IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 September 2023, accepted 13 October 2023, date of publication 20 October 2023, date of current version 30 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3326256

RESEARCH ARTICLE

Hierarchical Quadrant Spatial LSM Tree for Indexing Blockchain-Based Geospatial Point Data

JUNGHYUN LEE, TAEHYEON KWON, MINJUN SEO, AND SUNGWON JUNG
Department of Computer Science and Engineering, Sogang University, Seoul 121-742, Republic of Korea

Corresponding author: Sungwon Jung (jungsung@sogang.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) through the Korean Government [Ministry of Science and ICT (MSIT)], Development of High-Speed Analysis of Distributed Large-Scale Data for Wide Usability of Various Industries, under Grant 2021-0-00180; and in part by the Korea Institute for Advanced of Technology (KIAT) funded by the Korean Government [Ministry of Trade, Industry and Energy (MOTIE)] through the Human Resource Development (HRD) Program for Industrial Innovation under Grant P0020535.

ABSTRACT Blockchain technology is attracting attention for its high usability in various fields such as IoT and healthcare, and it is being used as an alternative to distributed databases. Despite their high usability, the techniques for efficiently indexing blockchain-based geospatial point data have not been studied much until now. In this paper, we propose a hierarchical quadrant spatial LSM tree(i.e., HQ-sLSM tree) which effectively indexes large amounts of geospatial point data from the blockchain by reflecting its write-intensive feature. The geospatial point data is linearized using Geohash before being inserted into our proposed HQ-sLSM tree. Furthermore, we present the concept of a spatial filter which enables low disk I/O generating algorithms to process range and kNN queries over the HQ-sLSM tree. Experiments confirmed that the HQ-sLSM tree performed exceptionally well for the insertion of point data, and the performance of range and kNN query processing resulted second to the best after the R-tree.

INDEX TERMS Blockchain, component table, hierarchical quadrant spatial LSM tree, kNN query, LSM tree, range query, spatial filter.

I. INTRODUCTION

As the prevention of forgery and falsification of geospatial point data stored in real time has become a core requirement for various geospatial data services, blockchain-based geospatial data services are attracting a lot of attention recently. IBM has developed a blockchain-based food distribution information system [14] that allows consumers to track every step of delivery. A spatial blockchain system FOAM [9] has developed a transaction service protocol that combines blockchain addresses and geographic point information. British university has developed a model that responds to disasters by combining blockchain decentralized P2P with drones and autonomous vehicles [4]. In addition, it is being used in various fields that require reliability and openness of information such as land transactions [3], IoT [7], [29], and healthcare [4], [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Sookhak.

Numerous spatial indexing for geospatial spatial point data such as the KD-tree [34] and Quad-tree [8] exist. However, these structures are not optimized for write-intensive workloads. In the case of the KD-tree, it is a memory-based spatial index structure which is not suitable for indexing massive amount of geospatial point data stored on a disk-based blockchain.

For the Quad-tree, the data space is recursively decomposed into quadrants until the number of points in each quadrant is less than the page capacity. However, it is not suitable for indexing geospatial point data stored in the blockchain since the insertion of a data point to a Quad-tree incurs disk I/Os every time, leading to a large number of disk I/Os in a write-intensive environment. This brings up the need to research effective spatial indexing techniques for geospatial point data stored in blockchains with a write-intensive feature.

Blockchain-based geospatial information service has different characteristics from the data management environment

assumed by existing geospatial databases [10], [30]. The first characteristic is the immutability of data. In other words, data stored on the blockchain is not deleted or changed. Therefore, as the service continues, the size of geospatial data in the blockchain increases, and we need an efficient disk-based spatial index for these massive amount of immutable geospatial data. Second, a blockchain is mostly managed in a write-intensive environment where data blocks holding large amounts of data are inserted in real time. Although there is some difference in the TPS(Transactions per second) between a centeralized blockchain system [20] such as VISA, PAYPAL and a decentralized blockchain system which are mainly used for cryptocurrency systems such as Ethereum [36], insertion of data is the majority action related with the blockchain. Therefore to index spatial data from the blockchain, we need an index structure that perfoms well against a write-intensive environment with large amounts of data insertion.

A spatio-temporal blockchain query processing method based on Merkle Block Space Index(BSI) was proposed [28]. The BSI is a modification of the Merkle KD-tree [21]. It was developed to only index spatial point data stored in each block of the blockchain. A separate BSI index is stored in each block to support a fast local search for point data in the block. Thus, BSI does not support indexing the entire point data in the blockchain, which means it cannot perform a global search. Various database systems use the R-tree [12] as a spatial data indexing technique. However, disk-based R trees are not optimized for indexing blockchain-based geospatial data due to the large number of random disk I/Os generated in the insertion process.

The LSM-tree [27] and its improved variations such as the LSM B-tree [31] are a non-spatial data indexing technique widely used and optimized for write-intensive environments. Some examples of databases that use the LSM-tree structure include AsterixDB [1], Google BigTable [5], Cassandra [18] and Apache HBase [2]. There were several approaches to utilize or expand the LSM-tree to cope with frequent spatial data insertions and to efficiently support global spatial data search.

One approach based on LevelDB [11], was to store spatial data in a tree-like supplementary index [23], [37]. Instead of directly searching the LSM-tree when processing a query, the supplementary index is searched first. Although this method provides a solution for indexing multi-dimensional data, two weaknesses make this approach not suitable for our needs. First, the need for reading two index trees to process a query triggers read amplification. Second, the supplementary index tree also needs to be merged when the components of LSM-tree are flushed and merged causing a large update cost. Recently, an embedded R-tree(ER-tree) [13] was proposed which improved the two weaknesses of using a supplementary index tree. However, the ER-tree is not suitable for practical applications due to the fact that it provides a decent performance only on a read-intensive workload.

Another approach was to expand the LSM-tree structure itself to directly index multi-dimensional data. As a result, the LSM R-tree [1], [16], [33], the Dynamic Hilbert B^+-Tree(DHB-tree) [19] and the Dynamic Hilbert-Valued B^+-Tree(DHVB-tree) [19], SHB-tree [17], SIF [17] index have been proposed.

