have to be scanned, which could lead to unsatisfactory performance.In this paper, we present a cloud based data management and query system *TNBGR*(Telecom Network Browsing Gate way Records). We build TNBGR by extending and optimizing from HBase and MapReduce [2], to provide highly efficient query support of typical multiple-fields queries for telecom applications.

The rest of this paper is organized as follows. Related work is discussed in Section 2. Section 3 presents our data allocation strategy and implementation approaches. Query processing is discussed in Section 4, including the query processing workflow, and query decomposition. Section 5 discusses additional query optimization techniques. Performance study is discussed in Section 6, followed by conclusion.

2 Related Work

2.1 Query Optimization through MapReduce

MapReduce is a software framework introduced by Google to support distributed computing on large data sets on clusters of computers. Computational processing can occur on data stored either in a unstructured filesystem or in a structured database. It has two steps named *map* and *reduce* and allows for distributed processing of the map and reduction operations. To accelerate multiple-fields query in Bigtable based cloud systems like HBase, some previous effort was made by using MapReduce job to do parallel query processing.

When MapReduce job is used to do multiple-fields query processing on Bigtable based cloud systems, each *maptask* does sub query on one tablet and query

parallelism is achieved through MapReduce. MapReduce tasks can fetch data directly from the distributed file system or from the database layer, but redundant data scan can not be avoided.

2.2 Query Optimization through Indexes

Another way to improve the efficiency of multiple-fields query on Bigtable based cloud systems is through using indexes. According to the data structures, we can divided the indexes into two categories: one-dimensional indexes and multiple-dimensional indexes.

ITHBase[6], IHBase[9], CCIndex[11] and Asynchronous views[1] are four representative related work that use several one-dimensional indexes to accelerate multiple-fields query processing. For each column that is frequently used by user queries, a one-dimensional index was build on it.

RT-CAN[8], QT-Chord[3], EMINC[10] and A-Tree[7] are four types of multi ple-dimensional indexes suitable for cloud data management systems. RT-CAN integrates CAN based routing protocol and the R-tree based indexing scheme to support efficient multi-dimensional query processing in a Cloud system. QT-Chord integrates Quad trees and Chord protocol together. EMINC[10] uses K-d tree as its local index and R-Tree as its global index. A-Tree[7] is one types of R-tree with bloom filter. The upper four indexes can support multiple-fields query very well with good scalability. But the size of index data should be very large, and the cost of maintaining and updating these indexes is very high too.

3 Data Allocation Strategy

One approach to accelerate the efficiency of a query is to reduce the candidate data that will be accessed during the query procedure. In this paper, we try to reduce the candidate data during the query. To achieve this, we propose a novel data allocation strategy named MDRO algorithm (Multiple-Dimensional Region Organization Algorithm). MDRO algorithm consists of three parts: fields selection, region organization strategy and row key generation algorithm.

3.1 Fields Selection

Queries based on the row key are normally efficient in HBase. To provide efficient queries, we should try to transform our queries into queries based on the row key. As we mentioned above ,there are two typical queries: **Q1**and **Q2**, so the query constraints are based on three fields: user's phone number *msisdn*, the *url* of the web site and access timestamp *ts*. Thus we would like to use *msisdn*, *url* and *ts* for row key generation.

However, not all fields related with query constraints are suitable for row key generation. We generalize three rules to classify fields based on suitability for used as row key, discussed below.

- *Rule 1: Identification.* The selected fields should uniquely identify a row key. This rule is due to the uniqueness requirement of a row key.
- *Rule 2: Usefulness for queries.* This is because in Bigtable based cloud systems the efficiency of query based on row key is much better than that on non row key. So we should select as many fields as possible for those which are frequently used with query constraints.
- *Rule 3: Conducive to data and access workload distribution.* This rule is based on the consideration of system workload balance.

Based on the upper three rules, we chose fields *msisdn* and *ts* for row key generation in our solution for CDR data storage.

