2013 IEEE Sixth International Conference on Cloud Computing

HGrid: A Data Model for Large Geospatial Data Sets in HBase

Dan Han, Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
{*dhan3, stroulia*}*@ualberta.ca*

***Abstract*—Cloud-based infrastructures enable applications to collect and analyze massive amounts of data. Whether these applications are newly developed or they are being evolved from existing RDBMS-based implementations, NoSQL databases offer an attractive platform with which to address this challenge. However, developers find it difficult to effectively manage data in NoSQL databases, because these platforms do not offer much support for data organization. Since poor data organization may abuse the features of the NoSQL database and result in unsatisfactory performance, developing a systematic method for NoSQL database data-schema design is a timely and important problem.**

In this paper, we focus on geospatial applications, as a family of big-data systems with distinct data types and usage patterns, in need of scalability. We propose the *HGrid* data model for HBase, based on a hybrid index structure, combining a quad-tree and a regular grid as primary and secondary indices correspondingly. We have comparatively evaluated the performance of *HGrid* with uniform and skewed data, against two other data models based on quad-tree and regular-grid indices. Our results demonstrate that *HGrid* scales well and supports efficient performance for range and k-nearest neighbor queries. Although this model does not outperform all its competitors in terms of query response time, it is more flexible for discontinuous and skewed space, and its index requires less space than the corresponding quad-tree and regular-grid indices, which makes its deployment possible with less resources. Through this study, we also formulate a set of guidelines on how to organize data for geospatial applications in HBase.

***Keywords*-Data Model; Data Schema; Geospatial Data Set; HBase; Coprocessor**

I. Introduction

With the explosive increase of location-aware devices (GPS-enabled smartphones and vehicles, RFIDs, tablets, etc) and the proliferation of sensor-based systems, location-based services that contextualize the user experience are growing. A prominent example of this phenomenon is location-aware advertisement and recommendation, where the user is provided with advice on real-service opportunities close to her. Taking advantage of these location-aware services are millions of users, who continuously register their location updates through their wireless providers. In addition to user-facing services, smart systems embed sensors and activators in our environment for monitoring and management; these systems also generate massive amounts of data updates and rely on analyzing this data over time and across space.

The challenge with these applications is how to guarantee satisfactory performance for real-time analysis, while at the same time, supporting millions of location updates per minute. To address these requirements, database-management systems (DBMS) must scale up while maintaining good load balancing and high up-time [1]. As most typical queries of these applications involve the retrieval of multi-attribute values related with some proximity function to a given geographic location, efficient multi-dimensional geospatial data access is also an important requirement. Relational database-management systems (RDBMSs) support efficient spatial queries with special-purpose index structures, such as K-d tree [2], quad-tree [3] and R-tree [4]. However, RDBMSs are challenged by the scaling requirements of this new breed of applications, requiring complex hardware setup and configuration. NoSQL (Not Only SQL) databases, endowed with availability, elasticity and scalability through their easy deployment on cloud-computing platforms, become a more attractive solution for these applications [5]. However, data in NoSQL databases is stored in an unstructured manner, based on a primary key and attributes organized in column families. Even though some rough guidances about schema design have been provided by the specific NoSQL database offerings, such as HBase [10], there is still no systematic method for how to actually design the structure of the "NoSQL Big Tables" for a particular application. The data organization has a great impact on the performance of the queries implemented on these tables, and therefore, *a systematic method for NoSQL data-schema design for geospatial applications* is a timely and important problem in this area.

This is exactly the problem we aim to address with our work on HBase, the open-source implementation of BigTable [6]. To that end, we have developed the *HGrid* data model for organizing geospatial data sets, a hybrid two-tier index structure, tailored to the HBase three-dimensional storage mechanism. The primary index is a quad-tree that divides the data space into rectangular tiles, and encodes each tile according to a Z-ordering traversal [7]. Next, a regular-grid index structure is used to divide each quad-tree tile into a sequence of contiguous rectangular cells. Each cell is assigned a unique identifier, constructed as the concatenation of the cell's *row index* and *column index* in a

978-0-7695-5028-2/13 $26.00 © 2013 Crown Copyright
DOI 10.1109/CLOUD.2013.78

grid. In this data model, the *row key* of each data point is the concatenation of the *z-value of the quad-tree tile* in which the data point belongs, and the *row index of its regular-grid cell* in the second-tier regular-grid index. The *column name* is constructed by concatenating the *column index of its regular-grid cell* and the *object id*. Finally, the various attributes of each object are stored in *the third dimension*.