A comparative performance analysis of these indexing methods has been conducted [17], [24], [25], [26]. In this study [17], DHB-tree and DHVB-tree were implemented using the main ideas from [19], which explains in detail about indexing spatial data points using a B-tree with space-filling curves and methods to support spatial range queries.

From the index trees mentioned above, the LSM R-tree was shown to perform better than the DHB-tree, DHVB-tree, SHB-tree, SIF in terms of the disk I/O cost required for data insertion and processing similarity (i.e., range and kNN) queries. However, since a single R-tree at each disk level covers the entire spatial data space, disk I/Os necessary for spatial data insertion and similarity query processing of LSM R-tree are still large.

To reduce the disk I/O costs of the existing spatial indexing schemes mentioned above, we propose a disk-based external index tree which resides outside a blockchain in this paper. The index tree is named the Hierarchical Quadrant spatial LSM tree(i.e., HQ-sLSM tree). We introduced the initial version of the HQ-sLSM tree in [32], with the name spatial LSM tree(i.e., sLSM tree). The sLSM tree lacked detail in the index structure and the similarity query processing algorithms were not given. We improved the sLSM tree to the HQ-sLSM tree by refining and optimizing the index structure to support the efficient processing of range and kNN queries.

The HQ-sLSM tree is a spatial extension of the LSM-tree to index geospatial point data. The coordinates of the point data are linearized via Geohash [22]. It also maintains the separate B-trees for all the quadrant spaces at each disk level of the LSM-tree, whose structure significantly reduces disk I/Os needed for geospatial point data insertions and spatial similarity query processing.

The key contributions of our proposed method are summarized as follows:

- We propose a disk-based external index named HQ-sLSM tree to efficiently index and handle similarity queries on massive amount of geospatial point data stored in a write-intensive blockchain environment where large amounts of point data insertions are generated in real time.
- We propose the disk I/O cost-effective range and kNN query processing algorithms based on the HQ-sLSM tree.
- We develop spatial filters for the HQ-sLSM tree to avoid traversing its unnecessary disk components when processing range and kNN queries.

The remainder of this paper is organized as follows. Section II analyses and compares related research works regarding using the LSM-tree structure to index spatial data. Section III describes the detailed structure of the HQ-

sLSM tree including its component table and spatial filters. Furthermore, the algorithm for inserting geospatial point data into the HQ-sLSM tree is also given in Section III. In Section IV and section V, we present the range and kNN query processing algorithms for the HQ-sLSM tree. The extensive experimental results of our algorithm is analyzed in Section VI. Finally, Section VII provides our concluding remarks.

II. RELATED WORKS

This section discusses the relevant existing research works on indexing spatial data using the LSM-tree structure. The LSM-tree [27] structure is proven to perform well for write operations. The one problem with this efficient structure is that multi-dimensional data cannot be indexed directly. Two approaches to utilize the LSM-tree structure for indexing spatial data were briefly mentioned in section I.

Many index trees that inherit the characteristic of the LSM-trees have been developed. By using the R-tree [12] widely used in spatial databases, a LSM R-tree [1], [16], [33] was developed to improve its performance for insertion and query processing. An LSM R-tree consists of an in-memory component and zero or more disk components. An in-memory component of an LSM R-tree consists of a traditional R-tree along with a deleted-key B^+-tree that captures deleted entries. A disk component is a variant of an R-tree, where it orders indexed entries using a Hilbert curve [15] when loading the tree. AsterixDB [1] is one of the main databases that currently uses a LSM R-tree.

The Dynamic Hilbert B^+-Tree(DHB-tree) [19] and the Dynamic Hilbert-Valued B^+-Tree(DHVB-tree) [19] use the Hilbert space filling curve and a B^+-tree to index spatial data. The two spatial indexes can be applied to the LSM-tree, which makes the direct indexing of point data possible. However, the computation of Hilbert values for every point triggers a large overhead in the insertion process. Thus, the LSM R-tree, DHB-tree and the DHVB-tree is not an efficient structure for indexing spatial point data in a blockchain environment.

The SHB-tree [17] is also based on an underlying LSM B-tree index. It uses the Hilbert curve and a grid based approach to index spatial data. The SIF [17] supports spatial indexing based on an LSM inverted index. The focus of the SHB-tree and SIF are set more towards a range query. It also tends to be more suitable for indexing spatial objects other than points because the point data is always only indexed in the last level of the SHB-tree.

Using a secondary index outside the LSM-tree was an another solution with a few drawbacks. The embedded R-tree(ER-tree) [13] was proposed as an efficient method for indexing spatial data while using a secondary index by improving the drawbacks. The ER-tree is in the form of a LSM-tree, built from a SER-tree index(embedded R-tree on a SSTable). It uses the Hilbert space filling curve [15] to sort the data points, and the disk nodes consist of a integration between the R-tree index and a SSTable. Rather than arranging the secondary index outside the ER-tree, it acts as if the index is inside the ER-tree. This architecture reduces the read amplication that occurs from reading two index trees in order to process a query. Experiments in the paper state that the ER-tree showed better results in query performance compared to the LSM R-tree.

Although the ER-tree uses the LSM-tree structure optimized for insertion, the R-tree indexes in the disk nodes trigger high overhead as the write ratio in a workload increases. While it may be efficient in processing queries, the ER-tree is not suitable to use in a write intensive blockchain environment, leading to the necessity of our proposed HQ-sLSM tree. Table 1 shows the characteristics of the indexing methods introduced above and our approach in a tabular form.

TABLE 1. Indexing methods for spatial data.

Index	Characteristics
LSM R-tree [1], [16], [33], DHB-tree [19], DHVB-tree [19]	Hilbert space filling curves. Expansion of the LSM-tree. Direct indexing of spatial data is possible using the LSM-tree. Large overhead in insertion by computing Hilbert values.
ER-tree [13]	Hilbert space filling curves. Secondary index approach. Not suitable for write-intensive workloads due to high update overhead. Focuses on query processing.
SHB-tree [17], SIF [17]	Hilbert space filling curves. Based on the LSM-tree. Grid structure. More suitable for spatial objects other than points.
Our approach - HQ-sLSM tree	Z-order space filling curves. Hierarchical quadrant space decomposition. Well suits a write-intensive environment. Efficiently processes range/kNN queries using the spatial filters and query filters.

