INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 11(5):613-630, October 2016.

Efficient Historical Query in HBase for Spatio-Temporal Decision Support

X.Y. Chen, C. Zhang, B. Ge, W.D. Xiao

Xiao-Ying Chen, Chong Zhang*, Bin Ge, Wei-Dong Xiao
Science and Technology on Information Systems Engineering Laboratory
National University of Defense Technology
Changsha 410073, P.R.China
chenxiaoying1991@yahoo.com, leocheung8286@yahoo.com
gebin1978@gmail.com, wilsonshaw@vip.sina.com
*Corresponding author: leocheung8286@yahoo.com

Abstract: Comparing to last decade, technologies to gather spatio-temporal data are more and more developed and easy to use or deploy, thus tens of billions, even trillions of sensed data are accumulated, which poses a challenge to spatio-temporal Decision Support System (stDSS). Traditional database hardly supports such huge volume, and tends to bring performance bottleneck to the analysis platform. Hence in this paper, we argue to use NoSQL database, HBase, to replace traditional back-end storage system. Under such context, the well-studied spatio-temporal querying techniques in traditional database should be shifted to HBase system parallel. However, this problem is not solved well in HBase, as many previous works tackle the problem only by designing schema, i.e., designing row key and column key formation for HBase, which we don't believe is an effective solution. In this paper, we address this problem from nature level of HBase, and propose an index structure as a built-in component for HBase. STEHIX (Spatio-TEmporal Hbase IndeX) is adapted to two-level architecture of HBase and suitable for HBase to process spatio-temporal queries. It is composed of index in the meta table (the first level) and *region index* (the second level) for indexing inner structure of HBase regions. Base on this structure, three queries, range query, kNN query and GNN query are solved by proposing algorithms, respectively. For achieving load balancing and scalable kNN query, two optimizations are also presented. We implement STEHIX and conduct experiments on real dataset, and the results show our design outperforms a previous work in many aspects.
Keywords: spatio-temporal query, HBase, range query, kNN query, GNN query, load balancing.

1 Introduction

Nowadays, either organizations or common users need sophisticated spatio-temporal Decision Support System (stDSS) [1] for countless geospatial applications, such as urban planning, emergency response, military intelligence, simulator training, and serious gaming. Meanwhile, with the development of positioning technology (such as GPS) and other related applications, huge of spatio-temporal data are collected, of which volume increases to PB or even EB. Consequently, this necessarily poses a challenge to stDSS applications. Traditionally, these data are stored in relational database, however, since the database can't resist such a huge volume, such architecture would bring performance bottleneck to the whole analysis task. Hence, the new structural storage system should back up stDSS. In this paper, we argue that HBase [2] is capable to accomplish such task, since HBase is a key-value, NoSQL storage system, which can support large-scale data operations efficiently.

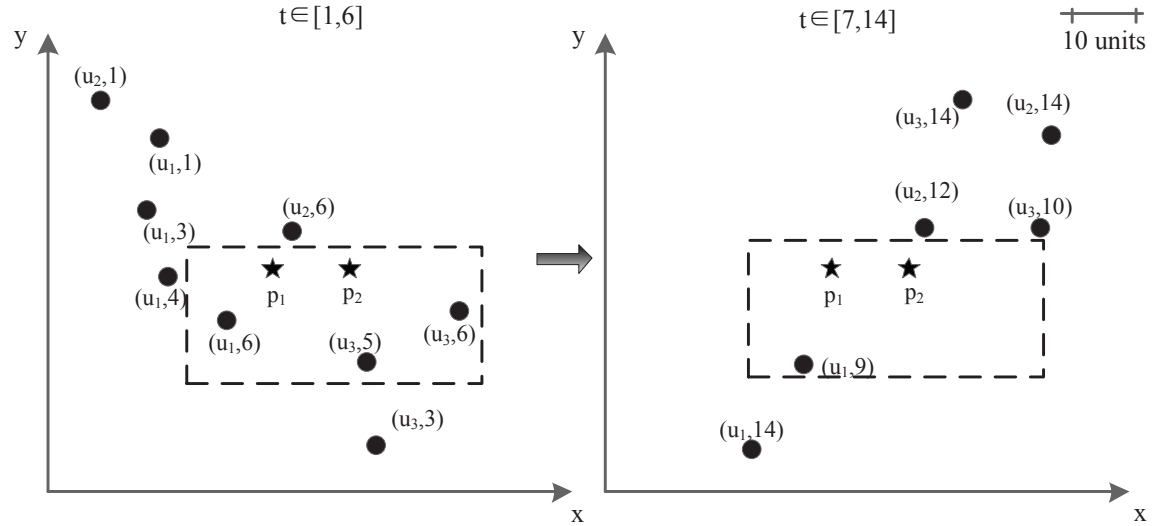
On the other hand, from system point of view, an ideal geospatial application designed to formulate and evaluate decision-making questions for stDSS should contain efficient presentation

of a basic set of spatio-temporal queries, such as: find doctors who can carry out rescue in a certain area, recently, find 5 flower shops nearest to Tony, a group of friends spreading over different places want to find nearest restaurant to them, aggregately, i.e., the sum of distances to them is minimum. These operations are supported well in relational database, however, they are not supported by HBase in a straightforward way. The main reason is that HBase do not natively support multi-attribute index, which limits the rich query applications.

Hence in this paper, we explore processing for basic spatio-temporal queries in HBase for stDSS. From a variety of applications, we mainly address three common and useful spatio-temporal queries as follows:

- **range query**: querying data in specific spatial and temporal range. For instance, in real-time monitoring and early warning of population, query the number of people in different time intervals within a specific area.
- **kNN query (k-Nearest Neighbor)**: querying data to obtain k nearest objects to a specific location during a certain period. For instance, in the past week, find 5 nearest Uber taxis to a given shopping mall.
- **GNN query (Group Nearest Neighbor)**: querying data to obtain k nearest objects aggregately (measured by sum of distances) to a group of specific locations during a certain period. For instance, during last month, find the nearest ship to the given three docks.

As an example, Figure 1 shows the spatial distribution of users during two time interval [1, 6] and [7, 14]. For range query, find the users who are in the spatial range marked by the dashed line rectangle within time period [1, 6], apparently, $\{u_1, u_3\}$ is the result. For 1NN query, if we want to find the users who are nearest to p_1 during time period [1, 6] and [7, 14], respectively, the result is u_2 for [1, 6] and u_1 for [7, 14]. For GNN query, if we want to find the user who are nearest to p_1 and p_2 by summing the distances during time period [1, 6], the result is u_2.