We empirically evaluated the performance of this data organization with two synthetic data sets, with uniform and skewed data distribution correspondingly. Compared against a pure quad-tree data model and a pure regular-grid data model, we found that *HGrid* can be flexibly configured for a range of cell sizes, and although it exhibits slightly poorer performance than the regular-grid data model, its index requires less space than the corresponding quad-tree and regular-grid indices, which makes its deployment possible with less resources. It is more scalable and suitable for homogeneously covered and discontinuous spaces.

The rest of the paper is organized as follows. Section II reviews the background of this work on geospatial data, multi-dimensional index structures, linearization methods, HBase and introduces related works of geospatial data studies. By comparing with quad-tree against the regular-grid data model, we describe the *HGrid* data model and evaluate it with range query and k-nearest neighbor query in Section III. Section IV reports the experiment result with different data distributions under different data models and summarizes a set of suggestions about HBase schema design and query implementation. We conclude our contributions and future work in Section V.

II. Background and Related Work

The data points in geospatial data sets are typically multi-dimensional, including their coordinates (latitude and longitude), a timestamp, and a description (identifier and attributes) for the domain object at the data point [1]. Range queries (identifying the data points within a radius from a given location) and k-nearest neighbor queries (identifying the k data points closest to a given location) are the most common queries on these data sets.

A. Multi-Dimensional Indices

Spatial data are typically organized using "space-driven" or "data-driven" indices. In data-driven structures, such as R-tree [4], the distribution of the objects to be stored determines the partitioning of space. Since the most common queries in geospatial applications are typically based on locations, in our work we focus on space-driven approaches to data organization. An example of "space-driven" organization is a grid where objects are associated with a grid cell based on their position in the space, and an index of grid-cell identifiers enables rapid access. In this organization, the grid-based spatial index is created first and the data is added incrementally without causing any change to the index structure.

The ***regular grid*** is the simplest grid-based spatial index. It partitions a rectangular domain using rectangular cells of equal size [8]. An associated matrix, i.e., a two-dimensional array, maps each grid cell to the array of data points located within the space covered by the cell. The ***quad-tree*** recursively splits the space into subspaces organized in a search tree. Two methods are commonly used to split the given space [1]: the trie-based approach splits the space at the mid-point of a dimension, resulting in equal-size subspaces. The point-based technique splits the space in subspaces with equal number of data points [2]. Quad-tree is commonly coupled with space-filling curves [9] to linearise the subspaces. Z-ordering [7] is an easy-to-compute example of a space-filling curve.

B. HBase Storage Model Overview

HBase uses the Hadoop File System (HDFS) as its underlying data-storage platform. Unlike the two-dimensional tables of traditional RDBMSs, HBase organizes data in a three-dimensional cube. The basic data storage unit in HBase is a cell, which is identified with its *row key*, *column-family name*, *column name* and *version*[10]. Cells with multiple versions of data can be stacked in the third dimension. For example, the third dimension is used to stack the contents of message IDs in the Facebook messaging system [11]. At the physical level, each column family is stored contiguously on disk, and the data is physically sorted by *row key* , *column name* and *version*.

The HBase *Coprocessor* framework, inspired by Google's BigTable *Coprocessor* [6], provides a library and runtime environment for executing user code on the HBase region servers. Coprocessor implementations are executed remotely at the target region(s) hosted by region servers, and their execution results are returned to the client. This design decreases the communication overhead involved in transferring data from the region servers to the client, and enables dramatic performance improvement by pushing the computation to the server, where it can operate on the data directly. To reap the benefits of this framework, an appropriate partitioning of the data is necessary, which implies the need for a well-designed data schema. This is the reason why, in our work, we have focused on investigating the impact of different HBase table schemas on the performance of query execution using the *Coprocessor* framework.

To date, two proposals have been put forward for the organization of geospatial data in HBase. S. Nishimura et. al[1] built a multi-dimensional index layer on top of HBase to perform spatial queries. Ya-Ting Hsu et. al [12] presented a novel key-formulation schema, based on R+-tree for spatial index in HBase. Both studies investigate how to efficiently access the multi-dimensional data with spatial indices, which is part of the problem that we are addressing in this paper. Their methods demonstrate efficient performance with the spatial indices. However, they only focus on the design

of the HBase *row key* with no or little discussion about the *column* and *version* design. To design an appropriate data model for geospatial datasets, which can be easily and directly applied to geospatial applications, in addition to the *row key*, one need also take into account the design of the *column name* and the role of the third dimension. Furthermore, in our work, we implemented the queries with HBase *Coprocessor* to harness the parallelism benefits, while the above studies processed the queries with HBase *Scan*.

III. The *HGrid* Data Model for HBase

In this section, we first review the two data models underlying the design of *HGrid*. Next, we describe the *HGrid* data model and the index-construction process. Finally, we explain the implementation of two queries commonly used in location based service under this data model.

