ABSTRACT
With wide-spread use of location-based services, spatial data is becoming popular. As the data is usually huge in volume and continuously arriving to the storage in real-time, designing systems for efficiently storing this type of data is challenging. Two major issues that make building such system become complicated are the skewed distribution of data and the need of scaling the storage on multiple machines. In this paper, we propose a novel scalable in-memory density-based index for spatial databases. The key principle underlying our design is the exploitation of the stable spatial distribution of the datasets to deploy a simple but efficient index structure. We used information extracted from data in the past to split the entire space into independent pieces with similar density to ensure load-balancing and scalability. Experimental results show that the proposed solution scales well in distributed environment and outperforms common indexes in many cases.

CCS Concepts
• Information systems → Data structures; Distributed storage;

Keywords
Spatio-temporal databases, In-memory storages

1. INTRODUCTION
The wide-spread use of mobile devices together with the rapid development of sensor networks and satellite technologies have enabled classes of application based on collecting data from moving objects to be popular and become the most promising Internet applications. In the core of these applications, accelerating range queries by employing index structures for spatial components plays a crucial role.

There are attempts to develop indexes for spatial databases. The primary challenge addressed in these studies is the skewed distribution of spatial datasets. This often results a trade-off between read and write performance in the design of indexes. For in-memory indexes, the key to improve range query performance is to minimize the time spent on retrieving and testing objects for intersection with the query boundary. Recent approaches reach this goal by increasing CPU-cache utilization [7], implementing secondary indexes [11] or producing multiple layers [8].

With rapid development of location-based services, it is hard to keep the whole dataset in memory of a single machine. Scaling the storage on multiple machines is necessary. However, only a few studies addressed this problem [1].

This paper introduces a Scalable In-memory Density-based Index, named SIDI, a novel spatial index which tackles both skewed distribution and scalability issues. The key principle underlying SIDI design is the exploitation of the stable spatial distribution of the datasets to deploy a simple but efficient index structure. Traditionally, indexes are built from scratch and use dynamic adaptation mechanisms to adapt with incoming data. This approach causes a significant drop in performance since indexes require extra cost to change their layout to be suitable for the dataset. SIDI avoids this issue by learning characteristics of the dataset in advance to construct an appropriate layout that well fits the data. Furthermore, SIDI splits the space into pieces which do not interfere each other. Such organization not only allows it to be effectively expanded across a cluster of multiple data nodes but also is easy to maintain and recover from crashes.

The remainder of the paper is organized as follows. Section 2 reviews previous work related to ours. Section 3 introduces our assumptions about spatial distribution. Section 4, 5 and 6 describes SIDI in detail. Section 7 provides several performance experiments. Finally, we conclude in Section 8.

2. RELATED WORK
Tree-based spatial indexes split the whole space into small pieces and manage them in a hierarchical manner. Well-known structures such as R-Tree [5], QuadTree [10] and their variants are widely used in commerce products. Still, such organization scales poorly and often suffers from performance trade-off. Although QuadTree and its variants are widely used in computer graphic and simulation owing to their simplicity [8], recent researches tend to focus on R-Tree. Sidiaskas et al. [11] found that facilitating update by a secondary hash index speeds up R-Tree’s update performance significantly. CR-Tree [7] implements an inexpensive method to compress the Minimum Boundary Rectangular keys in order to enhance cache behavior. Hwang et.
al. [6] incorporated Hilbert R-Tree with CR-Tree’s compression technique and R*-Tree’s insertion and overflow treatment policy to produce other six memory-based variants of R-Tree. However, mentioned studies are dedicated to centralized system. Although there are attempts to deploy tree-based index on distributed environment, such as SDRTree [4], scalability is still the major problem as hierarchical structure cannot provide full decentralization.