	$(u_2,6)$	$(u_1,6)$	$(u_1,4)$	$(u_3,5)$	$(u_1,3)$	$(u_1,1)$	$(u_3,6)$	$(u_3,3)$	$(u_2,1)$
$\|p_1,u_i\|$	9	16	23	28	30	38	42	44	53
$\|p_2,u_i\|$	14	27	37	20	43	48	27	39	63
sum	23	43	60	48	73	86	69	83	116

	$(u_1,9)$	$(u_2,12)$	$(u_1,14)$	$(u_3,10)$	$(u_3,14)$	$(u_2,14)$
$\|p_1,u_i\|$	21	22	42. 5	45	47	52
$\|p_2,u_i\|$	30	10	52	32	38	42. 5
sum	51	32	94. 5	77	85	94. 5

Figure 1: An example for range, kNN and GNN query

1.1 Motivation

Our motivation is to adapt HBase to efficiently process spatio-temporal queries as basic operations for spatio-temporal decision support system. Although some previous works propose distributed index on HBase, but these works only consider spatial dimension, more critically, most of these works only concern how to design schema for spatial data, which do not tackle the problem from the nature level of HBase, except one, MD-HBase [5] is designed to add index structure into the meta table, however, it doesn't provide index to efficiently retrieve the inner data of HBase regions. Our solution, STEHIX (Spatio-TEmporal Hbase IndeX), is built on two-level lookup mechanism, which is based on the retrieval mechanism of HBase. First, we use Hilbert curve to linearize geo-locations and store the converted one-dimensional data in the meta table, and for each region, we build a *region index* indexing the *StoreFile*s in HBase regions. We focus on range query, kNN query and GNN query for such environment in this paper.

1.2 Contributions and paper organization

We address how to efficiently answer range query, k nearest neighbor (kNN) query and GNN query on spatio-temporal data in HBase. Our solution is called STEHIX (Spatio-TEmporal Hbase IndeX), which fully takes inner structure of HBase into consideration. The previous works focus on building index based on the traditional index, such as R-tree, B-tree, while our method constructs index based on HBase itself, thus, our index structure is more suitable for HBase retrieval. In other way, STEHIX considers not only spatial dimension, but also temporal one, which is more in line with user demand.

We use Hilbert curve to partition space as the initial resolution, the encoded value of which is used in the meta table to index HBase regions, then we use quad-tree to partition Hilbert cells as the finer resolution, based on this, we design *region index* structure for each region, which contains the finer encoded values for indexing spatial dimension and time segments for indexing temporal dimension. And later, we show such two-level index structure, meta table + *region index*, is more suitable for HBase to process query in the experiment. Based on our index structure, algorithms for range query, kNN query and GNN query are devised, and load balancing policy and optimization to kNN query are also presented to raise STEHIX performance. We compare STEHIX with MD-HBase on real dataset, and the results show our design philosophies make STEHIX to be more excellent than the counterpart. In summary, we make the following contributions:

- We propose STEHIX structure which fully follow inner mechanism of HBase and is a new attempt on building index for spatio-temporal data in HBase platform.
- We propose efficient algorithms for processing range query, kNN query and GNN query in HBase.
- We carry out comprehensive experiments to verify the efficiency and scalability of STEHIX.

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 formally defines the problem and prerequisites. Section 4 presents STEHIX structure. In section 5, algorithms for range query kNN query and GNN query are presented. Section 6 reports the optimizations to the index. And we experimentally evaluate STEHIX in section 7. Finally, section 8 concludes the paper with directions for future works.

2 Related Works

To overcome the drawbacks of traditional RDBMS, as an attractive alternative for large-scale data processing, Cloud storage system currently adopts a hash-like approach to retrieve data that only support simple keyword-based queries, but lacks various forms of information search. For data processing operations, several cloud data managements (CDMs), such as HBase, are developed. HBase, as NoSQL databases, is capable to handle large scale storage and high insertion rate, however, it does not offer much support for rich index functions. Many works focus on this point and propose various approaches.

Nishimura et al. [5] address multidimensional queries for PaaS by proposing MD-HBase. It uses k-d-trees and quad-trees to partition space and adopts Z-curve to convert multidimensional data to a single dimension, and supports multi-dimensional range and nearest neighbor queries, which leverages a multi-dimensional index structure layered over HBase. However, MD-HBase builds index in the meta table, which does not index inner structure of regions, so that scan operations are carried out to find results, which reduces its efficiency.

Hsu et al. [6] propose a novel Key formulation scheme based on R^+-tree, called KR^+-tree, and based on it, spatial query algorithm of kNN query and range query are designed. Moreover, the proposed key formulation schemes are implemented on HBase and Cassandra. With the experiment on real spatial data, it demonstrates that KR^+-tree outperforms MD-HBase. KR^+-tree is able to balance the number of false-positive and the number of sub-queries so that it improves the efficiency of range query and kNN query a lot. This work designs the index according to the features found in experiments on HBase and Cassandra. However, it still does not consider the inner structure of HBase.

Zhou et al. [7] propose an efficient distributed multi-dimensional index (EDMI), which contains two layers: the global layer divides the space into many subspaces adopting k-d-tree, and in the local layer, each subspace is associated to a Z-order prefix R-tree (ZPR-tree). ZPR-tree can avoid the overlap of MBRs and obtain better query performance than other Packed R-trees and R^*-tree. This paper experimentally evaluates EDMI based on HBase for point, range and kNN query, which verifies its superiority. Compared with MD-HBase, EDMI uses ZPR-tree in the bottom layer, while MD-HBase employs scan operation, so that EDMI provides a better performance.

Han et al. [8] propose HGrid data model for HBase. HGrid data model is based on a hybrid index structure, combining a quad-tree and a regular grid as primary and secondary indices, supports efficient performance for range and kNN queries. This paper also formulates a set of guidelines on how to organize data for geo-spatial applications in HBase. This model does not outperform all its competitors in terms of query response time. However, it requires less space than the corresponding quad-tree and regular-grid indices.

HBaseSpatial, a scalable spatial data storage based on HBase, proposed by Zhang et al. [9]. Compared with MongoDB and MySQL , experimental results show it can effectively enhance the query efficiency of big spatial data and provide a good solution for storage. But this model does not compare with other distributed index method.

All the previous works we have mentioned above only consider the spatial query. For moving objects, a certain type of geo-spatial applications, requires high update rate and efficient real-time query on multi-attributes such as time-period and arbitrary spatial dimension. Du et al. [10] present hybrid index structure based on HBase, using R-tree for indexing space and applying Hilbert curve for traversing approaching space. It supports efficient multi-dimensional range queries and kNN queries, especially it is adept at skewing data compared with MD-HBase and KR^+-tree. As this work focus on moving objects, it is different for our goal, and it also does not take the inner structure of HBase into account.

To address the shortcoming which have mentioned above, the STEHIX structure which fully follow inner mechanism of HBase and is a new attempt on building index for spatio-temporal data in HBase platform is proposed.

3 Problem Definition and Prerequisites

In this section, we first formally describe spatio-temporal data, and then present the structure of HBase storage. For simplicity, only two-dimensional space is considered in this paper, however, our method can be directly extended into higher dimensional space.

A record r of spatio-temporal data can be denoted as $\langle x, y, t, \mathcal{A}\rangle$, where (x, y) means the geo-location of the record, t means the valid time when the data is produced, $\mathcal{A}$ represents other attributes, such as user-id, object's shape, descriptions, and etc. We give the descriptions for structure of storage and index in HBase [11], [12], for simplicity, some unrelated components, such as *HLog* and version, are omitted. Usually, an HBase cluster is composed of at least one administrative server, called *Master*, and several other servers holding data, called *RegionServer*s.

Logically, a table in HBase is similar to a grid, where a cell can be located by the given row identifier and column identifier. Row identifiers are implemented by row keys (rk), and the column identifier is represented by column family (cf) + column qualifier (cq), where a column family consists of several column qualifiers. The value in a cell can be referred to as the format (rk, cf:cq). Table 1 shows a logical view of a table in HBase. For instance, value v_1 can be referred to as (rk_1, cf_1:cq_1).