A. Preliminary Data Models

As we have already discussed, the *HGrid* data model for HBase is inspired by two simpler data models: the quad-tree data model and the regular-grid data model.

1) The Quad-Tree Data Model: The quad-tree data model relies on a trie-based quad-tree index, where Z-ordering [7] is applied to transform the two-dimensional spatial data into a one-dimensional array. In this model, the *row key*, which should be kept as short as possible, is the Z-value in decimal encoding, i.e., "0","1","2","3". The *column name* is the object ID, and each cell stores one data point in JSON format.

There are three performance concerns about this data model. First, as the quad-tree becomes deeper, the data points end up being organized into more rows. As a result, queries have to scan more grid cells in order to retrieve all data points within a range, or close to a location; at the same time, the more rows are scanned, the more unrelated-to-the-query data is accessed, causing performance deterioration. Moreover, the Z-ordering linearization technique, although appealing because of its simple computation, does not maintain good data locality, and subsequent grid cells are not necessarily close to each other in space. As a result, scanning more contiguous rows is even more likely to inspect irrelevant rows (i.e., rows corresponding to grid cells not sufficiently close to the query location), which increases the amount of accessed data and causes performance to suffer further. The last but not least issue is with the construction of the quad-tree. If the index is built in real time for each query, the construction cost dominates in small queries. If the index is maintained in memory, the granularity of the grid is limited by the amount of memory available, since the memory needed to maintain the index increases as the depth of the tree increases and the size of the grid cells becomes smaller.

2) The Regular-Grid Data Model: In the regular-grid data model, the *row key* is the row index of the cell in the grid, the *column name* is the column index of the cell, and each storage cell represents one object in JSON format holding all other attributes and values. The third dimension holds a stack of data points located in the same grid cell, and an index is maintained to keep the count of objects in each cell stack in order to support updates.

This index structure maintains data locality: data points close in the y dimension are likely to be in the same or neighboring rows and data points close in the x dimension are likely to be in the same or neighboring columns. With this data model, the data point can be determined efficiently by both of row and column and the unrelated data can be pruned with the Bloom filter.[1] However, in densely populated spaces, the number of objects in columns in each row increases. Because the time to retrieve a row with n data points more than doubles with n (when n is large) [12], the query performance will decrease. In addition, as there is no mechanism to filter the objects located in one column in this data model, more objects are retrieved in a query, which results in a higher number of false positives. A finer-grained grid would reduce the false positives but it would imply a larger cell-stack in-memory index, which may not be possible due to memory limitations. Given a certain amount of memory, this data model reaches a bottleneck when it comes to a large space with finer-grained grid cells.

Summarizing the relative advantages and disadvantages of the quad-tree and regular-grid data models, we note that the quad-tree data model is not efficient when it comes to large queries, as more irrelevant data must be scanned. The regular-grid data model is preferable in that respect because it has better localization and can provide very good pruning of the unrelated data. Query processing becomes inefficient in the regular-grid data model when it comes to large high-density spaces, as the amount of objects grouped in one row increases rapidly. Both the quad-tree data model and the regular grid data model are constrained by the size of available memory, in terms of how fine-grained the grid cell may become.

Considering the advantages and disadvantages of these two data models, we designed the *HGrid* data model. Using a two-level index structure, the *HGrid* data model avoids the regular-grid drawback by splitting large geographic spaces in tiles using a quad-tree index, and takes advantage of the localization feature of the regular-grid data model in the second-level index.

B. The HGrid Data Model Representation

Figure 1 diagrammatically depicts the *HGrid* data model. First, the space is divided into equally sized rectangular tiles

[1]The Bloom filter is a space-efficient probabilistic data structure to be used to check whether an element is a member of a set [13]. It is supported in HBase to reduce the disk lookups for unrelated rows or columns.

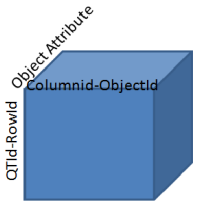

Figure 1. *HGrid* Data Model

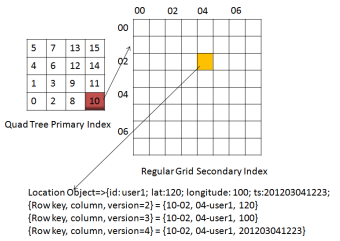

Figure 2. An example of *HGrid* Index

T, encoded with their Z-value. Next, the data points are organized in a regular grid of continuous uniform fine-grained cells. In this model, each data point is uniquely identified in terms of its *row key* and *column name*. The *row key* is the concatenation of the quad-tree Z-value and the regular-grid row index. The *column name* is the concatenation of the regular-grid column index and the object id of the data point. The attributes of the data points are stored in the third dimension.