Beside tree-based structures, grid-based approaches are also widely used for constructing in-memory indexes. BLOCK [8] deploys multiple grids with different levels of granularity and evaluates the query against cells in order from the highest resolution to the lowest one to reduce significantly the number of not only cell intersection tests but also object intersection tests. PASTIS [9] speeds up operations on spatio-temporal databases by employing two index layers: the spatial domain is decomposed by a grid and in each cell, a partial temporal index is maintained to keep tracks of moving objects. However, PASTIS only keeps the latest data on memory while dated ones are moved to disk. Mentioned grid-based indexes also do not scale well since they do not employ any mechanism to ensure data is distributed evenly to data nodes in case of skewed spatial distribution.

Hybrid structures use space-filling curves such as Hilbert and Z-curve to build tree structures on top of grid. The key idea of those methods is the exploitation of the locality preservation property of space-filling curves to linearize the space then use one-dimensional structures, for example, binary search trees or B-Tree for indexing data. B^<sub>dual</sub>-Tree [12] and ST^2B-tree [2] are representative examples of this approach.

Throwaway indexes create multiple indexes valid in a very short period of time in order to exploit memory speed to eliminate the overhead of keeping tracks of updates. MOVIES [3] and ToSS-it [1] are notable recent researches of this approach. One remarkable limitation of such method is that only a small part of data is indexed in memory so queries integrated by temporal constraints may be slowed down significantly when they require too old data.

### 3. SPATIAL DISTRIBUTION ANALYSIS

Perhaps most of spatial datasets are those containing tracking information from moving objects. These objects has their own trajectory pattern reflecting to their behaviors and attributes (e.g. buses always travel along predefined routes) so spatial presentation of the dataset is the combination of their trajectories. The context of data space (e.g. the transportation infrastructure or geographical factors) also affects on spatial distribution. We collected GPS data from buses and mobile devices in Ho Chi Minh City in different days and plotted it on a $40 \times 40$ grid as shown in Figure 1. The number of points falling within a cell is illustrated by its color. Clearly, the distribution pattern reflects city’s infrastructure and almost does not change by time.

![Figure 1: The density of GPS data collected from buses in Ho Chi Minh City over a $40 \times 40$ grid.](image)

Obviously, the trajectory varies from different periods of time but its presentation in long term is stable as object’s behavior and attributes hardly change. For example, the schedule may have a student go to different campuses on different days which makes his trajectory vary from day to day. However, the difference will disappear as the trajectory repeats by weeks. Thus, if the spatial distribution is analyzed in a long enough period of time, the minor change of objects’ trajectory would be trivial.

Therefore, as the two major causal factors are stable, we conclude that the spatial distribution of a group of multiple mobile objects is also stable. We consider this conclusion as the fundamental assumption for SIDI’s design.

### 4. INDEX STRUCTURE

#### 4.1 Key Concepts and Notations

Let us consider $P$ be a set consisting of every point belonging to a bounded space $\Omega$. Each point $p \in P$ is a pair of two real values $(x, y)$ representing its coordinates. $\Omega$ is a rectangular restricted by lower-left point $\omega_l$ and upper-right point $\omega_r$. A range query is defined by a rectangular $R = (r_x, r_y)$ where $r_x, r_y \in P$. The result of the range query over spatial dataset $D \subseteq P$ is a set of points $p \in D$ that fall into its rectangular as described in function $\text{search}$:

$$\text{search}(R) = \{p \in P : r_x \leq p_x \leq r_y \wedge r_x \leq p_y \leq r_y\}$$

(1)

$D$ is stored in a distributed system $S$ consisting of $N$ data nodes, named $s_0, s_1, \ldots, s_{N-1}$. In practice, each point $p \in D$ should have many copies scattered across multiple data nodes to strengthen fault tolerance ability and maintain availability. However, since memory capacity is much less than that of secondary storage, we decided to keep just one copy of each record. Thus, there must be only one data node $s \in S$ taking care of $p$.