3.2 Region Organization

In HBase, each big table is partitioned into a lot of regions. Most data maintenance and management operations are based on regions. The organization of regions affects the efficiency of data insert, delete, update and queries. We propose to organize the regions into a multiple layer grid tree (MLGT). The achitecture of MLGT is shown in Fig. 1:

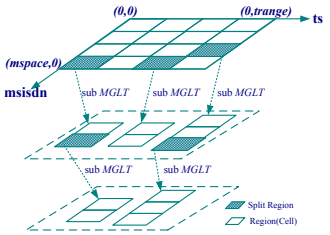


Fig. 1. Multiple Layer Grid Tree(MLGT)

In $MLGT$, regions are firstly organized into a two dimensional grid with two axes: ts as the horizontal axis and $msisdn$ as the vertical axis. The domain of ts is initially partitioned into N parts while $msisdn$ is partitioned into M sub ranges. Each cell in $MLGT$ maps to a region. As data is inserted into the table, the size of a region may reach the threshold, then the corresponding region will be split into a sub grid. By changing the parameter M and N, we can minimize region splits to reduce the depth of $MLGT$.

In our solution, for each sub grid in $MLGT$, the value of N is always equal to 1 (the reason will be explained later in discussions on row key generation algorithm). We always count the work load of data insert operations for each region, and the value of parameter M is decided by the work load statistics of the given region. The data structure of $MLGT$ can be viewed as follows:

$$
\begin{aligned}
&Typedef\ struct\ MLGT\{\\
&\quad int\quad N;\\
&\quad int\quad M;\\
&\quad bool\quad bm[m][n];\\
&\quad long\quad inserts[m][n];\\
&\quad long\quad trange;\\
&\quad long\quad mspace;\\
&\quad Map < RegionID, MLGT > SR;\\
&\}MLGT
\end{aligned}
$$

The variables are described below:

- N: *int* type, which represents the number of splits happened for the grid's total time range, and the granularity of time is millisecond.
- M: *int* type, which represents the number of splits for the grid's total $msisdn$ space.
- $bm[m][n]$: *boolean* type, which represents a two-dimensional array, with size of $M * N$. if $b[m][n] = true$, it means that the region in the mth row and nth column of the given $MLGT$ has been split and has a key/vlaue pair in map SR. The key of the key/value pair is the $RegionID$ and the value is the root node of its sub multiple layer grid tree.
- $insert[m][n]$: *long* type, which represents a two-dimensional array, with size of $M * N$. $insert[m][n]$ stores the statistics of insert workload for the corresponding region.
- $trange$: *long* type, which represents the total number of possible ts for the given multiple layer grid tree.
- $mspace$: *long* type, which represents the total number of possible $msisdn$ for the given multiple layer grid tree.
- SR: a map, which maps a region which has been split before to the root node of its child multiple layer grid tree.

In our solution, $RegionID$ is the *startkey* of a given region. Thus when a region is split, the $RegionID$ of the parent region keeps the same to make it convenient to track split regions.

Now we will discuss the value of M and N. For the root layer of each table's $MLGT$, the initial value of M and N are affected by the following eight parameters:

- $Sizeof(R)$: an estimated value of the size of the table R.
- $NodeNumber$: the number of nodes of the cluster.
- $RegionTruncateTime$: the time for the cluster to drop and recreate a region.
- $SystemBearableTime$: the time duration that the upper applications can bear for the cluster to be out of service.
- $ConcurrencyDegree$: the number of regions that each node can concurrently accesses.
- $RegionSize$: the size of a region.
- $Domain(ts)$: the interval threshold of the ts column.
- $Domain(msisdn)$: the interval threshold of the $msisdn$ column.