III. HIERARCHICAL QUADRANT SPATIAL LSM TREE

This section describes the structure of the hierarchical quadrant spatial LSM tree (i.e., HQ-sLSM tree) along with its component table and spatial filters. This section also presents the algorithm for inserting geospatial point data into the HQ-sLSM tree, which was developed based on the disk I/O cost-effective component flush algorithm.

A. BASIC STRUCTURE OF THE HQ-sLSM TREE

As shown in Figure 1,the HQ-sLSM tree has two parts, the memory and disk. The memory part consists of one memory component and a component table which will be dealt in detail in III-C. The memory component (i.e., C_0) contains the most recent point data extracted from the blockchain in the form of a B-tree index. The disk part consists of disk components, where each disk component corresponds to a quadrant space at each hierarchical level. Likewise memory, a disk component contains the index for the distributed geospatial point data in the form of a B-tree.

Figure 2 shows an example of the hierarchical quadrant space decomposition in the HQ-sLSM tree. The memory component indexes the most recently inserted point data.

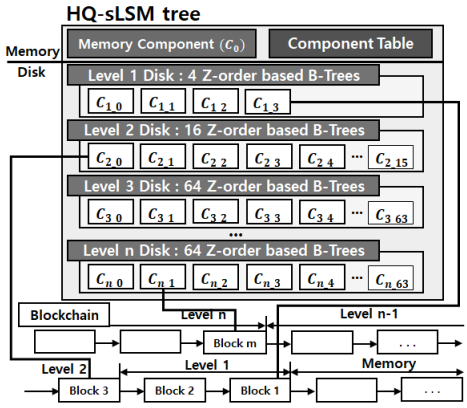


FIGURE 1. Structure of the HQ-sLSM tree for Blockchain-based geospatial point data.

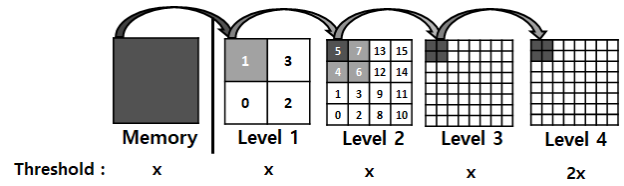


FIGURE 2. Hierarchical quadrant space decomposition for disk levels 1 to 4 with MDL 3.

When the size of the memory component exceeds a threshold x, the indexes in memory are partitioned into four quadrant space groups and flushed to level 1. Then, the flushed index from memory are merged with the four Level 1 quadrant space disk components(i.e., $C_{1,0}$, $C_{1,1}$, $C_{1,2}$, $C_{1,3}$) accordingly. Similarly, when the size of a disk component in level i exceeds a given threshold, the index of the disk components are partitioned into four, gets flushed and merged with level $i+1$ disk components. For example, if the size of the disk component $C_{1,1}$ in level 1 disk exceeds the threshold x, it is flushed and merged with the four level 2 disk components (i.e., $C_{2,4}$, $C_{2,5}$, $C_{2,6}$, $C_{2,7}$) corresponding to the quadrants 4, 5, 6, 7 at level 2 space as shown in Figure 2.

Each disk component in a level i is given a number(i.e., a geohash value of a component) ranging from 0 to $4^i - 1$. The geohash for the component uses a $2 \cdot i$-bit binary number. The component numbers in Figure 2 are shown as decimal numbers.

The hierarchical quadrant space decomposition increases the number of components in each level. Thus without any further restrictions, the total number of disk components of the HQ-sLSM tree will increase as the tree gets deeper. Along with it, the size of the component table holding metadata for the components and the maintenance overhead of the tree will get bigger as well.

In order to reduce the maintenance overhead, we set a limit on a disk level to stop the decomposition of quadrant space. We name this disk level as Maximum Decomposition Level (MDL). The formal definition of MDL is given in the following definition 1.

Definition 1: Given a HQ-sLSM tree with its component threshold $\mathbf{x}$, the disk level where the hierarchical quadrant space decomposition is no longer allowed is defined as the **MDL**(Maximum Decomposition Level) of the HQ-sLSM tree. The threshold of all the disk components at level i disk where $i > MDL$ is set to $2^{(i-MDL)} \times \mathbf{x}$.

Definition 1 states that the component space is no longer decomposed into smaller quadrants after MDL. Instead, the threshold for each component increases. Before the MDL, the threshold value for the components remain the same.

For example, Figure 2 shows the structure of the HQ-sLSM tree when the MDL is set to 3. From level 4 and onwards, the space does not get decomposed. The thresholds for all disk components up to level 3 are set to x. Threshold for disk components at level 4 are set to $(2^{(4-3)} \times x) = 2x$, which is twice the size of the threshold of level 3. If disk components of level 5 disk are added, their thresholds will be set to $(2^{(5-3)} \times x) = 4x$, which is twice the size of the threshold of level 4.

An node in the B-tree of the component has two attirbutes (*Geohash*, *Blockaddress*) to index the point data. The *Geohash* is a z-order number of point (x,y). *Geohash* becomes the search key used for building the B-tree each component $C_{i,j}$ maintains. Note that unlike the geohash values of disk components, a 32-bit binary number was used as a geohash value to linearize each point in this paper. The *BlockAddress* is the address of the blockchain where point data (x,y) and its associated additional data are stored.

B. SPATIAL FILTERS

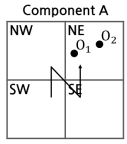
Each component in the HQ-sLSM tree has a spatial filter which indicates the existence of a point data in the four sub-quadrants of its component space. A spatial filter is a 4-bit string in the form of $b_0b_1b_2b_3$ where b_0, b_1, b_2, and b_3 represent SW (South West), NW (North West), SE (South East), and NE (North East) sub-quadrants of a component space.