#### 4.2 Index Structure

Since grid is relatively simple, requires low maintenance cost, and has great potential for full decentralization, we choose grid-based approach as a foundation for SIDI layout. The space $\Omega$ is divided into a grid $G$ of $n \times n$ uniform cells. Each cell is identified by its coordinates $(i, j)$ where $i$ is the number of cell in the same row on its left side and $j$ is the number of cell below it. We combine adjacent cells together to form a new unit, named zone. Zone is a rectangular identified by a 2-tuple $(bl, ur)$ where $bl$ is its lower-left cell and $ur$ is its upper-right one.

The aim of using zone is to reduce the density gap between areas: SIDI tries to add more cells to zones in low density regions to equalize their density with those in high density areas. Hence, distributing data according to zone will ensure even data distribution. Partitioning dataset to independent
zones also guarantees fast fault recovery since the system only needs to reload data from missing zones in crashed data nodes instead of the entire index.

As the spatial distribution is stable, we estimate cell density by sampling data from the past. The sample set should not be collected within too short interval since the distribution changes unpredictably in short periods of time. On the other hand, sampling in too long period of time not only consumes a lot of computation time but also makes the sample set tend to reflect characteristics of dated data so that it may not fit well the pattern of future dataset.

Since entire index is organized in memory, memory access becomes the most time consuming task. In case of range query, the majority of memory access is intersection test. Therefore, SIDI maintains pointers to data in both cell and zone in each data node to form two index layers with different levels of granularity to reduce such kind of test. For each query, the intersection with zone layer is tested first then cell layer and finally data points. Doing so improves the processing speed as it avoids testing cells in the center of query scope and points close to the side of the query.

Figure 2 depicts the overall design of SIDI. The space is decomposed into disjoint zones on top of a uniform grid. Those zones are distributed to data nodes in $S$. Inside data nodes, a vector called zone list is constructed to store data. Each element within zone list stores information related to one zone, which is metadata, pointers to data points belonging to this zone and an array named cell list. Each element within cell list holds pointers of data points falling into a cell covered by this zone. Data node also maintains a translation table storing a mapping between cells and zones to quickly identify zone given a cell.

5. PARTITIONING ALGORITHM

Since diversity changes continuously by time, SIDI does not keep the same data layout for every incoming data but periodically involves the partitioning process to update its structure. The process consists of three steps. Initially, SIDI samples a subset of recent data to construct grid $G$ over $\Omega$. After that, $G$ is split into zones by merging adjacent cells together. Finally, zones are distributed to data nodes.

5.1 Grid Construction

Recall $\Omega$ is bounded by a rectangular defined by $\omega_s$ and $\omega_e$ where $\omega_s$ is its lower-left point and $\omega_e$ is its upper-right point. Over the grid $G$ of $n \times n$ cells, the height of cells is $v(i,j)$ where $v(i,j) = (x, y)$ and the width is $w = (x_e - x_s)/n$. Given point $p \in P$, we can easily determine the cell $(i, j)$ into which it falls by using following equations:

\[
\begin{align*}
    i &= \left\lfloor \frac{p.x - \omega_s.x}{w} \right\rfloor \\
    j &= \left\lfloor \frac{p.y - \omega_s.y}{h} \right\rfloor
\end{align*}
\]

We use a sample set $D$ to construct index structure for incoming data, assuming that distribution patterns of current data has been captured by $D$. Let $v(i,j)$ be a set of points $p \in D$ plotted inside cell $(i, j)$, so the density of this cell is $|v(i,j)|$. We control the granularity of the grid by adjusting cell’s density: given dataset $D$ with predefined value $\eta$, grid $G$ is selected to layout data on $\Omega$ if it satisfies following condition:

\[
\forall (i, j) \in G : |v(i,j)| \leq \eta \times |D|
\]

This condition prevents $G$ from being too coarse. If $\eta$ decreases, $G$ must increase its granularity to break up cells into smaller ones with lower density to keep the condition still be hold. Among grids satisfy this condition, the sparsest one is set as outcome of this step.