Suppose the replication factor of the underlying Hadoop distributed file system is 3. Then the value of M and N are initialized by following five rules:

Rule 1: The value of $M \times N$ should be close to the cluster's finally region number for table R.

$$M \times N \dashrightarrow \frac{Sizeof(R)}{Regionsize} \tag{1}$$

Rule 2: Region truncating time has to be less than system bearable down time.

$$RegionTruncateTime \times M < SystemBearableTime \tag{2}$$

Rule 3: The system node number multiplying the degree of concurrency should be large than the number of splits for the grid's total $msisdn$ space.

$$NodeNumber \times ConcurrencyDegree > M \tag{3}$$

Rule 4: $Domain(ts)$ must be evenly divisible by N.

$$Domain(ts)\ \%\ N == 0 \tag{4}$$

Rule 5: $Domain(msisdn)$ must be evenly divisible by M

$$Domain(msisdn)\ \%\ M == 0 \tag{5}$$

3.3 Row Key Generation

In this section, we will discuss the details on how to generate a row key suitable for the region organization algorithm with a given $msisdn$ and ts that are selected out by the fields selection rules. In this paper, to maximize the system's performance, we design the row key generation algorithm by following constraints below:

- The row key must be unique.
- The row key generation function should be conductive to data distribution.
- Given a $msisdn$, the row key function is a time-based continuous function, which can make $Q1$ more efficient.
- To accelerate $Q2$, we should try to support range query without $msisdn$.
- When a region is split, the row key of all the records in the region keeps the same.

Based on the above constraints, we propose our row key generation algorithm. The row key of a given CDR record is decided by three parameters: $msisdn$, ts and the $MLGT$ of its corresponding table. Equation 6 is the row key generation function:

$$rowkey = key(msisdn, ts, MLGT) \tag{6}$$

Algorithm 1 is the actual body of the row key function, and each row key consists of two parts: $basekey$ and $offset$. The $basekey$ is the $startkey$ of its corresponding region while $offset$ is the offset. The value of $basekey$ can be calculated by function $Base(m, n, tr, mr, N)$ while function $Offset(m^{'}, n^{'}, tr)$ is used to generate the value of $offset$.

$$Base(m, n, tr, mr, N) = (m * N + n) * tr * mr \tag{7}$$

$$Offset(m^{'}, n^{'}, tr) = m^{'} * tr + n^{'} \tag{8}$$

Algorithm 1: row key generation algorithm
Input: $msisdn$, ts,$MLGT$
Output: $rowkey$
Variables:
MLGT: the root multiple layer grid tree
msisdn: user's phone number
ts: timestamp
Procedure:
1: $rowkey, basekey, tmpkey, offset = 0$;
2: $mr, tr, m, n = 0$; $tmpts = ts$;
3: $tepmsisdn = msisdn$; $flag = false$;
4: **do**
5: $mr = MLGT.mspace/MLGT.M$;
6: $tr = MLGT.trange/MLGT.N$;
7: $m = tmpmsisdn/mr$;
8: $n = tmpts/tr$;
9: $tmpkey = Base(m, n, tr, mr, MLGT.N)$;
10: $basekey = basekey + tmpkey$;
11: $tmpmsisdn = tmpmsisdn\%mr$;
12: $tmpts = tmpmts\%tr$;
13: **if** $(MLGT.bm[m][n])$
14: $flag = true$;
15: $MLGT = MLGT.SR.get(basekey)$;
16: **else** $flag = false$;
17: **while** $(flag)$
18: $offset = Offset(tmpmsisdn, tmpts, tr)$;
19: $rowkey = basekey + offset$;
20: **Return** $rowkey$;
End Procedure:

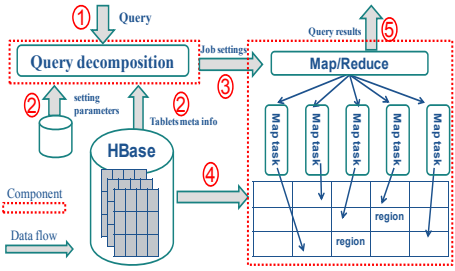


Fig. 2. Query work flow

Note that the row key generated by algorithm 1 can not comply with constraint 5. To solve this contradiction, we set the value of N for each $MLGT$ equal to 1 except that the $MLGT$ is the root node of the given table's multiple layer grid tree. This additional parameter setting can help us make sure the row key of all the records in a region keep the same even if when the region got split.