Figure 3 shows an example of a spatial filter of a given component A. The sub-quadrant space of the component A is shown in Figure 3a. At this point, only 2 data points(O_1, O_2) exist in the NE sub-quadrant. Therefore, the spatial filter for component A is 0001 as shown in Figure 3c. Figure 3b shows the insertion of point O_3 into the SW sub-quadrant. After this insertion, the spatial filter for component A will be updated to 1001 as shown in Figure 3d.

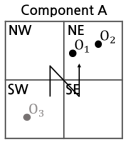
Spatial filters play a key role in reducing disk I/O-costs for processing range and kNN queries along with the hierarchical structure of the disk components and the quadrant space decomposition structure of the HQ-sLSM tree. With the spatial filter, it is possible to avoid accessing unnecessary disk components which result in a reduce of disk I/O costs. This will be dealt in sections IV and V.

C. COMPONENT TABLE

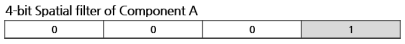
The component table is managed in memory and maintains metadata for all the components in the HQ-sLSM tree.



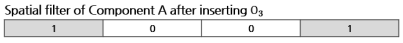
(a) **Point data indexed by component A**



(b) **Point data indexed by component A after inserting O_3**



(c) **The spatial filter of component A**



(d) **The Spatial filter of component A after inserting O_3**

FIGURE 3. **An example of the spatial filter of a component.**

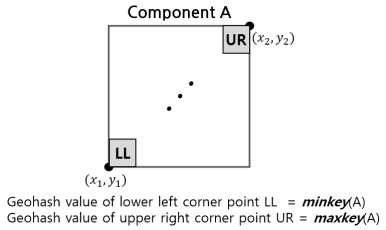


FIGURE 4. **Range of geohash values in a component.**

TABLE 2. **Column attributes for components in the component table.**

Atttibute : Notation	Description
Disk level: *dl*	the disk level number
lower bound: *lb*	the geohash value of LL
upper bound: *ub*	the geohash value of UR
minimum key : *min*	**minkey** value of the lower left corner point of LL
maximum key : *max*	**maxkey** value of the upper right corner point of UR
Spatial filter: *sf*	the spatial filter
Component address: *C_addr*	the disk address
Number of data: *num*	the number of data stored
Threshold: *th*	the threshold

The information in the component table helps to avoid unnecessary disk component access while processing range and kNN queries. Before creating the component table, a few terms need to be defined.

To utilize the spatial filter and the query filter explained in section III-B and , we define the *smallest unit quadrant*.

Definition 2: Given a HQ-sLSM tree with its *MDL*, the **smallest unit quadrant** is a quadrant with the area of dividing the total space into 4^{MDL+1} quadrants.

The smallest unit quadrants are also each given a geohash value. In each disk component of each level, we are able to obtain the range of the smallest unit quadrant geohash values that fall into the component. Going back to Figure 2 for example, component 0 in level 1 will contain the smallest unit quadrants numbered from 0 to 63. The component 0 in level 2 will contain 0-15, and component 1 of level 2 will contain 16-31 smallest unit quadrants respectively. This range of smallest unit quadrants is named the unit quadrant range of a component, and it is stored in the component table.

For a component, we define LL and UR in the following definitions 3 and 3. We illustrate the examples of the two definitions in Figure 4.

Definition 3: Given a component A in a HQ-sLSM tree, and let **LL** be the smallest unit quadrant at the lower left corner of A. **minkey**(A) returns the geohash value (32-bit binary number) of the lower left corner point (x,y) of LL of A.

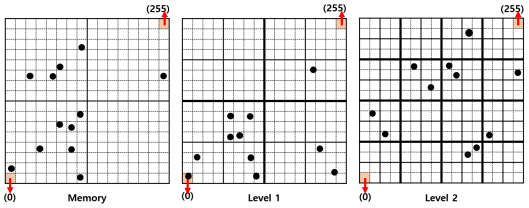
Definition 4: Given a component A in a HQ-sLSM tree, and let **UR** be the smallest unit quadrant at the upper right corner of A. **maxkey**(A) returns the geohash value (32-bit binary number) of the upper right corner point (x,y) of UR of A.

As shown in Figure 4, the lower left corner unit quadrant is named *LL*, and the upper right corner unit quadrant is named *UR*. The geohash value of *LL* and *UR* becomes the minimum and maximum geohash values for the component. Thus, the geohash values inside the component will be in between the minimum and maximum. We define the two functions to compute this minimum and maximum geohash values for a given component. The formal definition of these two functions are also given in 3 and 4. The column attributes of the component table are described in table 2.

Figure 5a shows an example of geospatial point data distribution indexed by an HQ-sLSM tree in memory, level 1, and level 2. The corresponding component table is given in Figure 5b. In this example, the *MDL* is set to 3, therefore 256 smallest unit quadrants are created with their geohash values ranging from 0 to 255. The first row of the component table stores information about the memory component. Since the memory component covers the entire space, its lower and upper bound for the unit quadrant becomes 0 and 255. The minimum and maximum geohash key values for the memory becomes *minkey*(0) and *maxkey*(255). The spatial filter is stored as 1101 because the 12 point data shown in the leftmost example of Figure 5a are distributed in the SW, NW, and the NE sub-quadrants.

The second to the fifth rows of the component table store the metadata of the four level 1 disk components(i.e., $C_{1,0}$, $C_{1,1}$, $C_{1,2}$, $C_{1,3}$) which currently indexes 11 point data. The second row has the meta data of level 1 $C_{1,0}$ disk component, which indexes the point data in the quadrant space with the unit quadrant range of 0 to 63. The minimum and maximum key for the component are *minkey*(0) and *maxkey*(63). Its spatial filter is stored as 1011 because the 8 point data shown in the middle example of Figure 5 are distributed in the SW, SE, and NE sub-quadrants of its component space.