5.2 Zone Construction

Denote $V_z$ as the set of points falling into zone $z$ then $|V_z|$ is the sum of density of cells forming it. Let $z_0 = ((0, 0), (n - 1, n - 1))$ be the largest zone covering the whole space $\Omega$, we construct zone structure $\zeta$ over $G$ by recursively splitting $z_0$ into smaller ones.

Following this, we define a cut $\chi$ over zone $z$ as a 2-tuple $(t, h)$ where $t$ indicates the direction of $\chi$: $t = 0$ if $\chi$ splits $z$
horizontally and \( t = 1 \) means it splits \( z \) vertically. \( h \) equals to the number of cells on the left (if \( t = 1 \)) or below (\( t = 0 \)) \( \chi \). Given a zone \( z \), we call \( E_z \) the set of all possible cuts that can split \( z \) into two parts.

Zone construction is processed as following: For each zone \( z \) and threshold \( \theta \), the process finishes if \( |V_z| < \theta \). Otherwise, select a cut \( \chi \) to split \( z \) into two child zones \( z_0 \) and \( z_e \) that the difference between their density (e.g. \(|V_z| - |V_{z_e}|\)) is minimum. Finally, apply the same process to \( z_0 \) and \( z_e \).

In the end, \( \zeta \) is a set of zones which are not split further. Obviously, \( \zeta \) must satisfy following condition.

\[
\forall z \in \zeta : |V_z| \leq \theta
\] (5)

This condition prevents \( z_o \) from being divided into only a few but too big zones. We do not need lower bound since the partition mechanism itself has already tried to avoid producing too many small zones.

Zones created by zone splitting algorithm is organized as a binary tree, whose root is \( z_0 \), called partition tree. In the tree, internal nodes are zones whose density is greater than \( \theta \). Their children are zones produced directly from splitting themselves. Leaf nodes are zones belonging to \( \zeta \). We denote the level \( h_z \) of zone \( z \) as the number of edges forming the path from \( z \) to the root.

Each zone is assigned a unique ID based on its position on the tree. The ID is a string whose characters are from the set \( S = \{0, 1, \tau\} \). The root’s ID is an empty string. For each zone \( z \), its zone ID \( zid_z \) is obtained by concatenating its parent’s ID \( zid_p \) with either 0 or 1 depending on its relative position with its sibling \( z’ \). If \( z \) is on the left side or below \( z’ \) then \( zid_z = zid_{z’} \cdot 0 \) and \( zid_{z’} = zid_z \cdot 1 \), otherwise, \( zid_z = zid_{z’} \cdot 1 \) and \( zid_{z’} = zid_z \cdot 0 \). (The dot ‘\( . \)’ denotes concatenate operator).

Over the set \( S = \{0, 1, \tau\} \), we define operator “\( \cdot \)” as \( 0 < \tau < 1 \). Given two arbitrary different zone IDs \( z_1 \) and \( z_2 \), we compare them as follows: if \( |z_1| \neq |z_2| \) then we add \( \tau \) to the end of the shorter zone ID until they have the same length. Thus, they can be rewritten as \( z_1 = wax \) and \( z_2 = why \) where \( w \) is the longest similar substring of them from the beginning, \( a \) and \( b \) are the first different characters and \( x \) and \( y \) are the rest of each ID. The comparison between \( z_1 \) and \( z_2 \) is now equivalent to the comparison between \( a \) and \( b \). For example, to compare two zone IDs \( z_1 = “0110” \) and \( z_2 = “0100” \), we firstly add \( \tau \) to the end of \( z_2 \) to equalize their length. Hence, \( z_2 \) is rewritten as “\( 01\tau \)”. \( z_1 \) and \( z_2 \) are then decomposed to \( w = “01” \), \( a = 1 \), \( b = 0 \), \( x = ”0” \), and \( y = “\tau” \). Since \( a = 0 \) but \( b = 1 \), we conclude \( z_1 > z_2 \).