Figure 2 illustrates with an example how the *HGrid* index is constructed. Given a specification of the overall space within which the geospatial data set is to be contained, the minimum boundary rectangle (MBR) of the space, i.e., the smallest rectangle that completely contains the space, is computed. The depth of the quad-tree is determined by the user-specified size of the tile. Each tile is associated with an index that corresponds to its rank according to the Z-ordering linearization. Each quad-tree tile contains all the data points whose coordinates belong in the space covered by the tile. Clearly, many data points may belong in the same tile and share the same tile code; there also may be empty tiles, with no data points at all. For the empty tiles, there are no relevant records stored in HBase.

Next, given the desired regular-grid resolution, the quad-tree tiles are decomposed into equally sized rectangular cells. Each cell is coded with the row and column index of the regular grid; this cell code becomes the secondary index for every data point in the cell.

There are two challenges in the configuration of this index-construction process: deciding (a) the appropriate granularity for tessellating the original space in quad-tree tiles, and (b) the appropriate granularity of the regular grid in the second stage. Based on our experience, we have found that the best resolution depends on the data distribution and the likely queries. The finer the quad-tree tile granularity, the more irrelevant rows will have to be pruned from the return set of *Scan* queries due to poor locality of Z-ordering. Alternatively, if direct *Get* access operations are used, more sub-queries will be required. Therefore, there is a trade-off between the size of tiles in the first stage and the size of cells in the second stage. Much experimentation with different levels of quad-tree and granularity of regular grid is needed in order to optimize the performance for a specific data set.

C. Query Processing

There are two ways for processing queries in HBase. Using a *Scan* operation, a number of rows corresponding to a range of row keys are retrieved and the response set is computed at the client-side. Using *Coprocessor*, partial response sets are computed in each region and are then aggregated at the client side.

Range queries are commonly used in location-based applications. Given the coordinates of a location and a radius, a vector of data points, located within a distance less than or equal to the radius from the input location, is returned. Relying on the *HGrid* data model and using *Coprocessor*, answering this query involves the following steps.
(1) Given the query input location and the range, the minimum bounding square that completely includes the implied circle around the input location is computed.
(2) Next, the quad-tree tiles that overlap with the computed bounding square are identified. The Z-codes of these tiles provide the primary index of the HBase rows of interest.
(3) Next, the overlap between the bounding square and each intersecting quad-tree tile is computed to identify the regular-grid cells involved. Based on these cells, the secondary index of the rows to be examined and the corresponding column indices become available.
(4) Having now computed the range of rows and columns involved in the query, a sub-query is issued for each selected tile or the parent tile of selected continuous tiles of the quad-tree and processed by user-level *Coprocessor* on the HBase regions; the results of the sub-queries are accumulated at the client-side.

k-Nearest Neighbor (kNN) queries identify a number (k) of data points near to an input location. The process for computing kNN queries on the *HGrid* data model, using a *Scan*-based implementation, is as follows.
(1) First, we apply the density-based range estimation method introduced by Liu et.al [14] to estimate the search range.
(2) Given the search-range estimate, the queried indices of rows and columns are computed as described in steps 2 and 3 of *Range Query* above.
(3) Next, a *Scan* query is issued to retrieve the relevant data points.
(4) If fewer than k data points are returned, the search-range estimation is expanded and the above steps are repeated.
(5) Finally, a sorting step orders the return set in increasing distance from the input location.

IV. Experimental Results

Our experiments were performed on a four-node cluster, running on four virtual machines on an Openstack Cloud. The virtual machines run 64bit Ubuntu 11.10 and have 2 cores, 4GB of RAM, and a 200 GB disk. We used Hadoop version 1.0.2, and HBase version 0.94. Hadoop and HBase were each given 2GB of Heap size in every running node.

HDFS was configured with a replication factor of 2. HBase was managing its own Zookeeper instance running on the same machine as the HMaster.

In the experiment, gzip compression was configured on the table to reduce the data-transmission time. Next, the ROWCOL filter [2] was applied on each table for narrowing the queried range. The scan cache size was set to 5000 and the block cache was set to true, for the query processing. Finally, minor and major compaction were manually done to avoid the ultra size of store files after the data uploading.

For all test cases, we ran the experiment 5 times and we report the mean of the last three. We implemented the Range Query processing with the *Coprocessor* framework and kNN Query with *Scan*, considering the relatively small queried range in kNN Query.