Table 1: Logical View for HBase Table

	cf_1			cf_2	
	cq_1	cq_2	cq_3	cq_a	cq_b
rk_1	v_1	v_2	v_3	v_4	v_5
rk_2	v_6	v_7	v_8	v_9	v_{10}

Physically, a table in HBase is horizontally partitioned along rows into several regions, each of which is maintained by exactly one *RegionServer*. The client directly interacts with the respective *RegionServer* when executing read or write operations. When the data, formally as $\langle rk$, cf:cq, $value\rangle$ (we alternatively use term key-value data in rest of the paper), are written into a region, the *RegionServer* first keeps the data in a list-like memory structure called *MemStore*, where each entry is pre-configured with the same fixed size (usually 64KB) and the size of a certain number of entries is equal to that of the block of the underlying storage system, such as HDFS. When the size of *MemStore* exceeds a pre-configured number, the whole *MemStore* is written into the underlying system as a *StoreFile*, the structure of which is similar to that of *MemStore*. Further, when the number of *StoreFile*s exceeds a certain number, the *RegionServer* will execute the compaction operation to merge *StoreFile*s into a new large one. HBase provides a two-level lookup mechanism to locate the *value* corresponding to the key (rk, cf:cq). The catalog table *meta* stores the relation {[table name]:[start row key]:[region id]:[region server]}, thus given a row key, the corresponding *RegionServer* can be found, and then the *RegionServer* searches the *value* locally according to the given key (rk, cf:cq). Figure 2 shows an example of HBase two-level lookup structure.

From above descriptions, we can see that HBase only provides a simple hierarchical index structure based on the meta table, and the corresponding *RegionServer* must do scan work to refine the results, which would be inefficient to handle spatio-temporal queries.

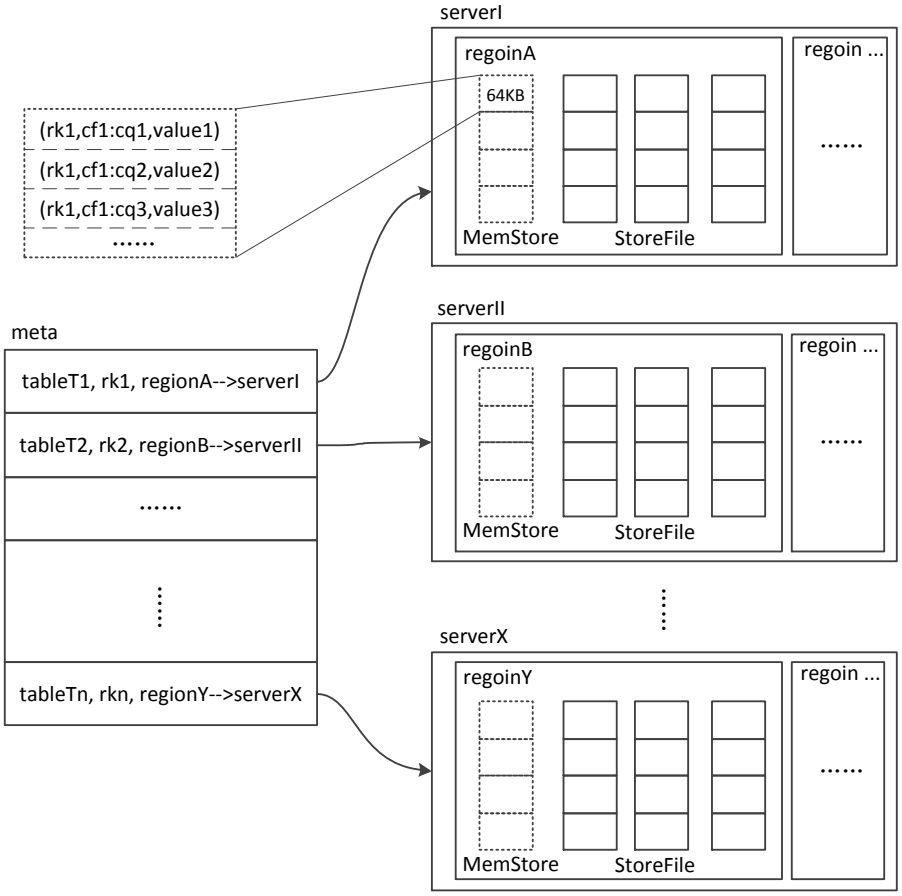


Figure 2: HBase Two-Level Lookup

4 STEHIX Structure

In this section, we present the structure of our index, STEHIX (Spatio-TEmporal Hbase IndeX). The following philosophies are considered during index design, 1) for applications, it is not necessary for users to dedicatedly to design schema for query spatio-temporal data, i.e., our index should add no restriction on schema design, but a inner structure associated with HBase, 2) the index should be in accordance with the architecture of HBase as identical as possible, 3) the index should be adaptive to data distribution.

For design rule 1), we don't care the schema design and generalize each record to be a key-value data in *StoreFile*(*MemStore*), formally $(rk, cf{:}cq, r)$, where $r{=}\langle x, y, t, \mathcal{A}\rangle$.

For design rule 2), our index is built on the two-level lookup mechanism. In particular, we use Hilbert curve to linearize geo-locations and store the converted one-dimensional data in the meta table, and for each region, we build a *region index* to index the *StoreFile*s. Figure 3 shows an overview of STEHIX architecture.

4.1 Meta Table Organization

We use Hilbert curve to partition the whole space as the initial granularity. According to the design rationale of HBase, the prefix of row key should be different so that the overhead of inserting data could be distributed over *RegionServer*s. And such design is able to satisfy this demand.

Hilbert curve is a kind of space filling curve which maps multi-dimensional space into one-dimensional space. In particular, the whole space is partitioned into equal-size cells and then