From the above definition of comparison, we sort zones in \( \zeta \) ascendingly according to their ID to form final outcome \( Z \). We use \( Z \) to construct the zone list inside SIDI. Sorting zones by their ID is necessary as it reserves locality. Obviously, given two consecutive zones \( f \) and \( l \) in \( Z \), there always exists a zone \( z \) in the partition tree that \( f \) is the rightmost leaf of the partition tree \( T_0 \) of its left child \( z_0 \) and \( l \) is the leftmost leaf of the partition tree \( T_1 \) of its right child \( z_1 \). More interesting, \( f \) must be on the upper-right corner of \( z_0 \) and \( l \) must occupy the lower-left corner of \( z_1 \). It means \( f \) and \( l \) lie on the opposite sides of the border of \( z_0 \) and \( z_1 \). So if partition tree of \( z \) has a few leaf zones, it is very likely that \( f \) and \( l \) are neighbors. Thus, sorting zone by zone ID ensures locality in small groups of consecutive zones.

We embed zone ID assignment and zone rearrangement into zone splitting process and describe it in Algorithm 1.

Algorithm 1 Zone construction

1: function ZoneConstruction
2: \( \text{return split} (\text{"}\), \( z_0) \)
3: end function

4: \( \text{if } |V_{z_0}| < \theta \) \( \text{then} \)
5: \( \text{return} (z) \)
6: \( \text{else} \)
7: \( \Delta \leftarrow \infty \)
8: \( \text{for each} (t, k) \in E_z \) do
9: \( \text{if } t = 0 \) \( \text{then} \)
10: \( z_0 \leftarrow (z.bl, (z.ur,i,k-1)) \)
11: \( z_1 \leftarrow ((z.bl,i,k), z.ur) \)
12: \( \text{else} \)
13: \( z_0 \leftarrow (z.bl, (k-1, z.ur,j)) \)
14: \( z_1 \leftarrow ((k, z.bl,j), z.ur) \)
15: \( \text{end if} \)
16: \( \text{if } |V_{z_0}| - |V_{z_1}| < \Delta \) \( \text{then} \)
17: \( \Delta \leftarrow |V_{z_0}| - |V_{z_1}| \)
18: \( z_1 \leftarrow z_0 \)
19: \( z_0 \leftarrow z_1 \)
20: \( \text{end for} \)
21: \( \text{return} \text{split}(zid_{z_0}, z_1) \cup \text{split}(zid_{z_1}, z_f) \)
22: end function

As zone partitioning algorithm tries to minimize the density gap between two child zones when dividing their parent, same level zones in the partition tree tend to have similar density. However, the outcome consists of only leaf zones so there exists the significant density gap between zones with different level. Fortunately, this gap can be estimated beforehand. Difference in zone level occurs when zone \( z \) is split into \( z_0 \) and \( z_e \) but \( |V_{z_0}| > \theta \) so \( z_1 \) is split one more time while \( V_{z_0} \leq \theta \) and \( z_e \) is put into \( \zeta \). Since the density gap between \( z_0 \) and \( z_e \) is kept as equal as possible, even though \( |V_{z_0}| > \theta \), it still tends to close to \( \theta \) as \( V_{z_0} \leq \theta \). Similarly, density of \( z_e \)’s children will tend to close to \( \theta/2 \) and obviously, they should not be split any more. Therefore, given threshold \( \theta \), zone density of the outcome of zone partitioning algorithm will cluster around the range from \( \theta/2 \) to \( \theta \).