The third row has the meta data of $C_{1,1}$ level 1 disk component, which is indexing the point data in the quadrant space with the unit quadrant range of 64 to 127. The minimum and maximum key for the components are *minkey*(64)

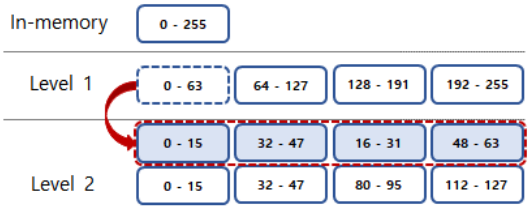


(a) **An Example of Geospatial Point Data Distribution**

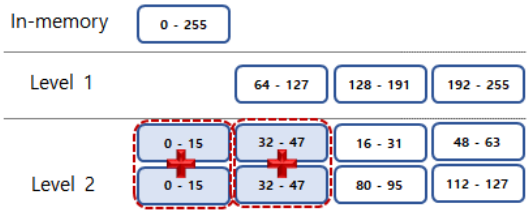
Disk level (dl)	Unit quadrant range		Key range		Spatial filter (sf)	Component address (C_addr)	Number of data (num)	Threshold (th)
	Lower bound (lb)	Upper bound (ub)	Minimum Key (min)	Maximum Key (max)				
0	0	255	*minkey*(0)	*maxkey*(255)	1101	0x12	12	12
1	0	63	*minkey*(0)	*maxkey*(63)	1011	0x15	8	12
1	64	127	*minkey*(64)	*maxkey*(127)	0000	Null	0	12
1	128	191	*minkey*(128)	*maxkey*(191)	0010	0x5f	2	12
1	192	255	*minkey*(192)	*maxkey*(255)	0010	0x62	1	12
2	0	15	*minkey*(0)	*maxkey*(15)	0000	Null	0	12
2	16	31	*minkey*(16)	*maxkey*(31)	0110	0x67	2	12
2	32	47	*minkey*(32)	*maxkey*(47)	0000	Null	0	12
2	48	63	*minkey*(48)	*maxkey*(63)	0000	Null	0	12
...	...	...	...	...	...	...	...	...

(b) **Example of a Component table**

FIGURE 5. An example of data distribution indexed by HQ-sLSM tree with a component table.



(a) **An Example of Flushing a Component**



(b) **An Example of Merging Components**

FIGURE 6. Flush and merge of components.

and *maxkey*(127). Its *sf*, *C_addr*, and *num* are stored as 0000, NULL, and 0 because no point data exists in the corresponding component space. The sixth row and thereafter rows of the component table store the metadata of the sixteen level 2 disk components (i.e., $C_{2,0}$, $C_{2,1}$, ..., $C_{2,15}$). Since the component space is continuously being decomposed into quadrants, the threshold value 12 remains the same up until level 2.

D. INSERTION OF POINT DATA

An update is needed in the HQ-sLSM tree when a new block is added to the blockchain. The point data contained in the new block are indexed in the memory component first. If the size of the memory component exceeds its threshold during data insertion, the memory component is flushed and merged to the level 1 disk components. The disk components at the lower level are also flushed and merged with their next level in a similar manner when they exceed their thresholds.

Algorithm 1 insert_data($P = (x, y)$, *b_addr*)

INPUT *P*:: a point (x, y)
b_addr:: the block address

1: *Global Component Table* CT;
2: *int* geo; ▷ geohash value of point
3: *geo* = *GeoHash*(*P*); ▷ compute the geohash value of point *P*
4: Let B_M be the B-tree of the memory component;
5: Insert (*geo*, *b_addr*) into B_M ▷ $CT[0].num$++
6: **if** ($CT[0].num \geq CT[0].th$) **then**
7: *Flush Memory Component*;

Algorithm 1 gives the method to insert point data into the memory component located at the top level of the HQ-sLSM tree. After receiving a data point and an address of the block as the input, the geohash value for the the point data(x,y) is computed(32-bit binary number). The newly computed geohash value and the address of the block is the actual data that gets inserted in the tree.

Figure 6a shows the flushing and merging of the first component of level 1(i.e. $C_{1,0}$) in detail. The range of numbers in the figure represents the unit quadrant range. When $C_{1,0}$ having the unit quadrant range 0 to 63 exceeds its threshold, it is first partitioned into four quadrant space groups with the corresponding unit quadrant ranges (i.e., 0-15, 16-31, 32-47, and 48-63). These four partitioned groups are flushed to the level 2 disk. In the level 2 disk, the components with the unit quadrant range of 0-15 and 32-47 already exists while the components with the unit quadrant range of 16-31 and 48-63 do not exist. Thus, among the four partitioned groups flushed to level 2, the two groups with the unit quadrant range of 0-15 and 32-47 are merged with the two existing level 2 disk components with the same unit quadrant range. The other two flushed groups with the unit quadrant range 16-31 and 48-63 become two new level 2 disk components. The process of merging components are shown in Figure 6b.

After flushing and merging processes are finished, an update of the component table for all the affected components are necessary(i.e., $C_{1,0}$, $C_{2,0}$, $C_{2,1}$, $C_{2,2}$, $C_{2,3}$). Since $C_{1,0}$ was flushed to level 2, it is empty with no data indexed. Thus, we update its metadata (i.e., $sf = 0000$, $num = 0$) in the component table. The metadata for the affected level 2 disk components are also done accordingly(*sf*, *C_addr*, and *num*).

While Figure 6 shows the process of flush and merge for a component in the level before MDL, this process works slightly differently for components in the level after MDL. The levels after MDL has no more space decomposition.

Thus, there is no need for the indexes in a component to be partitioned into four quadrant space groups. All the indexes in the component merely has to be flushed to the same quadrant space at the next disk level.

Regardless of the location of the component, we make sure that no component ever exceeds the threshold during the flush and merging process. This is assured by checking if the remaining space of the component corresponding to the quadrant at the next disk level where the flushed index data will be merged is bigger than the size of index data being flushed. The remaining space of a component is calculated by using the *th* and *num* attributes of the component table. We compare the size of the indexes being flushed with the value of $(th - num)$ of the component at the next level. If the latter value is smaller, it implies that there is currently no sufficient space in the component.

In this case, the component at the next level is pre-flushed to make space. This process is done recursively as many times as needed. After making space in the next level, we flush the indexes. The flushed indexes are merged with the pre-existing *B*-tree of the component at the next level.