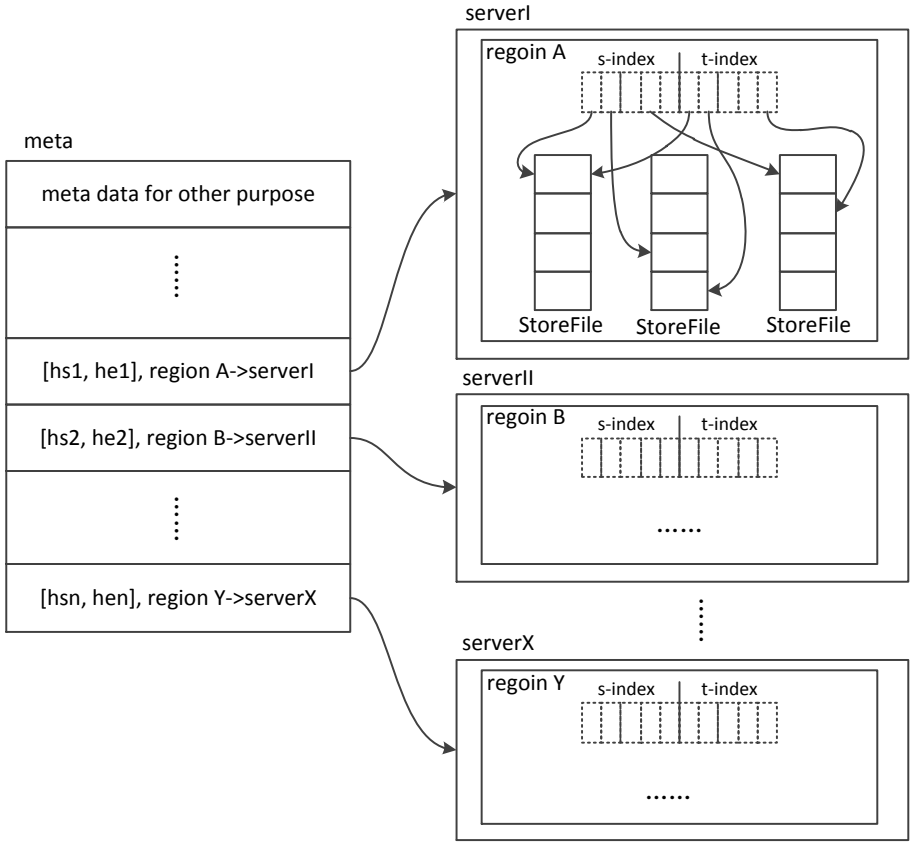


Figure 3: Overview of STEHIX

a curve is passed through each cell for only once in term of some sequence, so that every cell is assigned a sequence number. Different space filling curves are distinguished by different sequencing methods. Due to information loss in the transformation, different space filling curves are evaluated by the criteria, locality preservation, meaning that how much the change of proximities is from original space to one-dimensional space. Hilbert curve is proved to be the best locality preserved space filling curve [13]. With Hilbert curve, any object in the original space is transformed into $[0, 2^{2\lambda} - 1]$ space, where λ is called the order of Hilbert curve. Figure 4 shows four Hilbert curves in two-dimensional space with λ=1, 2, 3 and 4.

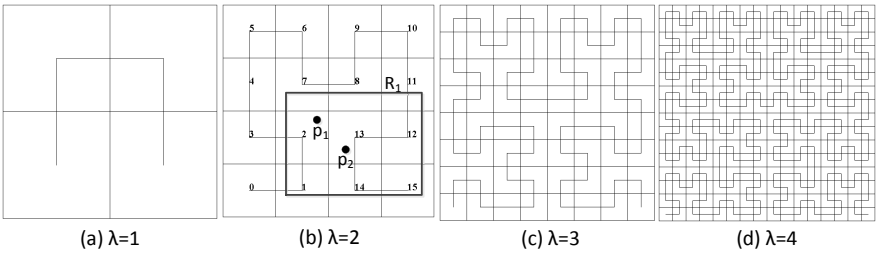


Figure 4: Hilbert Curves

We describe three functions for Hilbert curve, first one is mapping a point in the original space to a value in one-dimensional space, the second is mapping a range window to a series of intervals, and the third is retrieving proximity cells of a point. Specifically, for a Hilbert curve with order=λ,

- $coorToCell(p)$. Given a point p=(x_1, x_2, …, x_n) in n-dimensional space $\mathbb{S}$, $coorToCell(p)$ returns a cell number (between 0 and $2^{2\lambda} - 1$) referring the cell where p lies within $\mathbb{S}$.

- $rectToIntervals(R)$. Given a range window R=(x_1^l, x_2^l, ..., x_n^l, x_1^u, x_2^u, ..., x_n^u) in n-dimensional space $\mathbb{S}$, where x_i^l and x_i^u ($1 \leq i \leq n$) are the lower and upper bound of the ith-dimension, respectively, $rectToIntervals(R)$ returns a series of intervals representing the cells intersecting with R in $\mathbb{S}$.
- $getNeighborCells(p)$. Given a point p=(x_1, x_2, ..., x_n) in n-dimensional space $\mathbb{S}$, $getNeighborCells(p)$ returns a list of cell numbers referring the cells which are neighbors of the cell $coorToCell(p)$.

For instance, in Figure 4 (b), $coorToCell(p_1) = 2$, $coorToCell(p_2) = 13$, $rectToIntervals(R_1)$ = {[1,2], [7,8], [11,15]}, and $getNeighborCells(p_2)$={1, 2, 7, 8, 11, 12, 15, 14}.

Based on above descriptions, we use Hilbert cell value as row key in the meta table to index spatio-temporal data as first level, thus, each record can be placed into the corresponding region according to Hilbert value of spatial part of the record. In particular, the following mapping structure is built in the meta table (for simplicity, table name is omitted): {[start Hilbert cell, end Hilbert cell]:[region id]:[region server]}. Initially, assuming there are N regions across M *RegionServer*s, we can uniformly assign Hilbert cells to these regions, for instance, the first entry could be $\{[0, ((2^{2\lambda} - 1)/N) - 1] : regionA : serverI\}$, and the second $\{[((2^{2\lambda} - 1)/N), (2 * (2^{2\lambda} - 1)/N) - 1] : regionB : serverII\}$.

4.2 Region Index Structure

For retrieving local data efficiently, we design the *region index* which is kept in memory like *MemStore*. Considering *MemStore* is always kept in memory, *region index* is only to index *StoreFile*, however, for answering a query, *MemStore* must be scanned to guarantee the completeness of results.

Region index is a list-like in-memory structure, each entry of which points to a list of addresses referring to key-value data in the *StoreFile*. The *region index* consists of two parts, one is called *s-index* indexing spatial component of data, the other is called *t-index* indexing the temporal part, and such design is able to benefit query efficiency as we will see in next section.

For constructing *s-index*, the space is further partitioned at a finer granularity, i.e., each Hilbert cell is recursively divided by quad-tree and the resulting tiles are encoded with binary Z-Order. Such consideration is able to deal with the skewed data, i.e., when a hotspot is detected, quad-tree can be used recursively until the hotspot is eliminated. Later, we will use this idea to design an adaptive load balancing policy. After partitioning the Hilbert cell, each tile is corresponding to an entry in the *s-index*, i.e., the entry points to the key-value data whose geo-locations lie in that tile. For instance, Figure 5 shows an example of meta table and region index, where in the meta table, Hilbert cells [0, 1] indexes $regionA : serverI$ and [2, 3] for $regionB : serverII$, respectively. For $regionA$, Hilbert cells 0 and 1 are divided using quad-tree into 11 tiles, 7 of which are 2-bit tiles and 4 are 4-bit tiles, and for each entry in *s-index*, the identifier is a combination of Hilbert value and binary Z-Order value, for instance, entry 0-10, where 0 is the number of Hilbert cell 0 and 10 is the code of lower-right tile in Hilbert cell 0, points to a list containing two addresses referring to two key-value records in *StoreFile*.

For building *t-index*, we use a period T to bound the length of the list of *t-index*, and such consideration is based on the fact that there may be some cycle for the spatial change of objects. The period T is divided into several segments, each of which is corresponding to an entry in *t-index*. Each entry points to a list of addresses referring to key-value data in *StoreFile*, whose temporal component modulo T lies in the segment. Continuing the example, Figure 5 shows the structure of *t-index*. Let T=24, which means a period of 24 hours is a cycle, and let each segment = 3 hours, which means T is divided into 8 segments, and entry [3, 6) points to 8 key-value data whose temporal value modulo 24 between 3 and 6.