Figure 3 illustrates an example of zone splitting process on a \( 4 \times 4 \) grid with \( \theta = 8 \). Apparently, there are six possible cuts in total. Among those, the cut \( \langle 1,2 \rangle \) will produce two child zones with the smallest density gap. Thus, we split the zone according to this cut which generates two new child zones as shown in Figure 3b. Apply the same strategy to both of them, we obtain new four zones (Figure 3c) and since all of their density are smaller than \( \theta \), zone splitting process terminates. The process is tracked by the partition tree drawn in Figure 3d. Internal zones are eliminated from the outcome leaving leaf zones (00, 01, 10, and 11) as the final result. Obviously, the density gap in final result is much lower than that of uniform grid.
5.3 Load Distribution Policy

As $Z$ contains zones whose density is various, load balancing cannot be achieved by simply equally distributing them into data nodes. We tackle this issue based on capacity because it is a major feature of a storage node. Particularly, load distribution strategy ensures load balancing by guaranteeing the amount of data each data node obtains approximately equals to its capacity.

We consider the density of sample data in a zone as the amount of load this zone contributes to the total load on data node and distribute zones based on this criterion. As the density of zones with the same level tends to be equal to each other, we use an additional parameter, called load weight, to represent the load contribution of zones. Let $H$ be the highest level of zones in $Z$, every zone $z$ has level $h_z$ will have load weight $w_z$ determined as follows:

$$w_z = 2^{H-h_z}$$  \hspace{1cm} (6)

Equation 6 ensures load weight of zones at the same level is the same and equal to a half of that of their parent. Those features reflect the nature of zone structure.

Suppose data node $s_i$ has capacity $c_{pi}$ which is the maximum number of records it can keep in memory. Let $W$ be the total load weight of zones in $Z$ and $CP$ be the total capacity of the whole system, so $W = \sum_{z \in Z} w_z$ and $CP = \sum_{i=0}^{N-1} c_{pi}$. We separate $Z$ into $N$ disjoint sub-lists $l_0, l_1, ..., l_{N-1}$ where $l_i$ contains zones assigned to data node $s_i$. Load should be scattered to data nodes according to their capacity as follows.

$$\forall i \leq N: \sum_{w_z \in l_i} \frac{w_z}{W} \approx \frac{c_{pi}}{CP}$$  \hspace{1cm} (7)

We consider the condition described in equation 7 as a load distribution policy.

Beside load balancing, locality should also be reserved to avoid request being sent to so many data nodes causing communication overhead and producing too much load. Since consecutive zones in the outcome $Z$ tend to be close to each other, load distribution process should scatter zones across storage system by group of adjacent zones.

6. DATA MANIPULATION

6.1 Insertion

As SIDI is fully decentralized, new data point $p \in P$ can be sent to every data node in the system. After receiving $p$, the node will determine the right data node for $p$ and forward it to this node. This task can be seen as translating $p$ to a data node $s \in S$. In each data node, we construct additional translation tables to translate cell to zone ID and zone ID to address of data node holding this zone based on the zone distribution policy. Thus, to find proper location for $p$, we first calculate its corresponding cell using Equation 2 and 3. After that, we lookup translation tables to determine $s$ and forward $p$ to it. $s$ then puts $p$ to the storage and updates its pointers in both cell and zone layer.

6.2 Query Processing

According to zone ID assignment process, given a zone $z$, every zone intersecting with the area bounded by its lower-left point and upper-left point of $z$ (area labeled $c$ in Figure 4a) must have zone ID greater than that of $z$. Similarly, zones intersecting with region bounded by $z$’s upper-right point and $z$’s lower-left point (Figure 4b) must have ID greater than that of $z$. Therefore, given two disjoint zones $s$ and $f$ (Figure 4c), every zone intersects with the area bounded by the upper-right point of $f$ and lower-left point of $s$ must have zone ID falling into the range between the ID of $s$ and $f$.

Thus, we process range queries as follows. For each query, we search for zones containing its lower-left point and upper-right point. Every zone $z$ in $Z$ whose zone ID falls into the range of two boundary zones is checked to extract data. If $z$ is contained entirely by the query then every point in $z$ is put into the outcome set. If $z$ partially intersects with the query, then cells related to the overlapped area are checked. For cells entirely contained by the query, we put their data to the outcome set while points in cells partially overlapping the query are scanned and only those falling into the query are put into the outcome set.