4 Query Processing

4.1 Query Work Flow

As showed in Fig. 2, the query work flow consists of two parts: the query decomposition component and the MapReduce component. The query decomposition component is response for generating the query plan, while the MapReduce component translates the query plan to MapReduce jobs to execute the query.

There are five steps to finish a query submitted by a client:

Step 1: The client submits a multiple-fields query request.

Step 2: The query decomposition component will set parameters, including parameters for tables and meta info for regions. The setting parameters include the table name stored in HBase, and parameters for the MLGT of the given table.

Step 3: The query decomposition component sends these MapReduce job parameters to the MapReduce component. With these parameters, the MapReduce component creates a MapReduce job and submits it to the jobtracker.

Step 4: The fourth step executes the MapReduce job, where each map task accesses one candidate region.

Step 5: This step returns the final query results to the client.

4.2 Query Decomposition

Once the query decomposition component receives a query request, the first thing it does is to parse the query statement to collect the related fields and constraints. Then it will classify the fields into two sets: RFQ(Row key related Fields in the Query) and $NRFQ$(Non Row key related Fields in the Query). At the same time, it also divides the query constraints into two categories: constraints on fields that are elements in RFQ and constraints on fields that are elements in $NRFQ$. It finally determines the candidate regions in the query table's multiple layer grid tree.

To facilitate discussion, we use arabic numerals to identify the cells of table $R^{'}s$ $MLGT$. As showed in Fig. 3, region 12 has been split into 3 regions: $21, 22, 23$. Region 13 has been split into region 17 and 18 while region 15 is split into region 19 and 20. Note that these arabic numbers are not the $regionID$ of its corresponding region: the $regionID$ is the start key.

Before discussing the details of the query decomposition procedure, we will introduce another data structure named $RegionInputSplit$ used for packaging the query parameters for each MapTask, which stores the $regioninfo$, the candidate $startRow$ and $endRow$ of the corresponding region. The range [$startRow$, $endRow$] is a subset of the region's key range. The candidate number of $RegionInputSplit$ for each query, the $regioninfo$ and the range [$startRow$, $endRow$] of each $RegionInputSplit$ are decided by query constraints based on RFQ fields. Other query constraints based on fields which are elements of $NRFQ$ are packaged as filter instance and used for the $map()$ function to do additional filtering operation if necessary. The data structure of $RegionInputSplit$ can be viewed as follows:

$$
\begin{aligned}
&Typedef\ struct\ RegionInputSplit\{\\
&\quad Filter\quad filter;\\
&\quad HRegionInfo\quad regioninfo;\\
&\quad key\quad startRow;\\
&\quad key\quad endRow;\\
&\}RegionInputSplit
\end{aligned}
$$

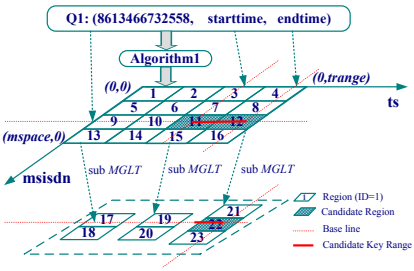


Fig. 3. Query decomposition for Q1

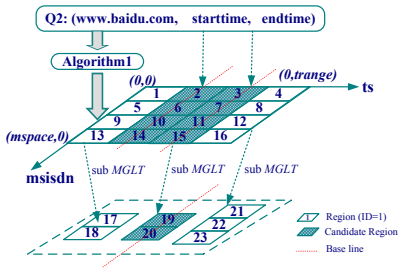


Fig. 4. Query decomposition for Q2

For $Q1$, as showed in Fig. 3, RFQ is $\{msisdn, ts\}$, and $NRFQ$ is an empty set. There are three query constraints on field ts and $msisdn$: $starttime < ts < endtime$ and $msisdn = 8613466732558$; With the two query constraints on field ts we can know that the candidate regions are cells in the third and fourth columns of the gird (marked with imaginary red line). What's more, based on the query constraints on $msisdn$ field, we can derive that the candidate regions for $Q1$ are region 11 and 12, since region 12 has been split into region $21, 22, 23$. With the child multiple layer grid tree of region 12 and the given $msisdn$(8613466732558), the final candidate regions for $Q1$ are region 11 and 22. The detailed procedure of query decomposition for $Q1$ can be seen in Fig. 3.