IV. RANGE QUERY PROCESSING

A. GENERATING QUERY FILTERS

A range query retrieves all the point data in the HQ-sLSM tree whose distance from a query point q is within a given range r. Our range query processing algorithm proposed in this paper for the HQ-sLSM tree first generates a query filter based on a given query range. The query filter is a set of *smallest unit quadrant* numbers whose corresponding quadrants overlap with our query range.

A detailed algorithm for creating the query filter is given in algorithm 2. The *create_query_filter* function recieves an MBR (Minimum Bounding Rectangle) as an input. With the given MBR, the smallest unit quadrant containing the lower left corner point of the MBR and the upper right corner of the MBR is computed respectively. All the smallest unit quadrants with the geohash value inbetween the two values computed above are put into a candidate set. Then a refinement step is performed. The overlapping area between the MBR and the smallest unit quadrants in the candidate set are checked. The geohash values of the overlapping smallest unit quadrants become the query filter.

The query filter gives us the set of specific components subject to access for a query. However, to perform an even better search, we compare the query filter with the spatial filters of the relevant components to filter out accessing unecessary components in the query filter. Figure 7 shows the difference of using a spatial filter together with the query filter in detail.

The shaded area in Figure 7 implies the query range. Because it overlaps with component A, the component will be included the query filter. If the spatial filter is not used as shown in Figure 7a, component A is a subject of access. However, using the spatial filter as shown in Figure 7b, we can see that no data exists in the area where the component and query range overlap. The first two bits of the spatial filter (i.e., 0011) of component A are both 0. Therefore although in the query filter, component A does not need to be accessed while processing this query. The spatial filter and the query filter together helps to reduce unnecessary disk component accesses, making our search very effective.

(a) **Without using spatial filter** (b) **Using spatial filter**

FIGURE 7. Example of using the spatial filter.

Algorithm 2 create_query_filter(M)

INPUT M:: an MBR

OUTPUT Q_filter:: a set of z-order numbers

1: *Set* Q_filter, Q_cand;

2: $Q\_filter \leftarrow \oslash$
3: $Q\_cand \leftarrow \oslash$

4: Compute the smallest unit quadrant L to which the lower left corner point of M belongs;
5: Compute the smallest unit quadrant R to which the upper right corner point of M belongs;
6: Compute the geohash value i for L;
7: Compute the geohash value j for R;
8: $Q\_cand = \{t | i \leq t \leq j\}$

9: **for** (each t in Q_cand) **do** ▷ refinement step
10: Let D be the smallest unit quadrant identified by the geohash value t;
11: **if** (D overlaps M) **then**
12: $Q\_filter \leftarrow Q\_filter \cup t$

13: **return** Q_filter

B. A RANGE QUERY ALGORITHM FOR HQ-sLSM TREE

Figure 8a shows an example of creating a query filter for a given range and utilizing it for a range query. The MDL is set to 2 in this example, thus by dividing the space into 4^{MDL+1} units, the query filter has a range between 0-63. With the given query point and the radius, we obtain the query filter containing 16 smallest unit quadrants(i.e, 2,3,4,...36,37) as shown in Figure 8b.

Using the spatial filter information from the component table given in Figure 8c, it is possible to filter out unnecessary

components from the query filter at each level. In level 1, the spatial filter of the component corresponding to the unit quadrant range of 32-47 is 0000. Thus, the component does not need to be accessed. Refering to the component table, we filter out 3 more components in level 2 as shown in Figure 8d.

Algorithm 3 receives a query point q and a radius r. The output is all the point data included in the query range. The algorithm uses a priority queue to store the query result. A node of the priority queue has two attributes(point, euclidean distance from query point). First, the memory component is searched to obtain points that are inside the query range. Then, an MBR for the circle centered at query point q with a radius of r is created and the MBR is passed to algorithm 2 to generate the query filter as shown in lines 10 and 11 of algorithm 3.

Next, we search the HQ-sLSM tree by going through each row of the component table from the top. One detail to take notice is that the component table is read from the second row as shown in line 13 in algorithm 3. Since the memory component has already been searched, the first row of the component table storing information about the memory component is ignored. From the second row and onwards, a row(component) can be skipped if any of the two following conditions are satisfied. First, if the *num* attribute of a component is 0. This implies that the component is empty. Second, if the area of a component does not overlap at all with the MBR created at the beginning of the algorithm, the component is not an interest for the query.

If a component does not fall into the two conditions mentioned above, the spatial filter and the query filter are compared to determine if the component needs to be accessed. The accessing of a component happens when two conditions are satisfied at the same time. First, a sub-quadrant of a component has a spatial filter value of 1. Second, the sub-quadrant mentioned in the first condition overlaps with the query filter. To check for the second condition, we use the function *in_qfilter* defined in line 31. Recieving a quadrant and the set of the query filter as inputs, the *in_qfilter* function computes the *lb* and *ub* of the quadrant. If a number between the *lb* and *ub* exist in the query filter, it implies that the quadrant overlaps with the query filter and 1 is returned.

Assuming that the two conditions are satisfied, the next step is to access the component and retrieve indexes which fits our query. When accessing a component, we use the function *RQ*. This function traverses the *B*-tree of the component and returns all search key data points e=(x,y) whose geohash value is between two given numbers. We compute the *minkey* and *maxkey* of the sub-quadrant and use it with the function *RQ*. Lines 27 to 29 describe the process of adding data points actually within the query range to the result set. Going through each search key data in the result of *RQ*, we calculate the Euclidean distance between the data point and the query point. If the distance is smaller than the range r, the data point is en-queued to the result set. After searching the last row of the component table, the result set is returned and the algorithm terminates.