An example of range query processing is depicted in Figure 5. Initially, we use Equation 2 and 3 followed by accessing translation table to determine zones covering the lower-left and upper-right points of the range query. After that, all zones whose ID is in the range between their IDs are tested for intersection. Data in zone 0101 does not take further assessment since their zone does not intersect with the query. In contrast, all data points inside zone 1100.
are added directly into the result as their zone is entirely contained by the query. Other zones partly overlap with the query so their cells must take further intersection test. However, we just apply the test on cells overlapping with query. For instance, in case of zone 10, the query intersects with only three upper-left cells so only data points in those zones are tested for intersection. During the test, data points whose coordinates are bounded by the query are added to the result.

7. EXPERIMENTS

7.1 Index Construction

To investigate the applicability of SIDI, we performed partitioning process on both uniform and skewed datasets. In each case, we randomly generated 8 million records in two-dimensional data space having a domain of [0, 1000] to form sample sets. We also constructed SIDI over real dataset consisting of GPS signals collected from buses in Ho Chi Minh City to demonstrate the effectiveness of SIDI on practical application. As buses travel according to scheduled routes, we selected signals generated within a day to form sample dataset. We carried out grid construction process with three different resolutions: sparse (η = 1.25%), medium (η = 0.5%), and dense (η = 0.125%). θ was equal to the production of η and the size of sample set for all settings. Results of index construction with medium setting are demonstrated on Figure 6.

The histogram of zone density distribution is shown in Figure 7. The result verified the prediction in previous section that most of zones’ density varies from θ/2 to θ. In addition, uniform dataset achieved best results while those of skewed dataset were the worst. This is because in skewed dataset, the density dropped significantly when moving far away from hotspots. It was difficult to select a proper cut that can divide the spaces around those points into two parts with similar density. As a result, density of child zones would be imbalanced. Even worse, this phenomenon also cascaded to further splits and caused a wide range of zone density distribution. Meanwhile, uniform dataset did not suffer from such difficulty as points were scattered evenly that made it easier to minimize density gap.

The real dataset was also skewed but points tended to cluster around areas instead of “hot” points. Hence, results from real dataset stood in the middle between the two extremes. Although the density varied over a wide range, its distribution formed a bell shape as most of values cluster around one point between θ/2 and θ. This result was good enough to distribute load evenly over the storage system.

7.2 Levels of Granularity

In this experiment, we analyzed the impact of partition schemes on query performance. We constructed the index with six selective pairs of parameter, named S-1, M-1, M-2, D-1, D-2, and D-8. The prefix letters denoted the levels of granularity of the grid, which were Sparse (η = 1.25%), Medium (η = 0.5%), and Dense (η = 0.125%). The postfix numbers indicate the resolution of zone layer, i.e., for D-8, θ = 8 × η× size of sample set. We conducted the experiment on a single machine equipped with an Intel Core i5 3470 3.20 Hz and 8 GB memory. During the test, 8 million data points collected from bus dataset were preloaded into index
structures then we generated random range queries and used them to evaluate the performance of each setting. Results of the experiment are shown in Figure 8.

We define the range as the percentage of the entire space. Apparently, query throughput increased as the size of query increased in all cases because executing large query enabled zone layer to rule out a lot of unnecessary intersection tests on their cells. Among settings, Medium schemes achieved the best performance as cells in Sparse scheme were too large so that the index had to perform many intersection tests. Meanwhile, Dense schemes were outperformed by others since they had to spend so much time on cell calculation.

It is noteworthy that increasing zone size just improves query throughput in some cases but degrades the performance in the rest. The reason is that when zone enlarges, the chance that it is entirely contained by the query’s scope decreases. The rise of zone size also increases the cost of cell calculation as the number of cell inside each zone increases. Thus, although reducing the resolution of zone layer eliminates more intersection tests in case related zones are within query’s scope, the overall performance still drops.