Since all the query constraints of $Q1$ are based on the fields in RFQ, for each record delivered from the record reader during the map phase, if it matches the query constrains, the $map()$ function does not need to perform any filtering operation and only needs to perform a region level aggregation operation.

For $Q2$ showed in Fig4, RFQ is $\{ts\}$, and $NRFQ$ is $\{url\}$. There are two query constraints on field ts. The first is $starttime < ts < endtime$. This can help us to find out the candidate regions of query $Q2$ are region $2, 3, 6, 7, 10, 11, 14, 15$. Here region 15 has already been split into two sub-regions: region 19 and region 20.For the query decomposition component to locate the candidate regions without query constraint on $msisdn$, we need to access all the tuples of each candidate region, i.e., tuples in the region's key range $[startKey, endKey]$ have equal query range $[startRow, endRow]$ of each $RegionInputSplit$. The constraint $url =$ "$www.baidu.com$" will be packaged as a filter instance into each $RegionInputSplit$. The $map()$ function in the map phase of the MapReduce is responsible for verifying whether the record matches the query constraint $url =$ "$www.baidu.com$". The procedure is expressed in Fig. 4.

For both query $Q1$ and $Q2$, after selecting out all the candidate regions, the query decomposition component will query the root region with the given table name to get the meta info of these candidate regions such as the location of each region(the region server that manages the region).

Finally the query decomposition component will generate a query plan and transform the query plan into a MapReduce job, where each map task accesses one candidate region and performs a region-level aggregation and shuffles the result to the reducer.

5 Query Optimization

5.1 Region Localization

Since the network transmission overhead affects the query efficiency, one way to reduce the query time is to reduce the network overhead: the RPC (Remote Procedure Call Protocol) time between datanodes and regionservers and the RPC time between regionservers and tasktrackers.

In *TNBGR*, we modify the region allocation algorithm in HBase to try to assign as many regions as possible in the nodes which hold most of the corresponding chunks. First, we set the region size multiple times bigger than the size

of each chunk in the underlying HDFS system. Then we assign the region to the physical node which holds most replicas of its corresponding chunks periodically.

5.2 Parallel Aggregation

Since both $Q1$ and $Q2$ are aggregate queries, the filter operations for different records are independent with each other. In TNBGR, multiple threads are introduced to aggregate the records in parallel. For each *maptask*, one single *reader thread* is introduced to read all the records from HDFS and has them cached in the buffer at first. Then multiple *query threads* filter the records in parallel, with each thread accessing one segment. Finally the *main thread* of the *maptask* collects all the results from *query threads* and performs a region level aggregation and shuffles the region level aggregation result into the reduce side.

6 Experiment Study

We present a detailed evaluation of our solution. We implement our system using HBase 0.90.2 and Hadoop 0.20.2. We evaluate the trade-offs associated with the different implementations for the storage layer and compare our solution with the base line solution. In the base line solution, the row key also consists of ts and $msisdn$, but the row key generation function is very simple:

$$key(msisdn, ts) = ts.tostring() + msisdn.tostring() \tag{9}$$

The row keys are sorted in alphabetical order and regions are organized into a linear row key space set by default by HBase and have not been pre-split. A region is split into two child regions in the middle of its row key space when its size reaches the region size threshold.

To finish $Q1$, for each possible ts value in range $(starttime, endtime)$, with the given value of $msisdn$ and the row key generation algorithm as showed in equation 9, one target row key is calculated. These discrete target row keys are then packaged into different *inputsplits* for MapReduce jobs according to the regions they belong to. Query $Q1$ can be finished without any unnecessary data scan. But for $Q2$, a lot of unnecessary data needs to be scanned, since for each t, all the records with ts equal to t will be accessed.