A. The Data Set

For our experiments, we used two synthetic data sets because (a) we needed a "big" data set, with a sufficiently large number of data points, and (b) we needed to control the data distribution and its impact on the performance of the three different data organizations. The synthetic data set was generated based on the Bixi data set [15], which includes minute-by-minute readings from 404 bike stations around the city of Montreal. Each reading consists of the following attributes: timestamp, station id, latitude, longitude, station name, terminal name, number of docks, and number of bikes. With the same format of station object in Bixi data set, the synthetic data set augments the number of stations from 404 to one hundred million, locating them in random coordinates, following a uniform and Zipf distribution, using *commons-math3-3.0.jar*. The factor of Zipf distribution is 1.0, which represents moderately skewed data. The simulated space area is 100km*100km. This data set basically represents one hundred million objects at a certain timestamp. The size of data set is about 70GB.

B. Index Configuration

The granularity of the cells in terms of which the space is tessellated is a very important variable that substantially affects the query-processing performance. This is why, before we compare the three data models against each other, we have explored the "best" cell configuration for each model. In this first round of experiment, we varied the size of the cell to observe how different cell sizes affect the performance of each data model. We set the size of the cell at 0.1km, 1km, and 10km. Consequently, in the regular-grid index, the 10,000km^2 space is divided into $1,000 * 1,000$, $100 * 100$, and $10 * 10$; in the quad-tree index, as the space is split at the mid-point of a dimension each time, the size of the cell is less or equal to the configured value above.

[2]This is a type of Bloom Filter. When applied, the hash of the *row key,column family, column family qualifier* is added to the bloom on each key insert. It can help prune the data from both row and columns sides.

1) The Quad-Tree and Regular-Grid Data Models: For our uniform-distribution data set of 100 million data points, configuring the individual cell to cover a 1km*1km space results in 100*100 square cells with an average number of 10,000 data points in each cell. As a row corresponds to a cell in the quad-tree data model, there are approximately 10,000 rows in total, and in each row, there are about 10,000 columns (one for each data point) with a depth of one. In the regular-grid data model, *row key*s correspond to the indices of the grid rows and the *column name*s correspond to the column indices; as a result there are at most 100 rows and 100 columns. In each cell, there may be a stack of about 10,000 data points. Given the same amount of data and the same grid granularity, the quad-tree data model results in a wide, shallow, and long table, while the regular-grid data model results in a narrow, deep, and short table. With a fixed cell size, as the amount of data increases, the quad-tree data model expands in the column dimension (i.e., the table becomes wider), and the regular-grid data model results in a deeper table as more data points get stacked on top of each other in the third dimension.

Table I reports the response time for range queries issued to the data set organized under the quad-tree and regular-grid data models. The label "≈0.1" in "QT" column refers to an experiment where the configured size of each cell is 0.1km, and the actual cell size is around 0.097km, with the quad-tree depth being 11. The grid is divided into $2^{10} * 2^{10}$ square cells. A range query is issued with the same reference location and a number of different radiuses, ranging from 0.01km to 12km. The *Target Objects* column reports the number of data points returned by that query. The column *FP* (i.e., *False Positive*) represents the percentage of data points returned to the client without actually belonging to the query return set. The higher this percentage, the more undesirable the situation since it implies that many irrelevant rows have been scanned, and have been transferred through the network to the client and have to be inspected and rejected by the client in the post-processing phase.

From Table I we can see that for the quad-tree data model, as the size of the cell decreases from 10km to 0.1km, the performance improves substantially for the small queries, while for the large queries, the result cannot be returned before the timeout. This is because, for smaller queries, only a small number of false positives rows is included in the data returned to the client. On the other hand, for the larger queries, many irrelevant rows have to be scanned, since the Z-value for cell ordering does not preserve a good locality (i.e., the neighboring relation) among subspaces. Even though the HBase *Coprocessor* framework parallelizes the query processing, at the core of the query processing lies a *Scan* operation; therefore, better pruning of unrelated data and fewer false positives remain the key of performance improvement. The same principle also applies to the regular-grid data model. The "RG" with the size of cell of 0.1km

Table I
EXECUTION TIME OF RANGE QUERY WITH VARIOUS SIZES OF CELL OF THREE DATA MODELS(S)