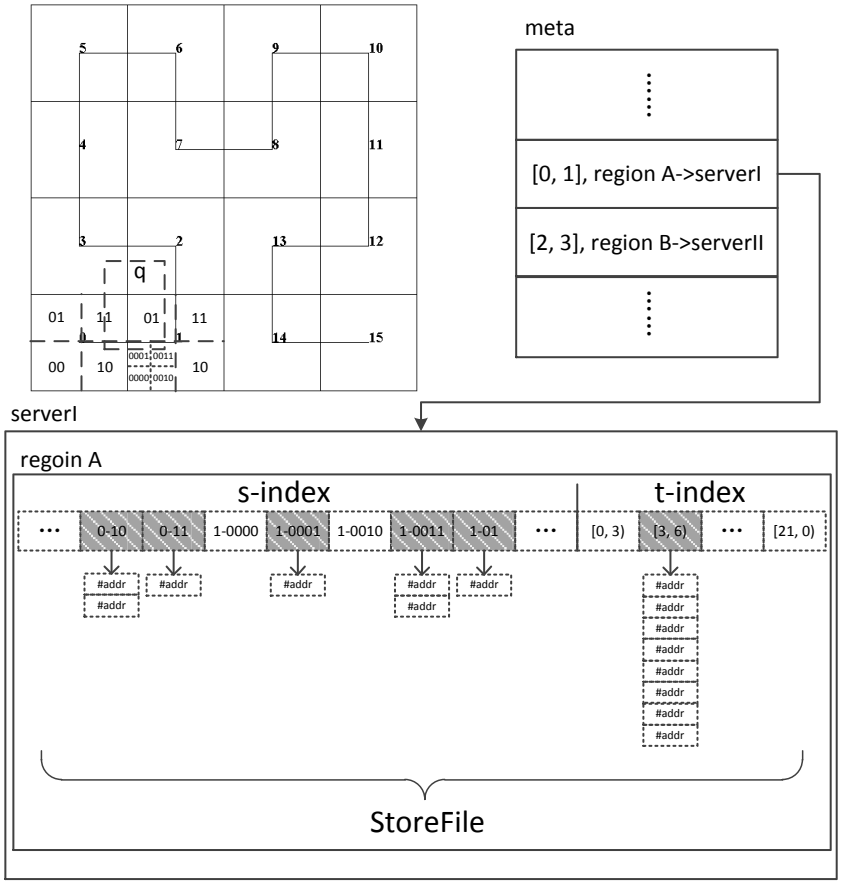


Figure 5: Region Index Structure

5 Query Processing

In this section, the processing algorithms for range query, kNN query and GNN query are presented.

5.1 Range Query

A range query q=(x_l, y_l, x_u, y_u, t_s, t_e), aims to find all the records, whose geo-locations lie in the range (x_l, y_l, x_u, y_u) during time [t_s, t_e].

The basic work flow for processing a range query q is described as follows, first, using Hilbert curve, spatial predicate (x_l, y_l, x_u, y_u) is converted into a set of one-dimensional intervals I_q, then according to mapping relation in the meta table, the involved *RegionServer*s are informed to search the corresponding regions locally, utilized by *region index*. Here we propose a query optimization, i.e., using *s-index* and *t-index* to calculate selectivity, which is helpful to choose the high-selectivity to filter more unrelated data, in particular, the spatial predicate is recursively divided by quad-tree, the results of which are intersected with the entries in *s-index*, and then the number of addresses to key-value data can be calculated, say sn, similarly, using *t-index* can also calculate a number, tn, then if sn is less than tn, *s-index* is followed to retrieve results, other wise *t-index* is used.

Algorithm 1 describes the range query processing for STEHIX. In line 1, the spatial predicate is converted into one-dimensional intervals I_q, and the temporal predicate is converted into [0, T] interval in line 2. In line 3, function $findRegions()$ finds the involved regions which intersect with I_q. From line 4 to 11, each corresponding *region index* is inspected to retrieve results, in

particular, *s-index* and *t-index* is used to calculate selectivity for the query, which is implemented by function $getCard()$, and the index with the lower cardinality is chosen to retrieve the results.

Algorithm 1 Range Query Processing

Require:
q=$(x_l, y_l, x_u, y_u, t_s, t_e)$
Ensure:
$Qlist$ //result list
1: $I_q = rectToIntervals(x_l, y_l, x_u, y_u)$
2: $key_s = t_s \ mod \ T$, $key_e = t_e \ mod \ T$
3: $Regions$=$findRegions(I_q)$
/*the following processing is executed separately in each *region*/
4: **for** each $region \in Regions$ **do**
5: sn=$region.s\text{-}index.getCard(x_l, y_l, x_u, y_u)$
6: tn=$region.t\text{-}index.getCard(key_s, key_e)$
7: **if** $sn \leq tn$ **then**
8: $Qlist$←$region.s\text{-}index.seachIndex(q)$
9: **else**
10: $Qlist$←$region.t\text{-}index.seachIndex(q)$
11: **end if**
12: **end for**
13: **return** $Qlist$

Figure 5 shows an example for range query processing in STEHIX. The spatial bound of q is depicted with dashed line and we assume that temporal predicate of q is [3, 6]. Then Hilbert cells 0 and 1 are intersected with q, thus, two entries in the meta table are examined, namely, $\{[0, 1] : regionA : serverI\}$ and $\{[2, 3] : regionB : serverII\}$. For instance, in $regionA$, the entries in *s-index* are intersected with spatial predicare of q, resulting 0-10, 0-11, 1-0001, 1-0011, 1-01 these 5 entries, which refer to totally 7 addresses to key-value data, and similarly, entry [3, 6) of *t-index* refers to 8 addresses, consequently *s-index* is followed to retrieve the results.

5.2 kNN Query

A kNN query could be formally defined as: given a set $\mathcal{R}$ of spatio-temporal data records, a kNN query q=(x_q, y_q, t_s, t_e, k), aims to find a set $\mathcal{R}(q) \subseteq \mathcal{R}$, such that $|\mathcal{R}(q)|$=k, and $d(o, (x_q, y_q)) \leq d(o\prime, (x_q, y_q))$, $\forall o \in \mathcal{R}(q)$, $o\prime \in \mathcal{R} \setminus \mathcal{R}(q)$, and $o.t$, $o\prime.t \in [t_s, t_e]$, where $d()$ is the Euclidean distance function.

We don't want to use n range queries to accomplish the kNN query, which means continuously enlarging spatial range of the query until k records are obtained [14], because we believe such a method would cause heavy querying overhead. We propose an approach utilized by incremental retrieval idea [15]. The basic work flow is, proximity objects of point (x_q, y_q) are constantly, incrementally retrieved until k results are found. In particular, first, Hilbert cell h containing point (x_q, y_q) is located, then the corresponding *region index* is utilized to retrieve all records lie in h, meanwhile, neighbor cells of h are also retrieved, and these records and Hilbert cells are all enqueued into a priority queue where priority metric is the distance from (x_q, y_q) to record or Hilbert cell. Then top element is constantly dequeued and processed, either being added to result list or being followed to retrieve neighbor cells to be enqueued, until k results are found.