6.1 Experiment Setup

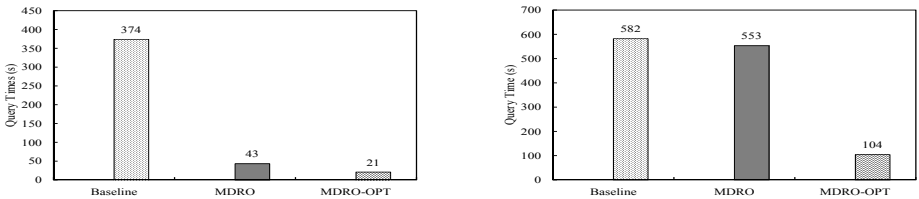
Our testing infrastructure includes eight machines which are connected together to simulate cloud computing platforms. One node is used as a master node and the other seven nodes are used as slave nodes. Each node contains two Intel Quad-Core 2.4GHz CPU, 16GB of main memory and 7 TB hard disk. The OS is Ubuntu 10.10, and the network communication bandwidth is 1Gbps.

The experiment data set is telecom CDR(Called Detail Record) data and the schema of table R is

$$R(msisdn, url, ts, size, otherdata)$$

During the experimental phase, R has two column families named FD(frequently accessed data) and OD(other data). $msisdn$, url, ts, $size$ are four qualifiers of FD in HBase. The size of table R is 1 TB with 2 billion records, and each record has 512 bytes. The $msisdn$ is a long key with value range of [8610000000000, 8619999999999] and ts is a randomly generated long value with the value in range of $[\,0, 90 * 24 * 3600 * 1000]$. The value of url for each tuple in zipfian distribution. The query time duration is the first two days of the 90 days. The size of each region is set to $512MB$. The parameters *mapred.tasktracker.map.tasks.maximum* and *mapred.tasktracker.reduce.tasks.maximum* are both set to 7.We conduct two types of experiments:

- Performance evaluation experiment. Compare the query time of $Q1$ and $Q2$ with the baseline solution, and the query time based on our data allocation strategy before optimization and after optimization.
- Performance on impact parameters. By changing the initial value of M and N, we evaluate the query time of $Q1$ and $Q2$ by tuning the settings of the initial value of M and N.



(a) Q1(msisdn + ts) (b) Q2(url + ts)

Fig. 5. Performance evaluation result

In our performance evaluation experiment, the initial value of M for the root layer grid of table $R^{'}s$ and $MLGT$ is set to 200 and N is set to 45, thus the total number of regions of table R is 9000. For query $Q1$, the number of candidate regions is 2, and for query $Q2$, the number of candidate regions is 400. The experimental results demonstrated in Fig. 5 show significant performance improvement for both queries with our solution. For $Q2$, the query time is reduced to about 20% of the query time from the baseline solution, and for $Q1$,the query time is reduced to 5.6% of the query time from the baseline solution.For $Q2$, our query optimization techniques provide major improvement of the performance.

Fig. 6 shows the settings of impact parameters used in the experiment study. The initial value of N for the first layer of $R's$ $MLGT$ is scaled up from 30 to 180 while M consist of values of $\{50, 100, 200, 300, 400, 500\}$. The query constraints are the same as the performance evaluate experiment. After query composition, the number of candidate regions (the number of map tasks for the corresponding MapReduce job of the query) can be seen in Table 1. The term MR means the candidate regions for query $Q1$ (msisdn based) and UR means candidate regions for query $Q2$ (url based). Fig. 7 and 8 show the query time of $Q1$ and $Q2$.

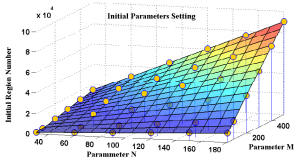


Fig. 6. Initial Parameter Settings

Table 1. Candidate Regions

N	30		60		90		120		150		180	
M	MR	UR	MR	UR	MR	UR	MR	UR	MR	UR	MR	UR
50	1	50	2	100	2	100	3	150	4	200	4	200
100	1	100	2	200	2	200	3	300	4	400	4	400
200	1	200	2	400	2	400	3	600	4	800	4	800
300	1	300	2	600	2	600	3	900	4	1200	4	1200
400	1	400	2	800	2	800	3	1200	4	1600	4	1600
500	1	500	2	1000	2	1000	3	1500	4	2000	4	2000

From Table 1 and Fig. 7, we discover that the query time of $Q1$ is mainly influenced by the value of N: as the value of N increases, the query time decreases. This is because as the value of N increases, the key range for each candidate region decreases, and it reduces the execution time of each map task. Although the parameter N influences the number of map tasks of the corresponding job, the number of map tasks does not reach the threshold of map slots that can be executed simultaneously in the cluster(49 in our cluster). We can conclude that once the value of N reaches the threshold, the query time will increase as well.