Radius	Target	QT (km)			RG (km)			HG (km)				
(km)	Objects	≈0.1	≈1	≈10	0.1	1	10	50:0.1	≈10:0.1	≈1:0.1	≈:10:0.01	≈10:0.001
0.01	1	0.112	0.252	7.710	0.131	0.208	6.196	0.208	0.185	0.211	0.188	0.177
0.05	72	0.145	0.249	7.743	0.135	0.221	6.242	0.222	0.231	0.178	0.202	0.241
0.1	315	0.141	0.240	7.731	0.147	0.213	6.257	0.246	0.238	0.175	0.225	0.292
0.5	7,868	0.539	0.692	7.644	0.285	0.478	6.277	0.504	0.509	0.454	0.556	0.906
1	31,411	0.846	0.767	8.252	0.572	0.870	6.232	0.914	0.926	0.803	1.052	1.166
4	502,587	8.787	7.655	9.589	4.544	5.763	7.711	6.224	6.410	7.426	7.243	7.619
8	2,012,583	**NA**	**NA**	**NA**	10.693	16.782	34.468	83.920	42.372	40.542	51.637	51.918
12	4,524,996	**NA**	**NA**	**NA**	27.545	34.576	39.570	*NA*	85.014	93.343	105.635	110.967

Table II
FALSE POSITIVE IN RANGE QUERY WITH VARIOUS SIZES OF CELL OF THREE DATA MODELS (%)

Radius	Target	QT (km)			RG (km)			HG (km)				
(km)	Objects	≈0.1	≈1	≈10	0.1	1	10	50:0.1	≈10:0.1	≈1:0.1	≈:10:0.01	≈10:0.001
0.01	1	99.91	99.99	99.99	99.00	99.99	99.99	99.00	99.00	99.53	80.00	50.00
0.05	72	95.29	99.70	99.99	82.48	99.28	99.99	82.48	82.48	82.65	35.71	28.00
0.1	315	79.41	98.71	99.98	65.42	96.86	99.97	65.42	65.42	65.03	30.62	23.17
0.5	7,868	86.07	91.93	99.50	36.21	80.19	99.21	34.71	34.71	30.12	22.89	21.67
1	31,411	63.68	67.77	97.99	28.92	65.03	96.86	28.92	28.92	26.25	22.48	21.79
4	502,587	59.72	60.45	67.87	23.40	37.96	49.74	23.39	23.40	23.65	21.66	21.48
8	2,012,583	**NA**	**NA**	**NA**	22.43	30.43	77.65	22.43	22.43	22.19	21.56	21.47
12	4,524,996	**NA**	**NA**	**NA**	22.11	27.64	49.74	*NA*	22.13	22.37	21.54	21.48

Table III
EXECUTION TIME OF RANGE QUERY WITH THREE DATA MODELS (S)

	(a) Uniform Data								
Radius (km)	**0.01**	**0.05**	**0.1**	**0.5**	**1**	**4**	**8**	**12**	**16**
QT:≈1	0.251	0.250	0.240	0.692	0.767	7.656	**NA**	**NA**	**NA**
RG:0.1	0.131	0.135	0.147	0.285	0.572	4.544	10.693	27.545	45.323
HG:≈10:0.1	0.185	0.231	0.238	0.509	0.926	6.410	42.372	85.014	141.308
	(b) Skewed Data								
Radius (km)	**0.01**	**0.05**	**0.1**	**0.5**	**1**	**4**	**8**	**12**	**16**
QT:≈1	0.398	0.359	0.375	1.172	1.274	30.240	**NA**	**NA**	**NA**
RG:0.1	0.120	0.132	0.142	0.424	1.140	12.349	**NA**	**NA**	**NA**
HG:≈10:0.1	0.260	0.314	0.317	0.868	2.015	16.843	**NA**	**NA**	**NA**

configuration outperforms the other two configurations. The reason is that the finer granularity of grid can enhance of the ability of pruning.

Finer-cell granularity results in improved performance for smaller queries in the quad-tree data model and for all queries in the regular-grid data model. However, there is a limit to how small the size of the cell can become. Smaller cell size implies that a greater number of rows must be scanned to respond to the query. If the number of rows exceeds the scan cache size, a higher number of *Scan* operations between server and client will be required, which will cause the performance to deteriorate. The size of the *scan cache* is constrained by the memory availability on both the client and server side. If a high-memory configuration is available, then increasing the cache size may result in some of the failed cases to work, but the performance trend remains fundamentally the same, because the number of irrelevant rows scanned will continue to increase. For the quad-tree data model, as the index is built and stored in the memory before the query-processing phase, smaller cell size and deeper quad-tree imply increased memory allocation.

From Table I, we can observe that, for the regular-grid data model, best performance can be obtained with the cell size of 0.1km; while for quad-tree data model, the acceptable cell size is approximately 1km, with the quad-tree depth of eight.