Algorithm 2 presents kNN query processing. The first line initializes a priority queue PQ where each element is ordered by the distance from (x_q, y_q) to the element. The element can be Hilbert cell or record, and if it is a Hilbert cell, the distance is $MINDIST$ [16], other wise, the distance is the Euclidean distance from (x_q, y_q) to geo-location of the record. In line 2, the Hilbert cell containing (x_q, y_q) is gained, and is enqueued in line 3. From line 4, the procedure constantly retrieves top element e from PQ (line 5) and processes it, in particular, if e is a Hilbert cell (line 6), find the corresponding region rg from the meta table (line 7), and then the corresponding *region index* is searched to retrieve all the records satisfying temporal predicate (line 8), which are enqueued into PQ (line 9 to 11), after that, the neighbor cells of e are obtained and enqueued into PQ (line 12 to 15); other wise, i.e., if e is a record (line 16), which means e is a result, e is added into $Qlist$ (line 17), and the above procedure is looped until the size of

Algorithm 2 kNN Query Processing

Require:
q=(x_q, y_q, t_s, t_e, k)
Ensure:
$Qlist$ //result list
1: PQ=$null$ //initial a priority queue
2: h=$coorToCell(x_q, y_q)$
3: $PQ.enqueue(h, MINDIST((x_q, y_q), h))$
4: **while** $PQ \neq \phi$ **do**
5: e=$PQ.dequeue()$
6: **if** e is typeof cell **then**
7: rg=$findRegions(e)$
8: RS=$rg.findRecords(e, (t_s, t_e))$
9: **for** each $record \in RS$ **do**
10: $PQ.enqueue(record, dist((x_q, y_q), record))$
11: **end for**
12: $CellSet$=$getNeighborCells(e.center)$
13: **for** each $cell \in CellSet$ **do**
14: $PQ.enqueue(cell, MINDIST((x_q, y_q), cell))$
15: **end for**
16: **else if** e is typeof record **then**
17: $Qlist \leftarrow e$
18: **if** $Qlist.size()$=k **then**
19: **return** $Qlist$
20: **end if**
21: **end if**
22: **end while**

$Qlist$ reaches k (line 18 to 20).

5.3 GNN Query

A GNN query in our work could be formally defined as: given a set $\mathcal{R}$ of spatio-temporal data records and a set of location point(s) P, a GNN query q=(P, t_s, t_e, k), aims to find a set $\mathcal{R}(q) \subseteq \mathcal{R}$, such that $|\mathcal{R}(q)|$=k, and the point(s) of $\mathcal{R}(q)$ with smallest sum of distances to all points in P ($|P|$=n), i.e. $\sum_{i=1}^{N} d(o,(x_i,y_i)) \leq \sum_{i=1}^{n} d(o\prime,(x_i,y_i))$, $\forall o \in \mathcal{R}(q)$, $o\prime \in \mathcal{R} \setminus \mathcal{R}(q)$, and $o.t$, $o\prime.t$ $\in$[t_s, t_e], where $d()$ is the Euclidean distance function.

Different from kNN query, GNN query aims to finds a group of point(s) that nearest to a set of points. In kNN query processing, firstly Hilbert cell h containing point (x_q, y_q) is located, while in GNN processing, we firstly find the ideal nearest neighbor p, which could not exist in the dataset $\mathcal{R}$. This approach is that the nearest neighbor is the point(s) "near" p. Let (x,y) be the coordinates of ideal 1NN point p and (x_i,y_i) be the coordinates of point $p_i \in P$, p minimizes the sum of distance function:

$$sum_{dist}(p,P) = \sum_{i=1}^{n} \sqrt{(x-x_i)^2 + (y-y_i)^2} \tag{1}$$

Partially calculate the derivation of function $sum_{dist}(p,P)$ with respect to variables x and y, let them equal to zero, we have:

$$\begin{cases} \dfrac{\partial sum_{dist}(p,P)}{\partial x} = \displaystyle\sum_{i=1}^{n} \frac{x-x_i}{\sqrt{(x-x_i)^2+(y-y_i)^2}} = 0 \\ \dfrac{\partial sum_{dist}(p,P)}{\partial y} = \displaystyle\sum_{i=1}^{n} \frac{y-y_i}{\sqrt{(x-x_i)^2+(y-y_i)^2}} = 0 \end{cases} \tag{2}$$

However, this equations can not be solved when $n > 2$. According to the method in [18], we start with the arbitrary initial coordinates $x = \frac{\sum_{i=1}^{n} x_i}{n}$, $y = \frac{\sum_{i=1}^{n} y_i}{n}$, then modifies as follows:

$$x = x - \eta \frac{\partial sum_{dist}(p,P)}{\partial x}, y = y - \eta \frac{\partial sum_{dist}(p,P)}{\partial y} \tag{3}$$

where η is a step size. The process is repeated until the distance function $sum_{dist}(p,P)$ converges to a minimum value. We call this processing $p = getNearest(P)$. The range around p in which we should look for points of $\mathcal{R}(q)$.

The basic work flow is similar to kNN query processing which introduce above. Algorithm 3 presents GNN query processing. In particular, first, Hilbert cell h containing point p (x, y) is located, then the corresponding *region index* is utilized to retrieve all records lie in h, meanwhile, neighbor cells of h are also retrieved, and these records and Hilbert cells are all enqueued into a priority queue where priority metric is the sum of distance from P to record or Hilbert cell. Then top element is constantly dequeued and processed, either being added to result list or being followed to retrieve neighbor cells to be enqueued, until k results are found.

Algorithm 3 GNN Query Processing

Require:
q=(P, t_s, t_e, k)
Ensure:
$Qlist$ //result list
1: PQ=*null* //initial a priority queue
2: $p = getNearest(P)$
3: h=$coorToCell(p)$
4: $PQ.enqueue(h, sum_{MINDIST}((P), h))$
5: **while** $PQ \neq \phi$ **do**
6: e=$PQ.dequeue()$
7: **if** e is typeof cell **then**
8: rg=$findRegions(e)$
9: RS=$rg.findRecords(e, (t_s, t_e))$
10: **for** each $record \in RS$ **do**
11: $PQ.enqueue(record, sum_{dist}((P), record))$
12: **end for**
13: $CellSet$=$getNeighborCells(e.center)$
14: **for** each $cell \in CellSet$ **do**
15: $PQ.enqueue(cell, sum_{MINDIST}((x_q, y_q), cell))$
16: **end for**
17: **else if** e is typeof record **then**
18: $Qlist \leftarrow e$
19: **if** $Qlist.size()=k$ **then**
20: **return** $Qlist$
21: **end if**
22: **end if**
23: **end while**

6 Optimizations

In this section, we propose two methods for raising performance of STEHIX from the aspects of load balancing and query optimization.

6.1 Adaptive Load Balancing

For achieving design rule 3), adaptive load balancing is considered. Our spatial partition procedure contains two phases, first is Hilbert curve, and the second is quad-tree. And load balancing is based on the second phase and region split, in particular, when the volume of a region exceeds a limit due to the hotspot in spatial dimension, the procedure detects which Hilbert cell is the hotspot, and uses a quad-tree to divide it into four subspaces, thus the original region is split into five regions, i.e., four corresponds to the four subspaces and one corresponds to the undivided Hilbert cell(s). After that, the meta table is also updated to renew the mapping information as well as the *region index*. Figure 6 shows an example of region split. We can see when a hotspot is generated in Hilbert cell 0, the cell is divided into four subspaces by quad-tree, and the corresponding region is split into five, namely, 0-00, 0-01, 0-10, 0-11 and 1, and the meta table and new regions are updated accordingly.