7.3 Performance Evaluation

We evaluated SIDI’s insertion and range query performance in a single machine and compared it with the R-Tree and QuadTree. Although those trees are relatively old, we still used them as reference indexes because they are still widely used and recent approaches are also based on their structure. We conducted the experiment on bus dataset containing 64 million GPS signals. The data node was equipped with an Intel Xeon E7 4870 2.4GHz and 32 GB memory. SIDI was constructed from D-1 setting in both experiments.

As Figure 9 shows, R-Tree and Quad-Tree did not handle insertion well due to overhead of splitting nodes. SIDI, however, achieved the best performance since it did not pay additional cost to reconstruct the index organization.

In range query experiments, we considered three separated selective ranges: small (0.01%), medium (0.25%) and large (1%). Queries were generated randomly using uniform distribution. Figure 10 illustrates query performance of those structures on different ranges. QuadTree was the worst in all cases since the skewed distribution had it traverse so many branches that slowed down the search process. In spite of being beaten up by R-Tree at processing small queries, SIDI outperformed other indexes in medium and large test cases. The result was predictable as enlarging the scope of queries helped reduce many cell calculations.

7.4 Scalability

To evaluate scalability, we deployed SIDI over a cluster of 12 data nodes connected by 1 GB Ethernet. Each data node was equipped with an Intel Xeon E312xx and 4 GB memory. During the test, real dataset was used and range queries were randomly generated using uniform distribution. We scattered data across the system according to load distribution policy described in Section 5. Zones were grouped by load weight then each group was distributed to data nodes in round-robin manner.

Figure 11 illustrates the effect of load distribution policy on insertion for varying number of nodes. The throughput increased almost linearly that verifies the SIDI’s scalability. The reason behind linear scalability is that zone
structure is fully decentralized allowing insertion to be easily parallelized. Furthermore, data is evenly distributed to data nodes so as the number of data nodes increases, the amount of data to be processed in each node decreases that leads to performance gain.

As expected, SIDI also ensures query scalability as being shown in Figure 12. Still, there were some cases that query throughput drops slightly when adding more data nodes. This is because range query execution time is dominated by both scanning and intersection test. Even though increasing the number of data node will reduce the number of zone scans in each data nodes, its effect on intersection tests is unpredictable. Since zones contained by or not intersecting with the query scope do not require further tests, most of intersection test occurs in zones that partly overlap with the query. Unfortunately, selecting those zones entirely depends on the query itself. Thus, changing the number of data nodes may cause performance degradation if most of zones partly intersecting with the query are assigned to the same node.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a novel index, SIDI, to facilitate data processing on spatial databases. Distinguished from other index structures, SIDI construction is based on exploitation of the stable spatial distribution of the datasets. SIDI's layout is sketched beforehand by exploiting the spatial characteristics of the dataset so that its structure fits well with incoming data without paying extra cost for maintenance. By partitioning data space into independent zones according to density, SIDI scales well across distributed systems and allows dynamic balancing with low cost. Keeping the whole structure in memory with two-layer index helps SIDI offer better read and write performance than other existing approaches in many cases.

However, there are still drawbacks in our approach. SIDI is inappropriate to datasets whose spatial distribution frequently changes due to its rigid organization. Furthermore, re-partitioning the index structure periodically is simple and inflexible. Thus, we are planning to develop a new module to help SIDI detect the change in spatial distribution of the dataset in time. Besides, SIDI performance still has the potential to improve such as implementing space-filling curves to enhance cache utilization or deploying multiple grids under zone layer to reduce the number of intersection tests. Several compression solutions should be considered to reduce SIDI's memory consumption.

9. ACKNOWLEDGMENTS

This research is funded by Vietnam National University Ho Chi Minh City under grant number B2014-20-07.

10. REFERENCES