Algorithm 3 range_query(q, r)

INPUT q:: A query point (x, y)
r:: Radius
OUTPUT Q:: A set of query results

1: *Global Component Table* CT;
2: *Global Set* Q_filter;
3: *Priority Queue* Q;
4: *Set* R;
5: *int* i; ▷ row number of CT
6: *int* check; ▷ return value of *in_qfilter* function

7: Let B_M be the B-tree of the memory component;
8: $\{p_1, \ldots, p_n\}$ = the set of points in B_M within the distance $Dist_E(r, q)$ to q;
9: $Q = <(p_1, Dist_E(q, p_1)), \ldots, (p_n, Dist_E(q, p_n))>$

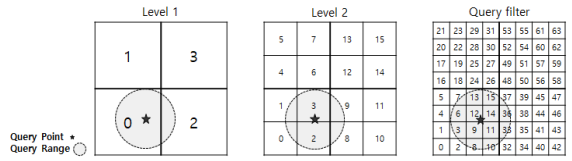
10: Construct a MBR M for the circle centered at q with a radius r;
11: $Q\_filter = create\_query\_filter(M)$;

12: Let N be the last row number of CT;
13: **for** $(i = 1; i \leq N; i++)$ **do**
14: **if** $(CT[i].num == 0)$ **then**
15: continue;
16: Let M' be the quadrant corresponding to CT[i];
17: **if** $(M' \cap M = \emptyset)$ **then**
18: continue;

19: Let $b_0b_1b_2b_3$ be the $CT[i].sf$
20: **for** $(k = 0; k \leq 3; k++)$ **do**
21: Let X be the corresponding quadrant for b_k;
22: $check = in\_qfilter(X, Q\_filter)$;

23: **if** $(b_k == 1$ & $check)$ **then**
24: B is the B-tree accessed by CT[i].C_addr;
25: min=**minkey**(X); max=**maxkey**(X);
26: $R \leftarrow RQ(B, min, max)$;
27: **for** (each search key data point e in R) **do**
28: **if** $(Dist_E(q, e) \leq r)$ **then**
29: En-queue<$(e, Dist_E(q, e))$>;
30: **return** Q

▷ Determines if quadrant is in the Query filter
31: **function** in_qfilter(X, Q_filter)
32: Compute the *lb* and *ub* of the quadrant X;
33: Let $F = \{s | lb \leq s \leq ub\}$
34: **if** $((F \cap Q\_filter) \neq \emptyset)$ **then**
35: **return** 1
36: **else**
37: **return** 0



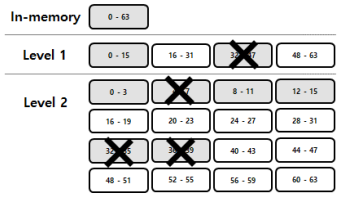
(a) **An example of processing a range query for HQ-sLSM tree**

2	3	6	7	8	9	10	11	12	13	14	15	32	33	36	37

(b) **An example of query filter**

Disk level (dl)	Unit quadrant range		Spatial filter (sf)
	Lower bound (lb)	Upper bound (ub)	
0	0	63	1111
1	0	15	1111
1	16	31	0000
1	32	47	0000
1	48	63	0010
2	0	3	0001
2	4	7	1000
2	8	11	1100
2	12	15	0110
...	...	...	...
2	32	35	0000
2	36	39	0000
...	...	...	...

(c) **Part of a component table**



(d) **Filtering unnecessary components using the spatial filters**

FIGURE 8. **Example of query filters for query processing.**

V. kNN QUERY PROCESSING

The basic idea of the kNN query processing algorithm is to quickly find the k-nearest neighbors to the query point in the HQ-sLSM tree. The kNN candidates and the query range are updated at the end of searching a level. We propose two types of algorithms to process the kNN query for the HQ-sLSM tree in this paper. The common idea of the two methods is that we narrow the search range as we traverse the levels of the HQ-sLSM tree.

A. A TOP-DOWN METHOD

The first is a Top-down method where we start by searching the memory component and then move to the higher level of disk components. Algorithm 4 is a kNN query processing algorithm for the corresponding approach.

Algorithm 4 has a similar structure to the range query algorithm proposed in section IV. The main difference is that the query range changes for each level of the HQ-sLSM tree. In other words, we conduct a range query at each level with a range that gets smaller as the level increases. The first step in algorithm 4 is to obtain the k-nearest neighbors and create a candidate set for the query point in the memory component. The initial distance is set to a large value in order to search the entire memory space. The candidate set is stored in a priority queue, with each node having two attributes(point, euclidean distance from query point). The queue is always kept in a descending order by the second attribute. Thus meaning that the kth closest point is at the front of the queue and the closest point is at the end of the queue. The reason for this order is to de-queue the kth point efficiently when a closer neighbor has been found.

Algorithm 4 TopDown_kNN (q, k)

INPUT q:: a query point (x, y)
k:: a number k
OUTPUT Q:: a set of query results

1: *Global Component Table* CT;
2: *Global Set* Q_filter;
3: *Global Priority Queue* Q; ▷ kept in descending order
4: *int* lvl = -1; ▷ current level
5: *int* check; ▷ return value of *in_qfilter* function
6: *int* r $\leftarrow \infty$; ▷ distance

7: Let B_M be the B-tree of the memory component;
8: $\{p_1, \ldots, p_k\}$ = the k nearest points in B_M sorted in descending order of their $Dist_E$ to q;
($\{p_m, p_{m+1} \ldots, p_k\}$ may be $\oslash$ if B contains $< k$ points)
9: $Q = <(p_1, Dist_E(q, p_1)), \ldots, (p_j, Dist_E(q, p_j))> (m \leq j \leq k)$

10: **if** ($|Q| == k$) **then**
11: $r = Dist_E(q, p_1)$;

12: Let N be the last row number of CT
13: **for** ($i = 1; i \leq N; i++$) **do**
14: **if** ($CT[i].dl > lvl$) **then** ▷ new level
15: Construct a MBR M for the circle centered at q with a radius r;
16: $Q\_filter \leftarrow \oslash$;
17: $Q\_filter = create\_query\_filter(M)$;
18: $lvl = lvl + 1$;
19: **if** ($CT[i].num == 0$) **then**
20: continue;
21: Let M' be the quadrant corresponding to CT[i];
22: **if** ($M' \cap M = \emptyset$) **then**
23: continue;
24: Q = $kNN\_comp\_access(i)$;
25: **return** Q

If k candidates are obtained from memory, the distance of the query range r is updated to the first element in the queue. If not, the distance of the query range does not change as we move on to visit level 1. With the current distance, a query filter is created for disk level 1. A search in a level follows the same process described in section IV. A search of the HQ-sLSM tree is made by visiting each row of the component table. As line 13 of algorithm 4 shows, the component table is scanned from the second row because the first row holds information about the memory component which was already searched earlier. We first check to see if the component can be skipped by looking at the *num* attribute and comparing the quadrant of the component with the MBR created earlier.