6.2 Optimization for kNN Query

From kNN algorithm we can see, each time for retrieving the records of a Hilbert cell, the meta table must be searched to locate the corresponding region, which would increase overhead of the query. To deal with such a problem, we add modifications to *region index*, in particular, each *region index ri* is connected to the regions whose Hilbert cells are the neighbors of *ri*'s

region's Hilbert cells. Thus, when *getNeighborCells*() method is invoked, the current region is able to retrieve records from proximity regions, however, not all the records can be retrieved, and for this case, the meta table should be searched. Nevertheless, this optimization would reduce the overhead of querying the meta table.

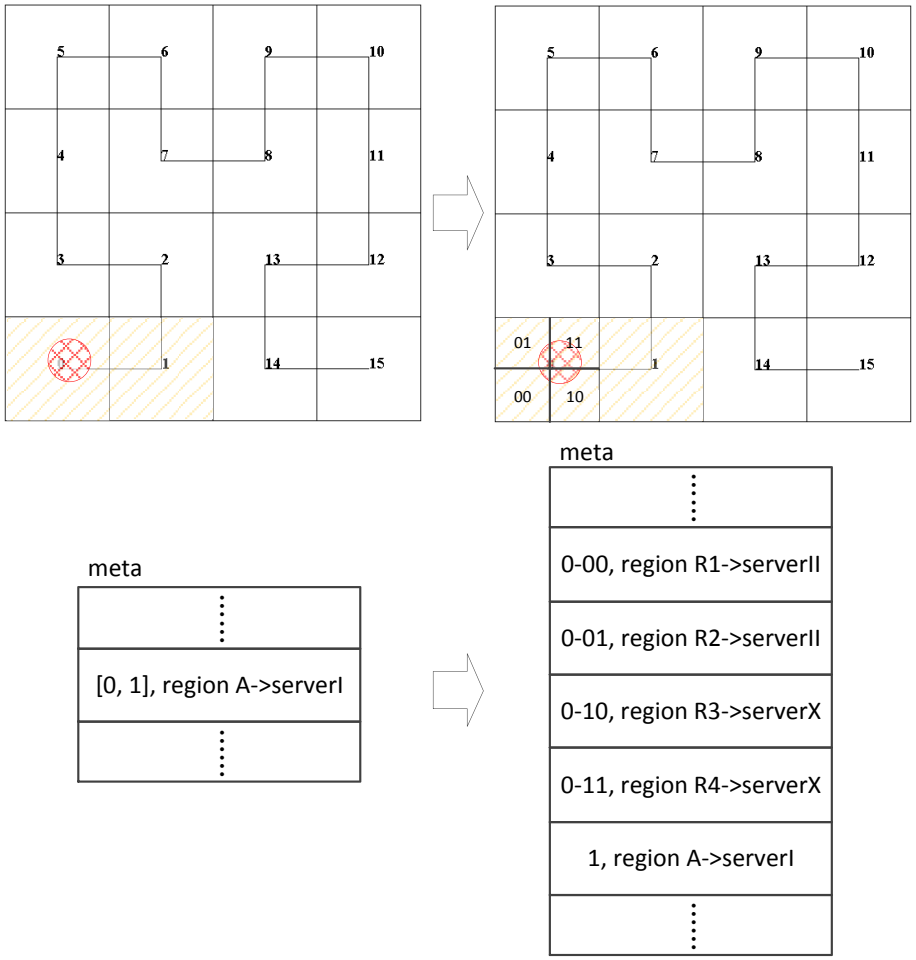


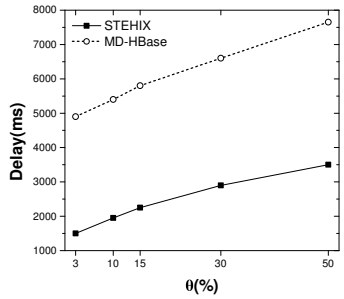
Figure 6: Load Balancing

7 Experimental Evaluation

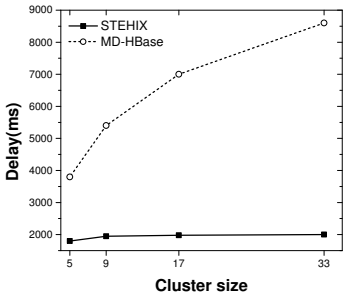
We evaluate our algorithms on real dataset, which contains trajectories of taxis in Beijing[1]. In particular, the dataset contains about 100 million records, and temporal range is from Nov. 1st to 3rd, and each record in the dataset contains vehicle ID, geo-location, recording time stamp, etc.

Our algorithms are implemented in Hadoop 2.5.1 and HBase 0.98.6, and run on a cluster with size varied from 5 to 33, in which each node is equipped with Intel(R) Core(TM) i3 CPU @ 3.40GHz, 4GB main memory (for *Master* 16GB), and 500GB storage, and operating system is CentOS release 6.5 64bit, and network bandwidth is 10Mbps. For comparison, we choose MD-HBase due to the similar function.

[1]http://activity.datatang.com/20130830/description



(a) Effect of selectivity



(b) Effect of cluster size

Figure 7: Experimental Results for Range Queries

7.1 Range Queries

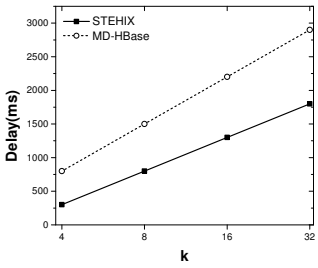
First, we evaluate the algorithm for range queries. And we introduce two parameters to test the algorithm under various conditions. One is selectivity θ defined as:

$$\theta = \frac{L_{(t_s,t_e)}}{L_t} \cdot \frac{A_{R_q}}{A_{\mathbb{S}}}$$

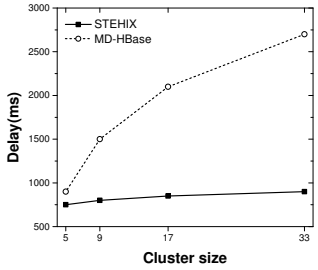
where $L_{(t_s,t_e)}$ means the length of query temporal range (t_s, t_e), L_t means the length of temporal extent of the dataset, A_{R_q} means the area of query spatial range R_q, and $A_{\mathbb{S}}$ means the area of the whole space. Selectivity specifies the size of the query range, and the larger θ is, the more spatio-temporal records are involved. In this experiment, the default values of θ and cluster size are 10% and 9, respectively. For each value of θ or size, we issue 10 queries with different temporal ranges and spatial ranges, and collect the average response time as the measurement of performance.

First, we vary θ from 3% to 50% and Figure 7(a) shows the results. We can see that response time increases with θ for both methods. This is because a larger selectivity would access more records to be retrieved and examined, which increases the processing time. However, we can see STEHIX outperforms MD-HBase, which can be explained by the design of *region index*. Although MD-HBase builds index in the meta table, it doesn't index inner structure of regions, thus, scan operations are carried out to find results, which cost heavily. Our STEHIX is adapted to the two-level architecture of HBase, and is able to use *region index* to efficiently search each region, which highly improve performances.