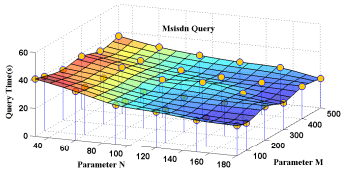


Fig. 7. Msisdn Query

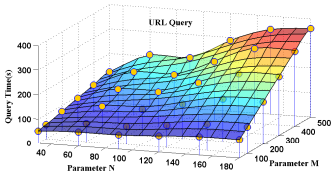


Fig. 8. URL Query

From Table 1 and Fig. 8, we find that the query time of $Q2$ are influenced by the values of both N and M: as the values of M and N increase, the query time increases too. As the values of N and M increase, although the size of each region deceases and reduces the execution time of each map task, the number of map tasks of the corresponding job grows so quickly. This will cause each map slot to execute several map tasks, and the time for map tasks initialization and scheduling is much longer than the execution time of each map task. Thus, the query time is mainly decided by the number of map tasks that each map slot should execute.

7 Conclusions

In this paper, we present *TNBGR*, a system developed and optimized based on HBase and MapReduce to support multiple-field based queries for telecom applications in China. The experiment study shows significant performance advantage of our system. *TNBGR* can support major real world applications with telecom

CDR data, and our system is being deployed with a major telecommunication company in China.

Acknowledgments. This research was partially supported by the grants from the Natural Science Foundation of China (No.61070055, 91024032, 91124001);the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University(No. 11XNL010); the National 863 High-tech Program (No. 2012AA010701, 2013AA013204).

References

1. Agrawal, P., Silberstein, A., Cooper, B., Srivastava, U., Ramakrishnan, R.: Asynchronous view maintenance for vlsd databases. In: SIGMOD 2009, pp. 179–192. ACM (2009)
2. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
3. Ding, L., Qiao, B., Wang, G., Chen, C.: An efficient quad-tree based index structure for cloud data management. In: Wang, H., Li, S., Oyama, S., Hu, X., Qian, T. (eds.) WAIM 2011. LNCS, vol. 6897, pp. 238–250. Springer, Heidelberg (2011)
4. Chang, F., Dean, J., Ghemawat, S., et al.: Bigtable: A distributed storage system for structured data. In: OSDI 2006, pp. 205–218 (2006)
5. Kellerman, J.: Hbase: Structured storage of sparse data for hadoop (2009), `http://hbase.apache.org/`
6. Kennedy, J.: Ithbase (2012), `https://github.com/hbase-trx/hbase-transactional-tableindexed`
7. Papadopoulos, A., Katsaros, D.: A-tree: Distributed indexing of multidimensional data for cloud computing environments. In: CloudCom 2011, pp. 407–414 (2011)
8. Wang, J., Wu, S., Gao, H., Li, J., Ooi, B.: Indexing multi-dimensional data in a cloud system. In: SIGMOD 2010, pp. 591–602. ACM (2010)
9. ykulbak. Ihbase (2012), `https://github.com/ykulbak/ihbase`
10. Zhang, X., Ai, J., Wang, Z., Lu, J., Meng, X.: An efficient multi-dimensional index for cloud data management. In: CloudDB 2009, pp. 17–24. ACM (2009)
11. Zou, Y., Liu, J., Wang, S., Zha, L., Xu, Z.: CCIndex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries. In: Ding, C., Shao, Z., Zheng, R. (eds.) NPC 2010. LNCS, vol. 6289, pp. 247–261. Springer, Heidelberg (2010)