2) The HGrid *Data Model:* In the *HGrid* data model, there are two variables that affect the *HGrid* index: the size of the tile (**T**) in the first tier and the size of the cell (**t**) within a tile. For our uniform-distribution data set of 100 million data points, if the individual cell is set as $1km^2$, the number of tiles can range from 1 (where there is only one tile and 10,000 number of cells), to 10,000 (where there is only one cell in each tile). Correspondingly, the number of rows are varying from 100 to 10,000, and the number of columns are from 1,000,000 to 10,000. Comparing these dimensions to the 10,000 rows and 10,000 columns in the quad-tree data model, and the 100 rows and 100 columns with stacks about 10,000 deep in the regular-grid data model, the *HGrid* table is neither as long as that of the quad-tree data model, nor as deep as that of the regular-grid data model.

Tables I and II report the query response times and false

positives for various tile sizes, given a fixed cell size in the *HGrid* data model. Smaller-tile organizations exhibit better performance because they support better pruning of irrelevant data. However, smaller tiles also imply a bigger number of sub-queries for every query. The "HG:≈10:0.1" organization, referring to the configuration with a T≈10km quad-tree tile and a t=0.1km regular-grid cell, involves fewer sub-queries and more false positives and outperforms the HG:≈1:0.1 organization with more sub-queries and fewer false positives. This is an evidence of the trade-off between the number of false positives and the number of sub-queries.

The number of rows involved in the query is also an important factor that influences the performance, as evidenced by the fact that the performance of the HG:≈10:0.01 organization is worse than that of the HG:≈10:0.1 organization, as shown in Table I. Thus, we conclude that the *HGrid* data model with tile size of T≈10km and cell size of t=0.1km approximates the best trade-off between the number of false positives and the number of sub-queries.

Table IV
EXECUTION TIME OF KNN QUERY WITH THREE DATA MODELS (S)

	(a) Uniform Data				
k	**1**	**10**	**100**	**1,000**	**10,000**
QT:≈1	1.766	7.689	7.432	7.759	7.231
RG:0.1	0.307	0.270	0.302	0.596	1.295
HG:≈10:0.1	0.320	0.357	0.373	0.807	2.003
	(b) Skewed Data				
k	**1**	**10**	**100**	**1,000**	**10,000**
QT:≈1	1.737	1.885	1.914	1.900	4.583
RG:0.1	0.147	0.139	0.151	0.480	1.592
HG:≈10:0.1	0.325	0.314	0.358	0.879	3.307

C. Comparison of the Three Data Models

In this section, we compare the performance of the three data models, with range and kNN queries. We used the appropriate configuration obtained in Section IV-B for each data model: QT:≈1, RG:0.1, and HG:≈10:0.1. In our experiments, we simply applied the same configuration into both uniform data and skewed data.

For uniform data, we randomly select a data point as the query input and a systematic variation of the radius. For skewed data, we selected three data points, each one with 20%, 50%, and 70% probability correspondingly in the Zipf distribution as the query input, and systematically varied the query radius from 0.01km to 4km.

1) Range Query: We evaluated the range-query performance under three data models with both uniform and Zipf distribution data. Table III shows the query response time of the three data models for various ranges when the system contains 100 million objects. As the radius increases, the size of irrelevant data vs. the return-set size ratio increases, and the running time also increases because more data points are retrieved. The regular-grid data model outperforms the others, because it supports better data locality and the percentage of irrelevant rows scanned is low. The *HGrid* data model performs much better than the quad-tree data model and slightly worse than the regular-grid data model. The same performance trends persist with both uniform and skewed data. In addition, in Table III, we can also see that for skewed data, the queries with the radius of 8km, 12km, and 16km, cannot get result under these three data models. The reason is that the data points in the result are so large that the execution time exceeds the client socket timeout.

2) k-Nearest Neighbor Query: We also evaluated the performance for k Nearest Neighbor (kNN) queries using the same data set, under the three data models. Table IV shows the response time (in seconds) for kNN queries, where k takes the values 1, 10, 100, 1,000, and 10,000. As the density-based range estimation method is employed [14], there is only one *Scan* operation in the query processing for uniform data, while for skewed data, more than one *Scan* iterations are invoked to retrieve the data. That is why the performance with skewed data under all data models is worse than that with the uniform data set. For both uniform and skewed data, the regular-grid data model performs best, followed by the *HGrid* data model with the quad-tree data model being the worst. The poor quad-tree locality contributes to the poor performance of the quad-tree data model, and also impacts the performance of *HGrid*, albeit less strongly. For skewed data, with too many false positives, the query with the data points having more than 70% probability cannot get the result below the timeout threshold under all data models when k equals to 10,000. To improve performance, a finer granularity is required to filter irrelevant data scanning.