Next, we vary cluster size from 5 to 33, and Figure 7(b) shows the results. It is apparent that STEHIX is excellent due to its nearly horizontal response time and good scalability. When the number of cluster size is increased, more *RegionServer*s take part in the processing and use their *region index*es parallel. However, due to lack of indexing *StoreFile*s, the scalability of MD-HBase is not good.



(a) Effect of k



(b) Effect of cluster size

Figure 8: Experimental Results for kNN Queries

7.2 kNN Queries

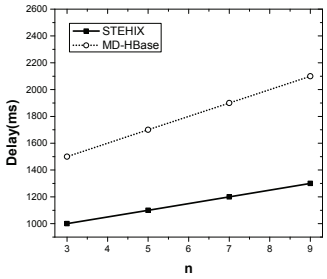
In this experiment, the default values of k and cluster size are 8 and 9, respectively. First, we vary k from 4 to 32, and Figure 8(a) shows that STEHIX outperforms MD-HBase. When k is increased, both methods need more time to process queries. STEHIX uses less time to retrieve k results, which can be explained by the same reason, i.e., the *region index* embedded in HBase region. And then cluster size is varied from 5 to 33, still, STEHIX is better than MD-HBase, Figure 8(b) shows the fact.

7.3 GNN Queries

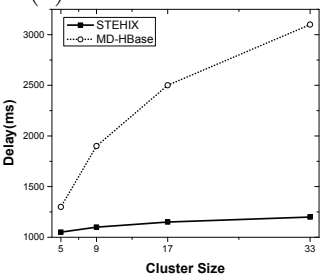
In this experiment, we vary the size of location set P and cluster to measure performance of STEHIX. Note that MD-HBase does not study GNN query, so we just simply use our virtual centroid method to apply to it. We vary n (size of P) from 3 to 9, and Figure 9(a) shows the results. We can see with increasing of n, the response time is also increased, this is because a larger size of P would cause more time to calculate the virtual centroid, however, we can the delay time does not increase very steeply, due to the fact that computing the virtual centroid only cost CPU time. Similarly, our STEHIX still outperforms MD-HBase in both varying n and cluster size.

7.4 Effect of Optimizations

We examine the effect of optimizations to STEHIX in this experiment, and Figure 10 show the results. First, we use maximum imbalance load ratio [17] as metric, and test our adaptive load balancing policy, the results of comparison with non-load balancing are plotted in Figure 10 (a). We can see with cluster size increased, both ratios are raised, this is because the more nodes participate in the cluster, the more difficult is to distribute load uniformly, however, we can see our load balancing method indeed takes effect, i.e., when load balancing policy is used, the ratio is averagely around 6, while the counterpart shows the performance about 38 to 70. Next, we test the effect of kNN optimization, from Figure 10 (b), we can see the connections among *region index*es give chances to reduce querying overhead.

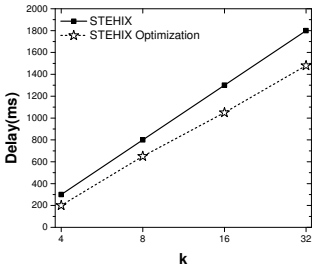


(a) Effect of size of P

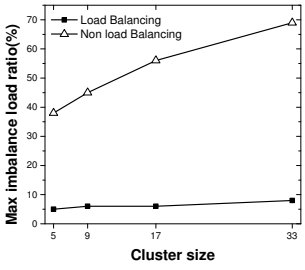


(b) Effect of cluster size

Figure 9: Experimental Results for GNN Queries



(a) Effect of k



(b) Effect of cluster size

Figure 10: Experimental Results for Optimizations

8 Conclusion and Future Works

With development of positioning technology, more and more spatio-temporal data need to be processed. To equip HBase with efficient and scalable spatio-temporal querying capability will benefit the whole spatio-temporal decision support system. In this paper, we argue that many previous works fail to tackle this problem due to lack of deep design for HBase, while we address the problem by proposing a novel index structure adapted to two-level architecture of HBase, which is suitable for HBase to process queries. Algorithms for range query, kNN query and GNN query are designed, what's more, the optimizations for load balancing and kNN query are also proposed. We carry out extensive experimental studies for verifying our index, and the results show that our approach for HBase is more efficient and scalable than the previous work.

In the future, we plan to utilize this idea to efficiently store and retrieve graph data and apply to social networks.

Acknowledgment

This work is supported by NSF of China grant 61303062 and 71331008. We would like to thank Peijun He for helping with the implementation.

Bibliography

[1] Van Orshoven et al. (2011), Upgrading geographic information systems to spatio-temporal decision support systems, *Mathematical and Computational Forestry & Natural Resource Sciences*, 3(1): 36-41.

[2] Wiki, H. HBase: bigtable-like structured storage for Hadoop HDFS. 2012-02-23)[2012-04-17]. http://wiki. apache, org/hadoop/Hbase.

[3] Ralph Kimball, Margy Ross (1996), *The data warehouse toolkit*, Wiley.

[4] Ralph Kimball, Margy Ross (2012), *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*, 2nd Edition, Wiley.

[5] Nishimura, S., Das, S., Agrawal, D., Abbadi, A. E. (2011, June). MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. *In Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, 1: 7-16.

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

[7] Zhou, X., Zhang, X., Wang, Y., Li, R., Wang, S. (2013), Efficient distributed multi-dimensional index for big data management. *In Web-Age Information Management*, Springer Berlin Heidelberg, 130-141.

[8] Han, D., & Stroulia, E. (2013), Hgrid: A data model for large geospatial data sets in hbase. *In Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, 910-917.

[9] Zhang, N., Zheng, G., Chen, H., Chen, J., Chen, X. (2014). Hbasespatial: A scalable spatial data storage based on hbase. *In Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, 644-651.

[10] Du, N., Zhan, J., Zhao, M., Xiao, D., & Xie, Y. (2015), Spatio-Temporal Data Index Model of Moving Objects on Fixed Networks Using HBase, *In Computational Intelligence & Communication Technology (CICT), 2015 IEEE International Conference on*, 247-251.

[11] HBase, A. (2012), Apache hbase reference guide. Webpage available at http://wiki. apache. org/hadoop/Hbase/HbaseArchitecture. Webpage visited, 04-04.

[12] George, L. (2011). HBase: the definitive guide, O'Reilly Media, Inc.

[13] Faloutsos, C., Roseman, S. (1989), Fractals for secondary key retrieval, *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 247-252.

[14] Wang, J., Wu, S., Gao, H., Li, J., Ooi, B. C. (2010), Indexing multi-dimensional data in a cloud system. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 591-602.

[15] Hjaltason, G. R., Samet, H. (1999), Distance browsing in spatial databases, *ACM Transactions on Database Systems (TODS)*, 24(2): 265-318.

[16] Roussopoulos, N., Kelley, S., Vincent, F. (1995). Nearest neighbor queries. *In ACM sigmod record*, 24(2):71-79.

[17] Vu, Q. H., Ooi, B. C., Rinard, M., Tan, K. L. (2009), Histogram-based global load balancing in structured peer-to-peer systems, *Knowledge and Data Engineering, IEEE Transactions on*, 21(4): 595-608.

[18] Hochreiter, S., Younger, A. S., Conwell, P. R. (2001), Learning to Learn Using Gradient Descent. *Artificial Neural Networks-ICANN 2001*, Springer Berlin Heidelberg.