In summary, the query performance of the *HGrid* data model is better than the quad-tree data model and worse than the regular-grid data model. The *HGrid* data model benefits from the good locality of the second-tier regular-grid index, but suffers from the poor locality of the Z-ordering linearization at the first tier. Better performance can potentially be obtained with alternative linearization techniques. In addition, the experiment result proved once again that a new configuration of these three data models for skewed data should be set.

D. Best Practices

Based on our experimental results, we have two types of guidelines for the organization of geospatial data in HBase. The first set guides the design of the data schema.

- The *row key* and *column name* should be short, since they are stored with every cell in the row.
- The *row key* should be designed to support pruning of unrelated data easily.
- The amount of data in one row should be kept relatively small. The cost (in time) of retrieving a row has n data increases more than twice with n (when n is large) [12].
- It is better to have one *column family*, only introducing more column families in the case where data access is

usually column scoped [10].

- The number of columns should be limited. A number in the hundreds is likely to lead to good performance.
- When the third dimension is used for storing other information rather than time-to-live values, it is preferable to keep it shallow, and be limited to containing up to no more than hundreds of data points, as deep stacks lead to poor insertion performance.
- The Bloom Filter [10] should be configured as it can accelerate the performance by pruning the data from both row and column sides.
- Compression can improve the performance by reducing the amount of data transmission.

The second set of guidelines refers to the implementation of the query-processing mechanism.

- It is more efficient to *Get* one row with n data points than n rows with one data point each [12].
- *Scan* operations are preferable to *Get* operations for retrieving discontinuous keys, even though the *Scan* result is bound to also include data points that are not part of the response data set.
- It is advisable to narrow the range of queried columns with the *Filter* mechanism.
- The number of rows to be scanned for a query should not exceed the scan cache size, which depends on client and server memory. Otherwise, it is better to split the query into several sub-queries.
- When there are too many unrelated rows within the defined scan range, splitting one query into multiple sub-queries with multiple *Scan* operations is more efficient than one query with *Filter* mechanism to retrieve rows one by one.
- The *Scan* operation is preferable for small queries, while *Coprocessor* for large queries.

V. Conclusions and Future Works

In this paper we proposed the *HGrid* data model for HBase, based on a hybrid index structure, combining a quad-tree and a regular grid as primary and secondary indices correspondingly. We comparatively evaluated the performance of the *HGrid* with uniform and skewed data, against the other two data models. Our results demonstrate that the *HGrid* organization scales well and supports efficient performance for range and k-nearest neighbor queries. Benefiting from the hierarchical index, the *HGrid* data model can be flexibly configured and extended. In the first tier, the quad-tree index can be replaced by the hash code of each sub-space or the point-based quad-tree index method is employed. In addition, the granularity in the second stage can be varied from sub-space to sub-space based on the various densities. Therefore, *HGrid* is more scalable and suitable for both homogeneously covered and discontinuous spaces.

In the future, we plan to experiment with alternative space-filling curves for the linearization of the quad-tree first-tier index, and to evaluate the model with real data.

Acknowledgment

This work has been funded by the SAVI Strategic Network, NSERC, AITF and IBM.

References

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

[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[3] R. Finkel and J. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[4] A. Guttman, *R-trees: a dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[5] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[6] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[7] G. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[8] C. Freksa and D. Mark, *Spatial Information Theory. Cognitive and Computational Foundations of Geographic Information Science: International Conference COSIT'99 Stade, Germany, August 25-29, 1999 Proceedings*. Springer, 1999, vol. 1661.

[9] J. Lawder and P. King, "Using space-filling curves for multi-dimensional indexing," *Advances in Databases*, pp. 20–35, 2000.

[10] A. S. Foundation, "Apache HBase Reference Guide," April 2012. [Online]. Available: http://hbase.apache.org/book/book.html

[11] K. Muthukkaruppan, "HBase @ FacebookThe Technology Behind Messages," April 2012. [Online]. Available: http://qconlondon.com/dl/qcon-london-2011/slides/KannanMuthukkaruppan_HBaseFacebook.pdf

[12] Y.-T. Hsu, Y.-C. Pan, L.-Y. Wei, W.-C. Peng, and W.-C. Lee, "Key formulation schemes for spatial index in cloud data managements," in *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, 2012, pp. 21–26.

[13] Wikipedia, "Bloom Filter," May 2013. [Online]. Available: http://en.wikipedia.org/wiki/Bloom_filter

[14] D. Liu, E. Lim, and W. Ng, "Efficient k nearest neighbor queries on remote spatial databases using range estimation," in *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*. IEEE, 2002, pp. 121–130.

[15] D. Han and E. Stroulia, "A three-dimensional data model in hbase for large time-series dataset analysis," in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*. IEEE, 2012, pp. 47–56.