A component gets accessed when a sub-quadrant of a component has a spatial filter value of 1 and the corresponding sub-quadrant overlaps with the query filter. This process is executed by the function *knn_comp_access*

Algorithm 5 kNN_comp_access (i)

INPUT i:: row number of component table

OUTPUT Q:: a set of query results

1: *Global Component Table* CT;
2: *Global Set* Q_filter;
3: *Global Priority Queue* Q; ▷ kept in descending order
4: *Set* R;

5: Let $b_0b_1b_2b_3$ be the $CT[i].sf$
6: **for** ($k = 0; k \leq 3; k++$) **do**
7: Let X be the corresponding quadrant for b_k;
8: $check = in\_qfilter(X, Q\_filter)$;

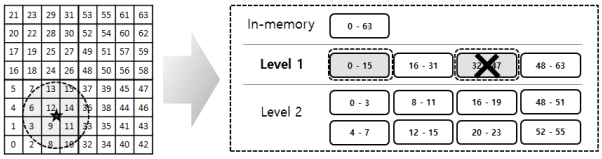
9: **if** ($b_k == 1$ & $check$) **then**
10: B is the B-tree accessed by CT[i].C_addr;
11: min=**minkey**(X); max=**maxkey**(X);
12: $R \leftarrow RQ(B, min, max)$;

13: **for** (each search key data point e in R) **do**
14: **if** ($Dist_E(q, e) \leq r$) **then**
15: **if** ($|Q| < k$) **then**
16: En-queue<(e, $Dist_E(q, e)$)>;
17: **else**
18: De-queue <(p_1, $Dist_E(q, p_1)$)>;
19: En-queue<(e, $Dist_E(q, e)$)>;
20: $r = Dist_E(q, p_1)$;
21: **return** Q

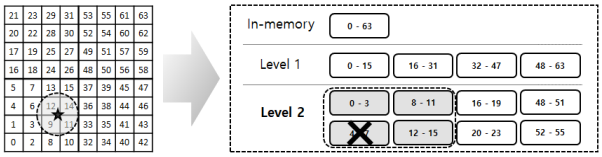
given in algorithm 5. The structure of algorithm 5 has a lot in common with the middle part of the range query algorithm. The same functions *in_qfilter* and *RQ* given in algorithm 3 are also used in algorithm 5. Going through each search key data in the result of *RQ*, we calculate the distance between the data point and the query point.

The difference between the two algorithms appear after function *RQ* has been processed. If there are k entries in the candidate set, and the computed distance of search key data point is smaller than the distance to the first element in the queue, the first element(current kth neighbor) is de-queued and the new search key data gets en-queued. Once the candidate set is sorted, the distance r is updated to the new first element in the queue. However, if there are less than k entries in the candidiate set, search key data is first en-queued until the candidate set contains k entries.

After a search of a level is completed, the query filter gets initialized. As described in line 14 of algorithm 4, the beginning of a new level is determined by checking the *dl* attribute of the component table. With the current distance r, a new query filter for new level is made. The distance for making the query filter will shrink at each level, which means less components will be a subject of access as we traverse the levels of the HQ-sLSM tree.



(a) **Level 1 Component Search**



(b) **Level 2 Component Search**

FIGURE 9. Example of top-down kNN query processing.

Figure 9 shows an example of a Top-down kNN query processing algorithm. Assume that the query range shown in Figure 8a and the query filter corresponding to Figure 8b are created after searching the memory component and obtain a candidate set of k-nearest neighbors. By the comparing the generated query filters with the given range and the spatial filters shown in Figure 8c, only the component corresponding to the unit quadrant range of 0-15 are searched in level 1. When the candidate set is updated after searching level 1, the distance for the query range is changed to the furthest data point entry in the set. As Figure 8b shows, the range has been narrowed down as we search level 2.

The rest of the process is carried out in a similar manner. Using the new distance range, we create a new query filter for level 2. By comparing it with the spatial filters we retrieve 3 components in level 2. The top-down algorithm terminates when we reach the last level of the HQ-sLSM tree, returning the set of query results.

B. A BOTTOM-UP METHOD

The Bottom-up algorithm for processing a kNN query is shown in algorithm 6. This method traverses the HQ-sLSM tree from the last level to the top. For this algorithm, the last level of the HQ-sLSM tree is defined as *MAX_LEVEL*. The first step of this algorithm is to visit the last level of the HQ-sLSM tree and access the component which corresponds to the quadrant where the query point is located in. From this component, we obtain k-nearest neighbors for the query point and form a candidate set. The candidate set is stored in the priority queue having the same structure as the top-down method.

If the k candidates are not obtained from the component at the last disk level, we check the spatially adjacent components in the last disk level and obtain the lacking number of candidates. k candidates must be obtained from the last level of the HQ-sLSM tree before moving on to another level. After creating the candidate set, the distance of the query range r is updated to the first element in the queue.

Before we start traversing the tree, a range query needs to be conducted with the distance r at the last disk level

Algorithm 6 BottomUp_kNN (q, k)

INPUT q:: a query point (x, y)
k:: a number k
OUTPUT Q:: a set of query results

1: *Global Component Table* CT;
2: *Global Set* Q_filter;
3: *Global Priority Queue* Q; ▷ kept in descending order
4: *int* i; ▷ row number of CT
5: *int* lvl=MAX_LEVEL + 1; ▷ current level
6: *int* check; ▷ return value of *in_qfilter* function
7: *int* r ← ∞; ▷ distance

8: Find the largest row number a of CT whose disk component contains the point q;
9: Let B be the B-tree accessed by CT[a].C_addr;
10: $\{p_1, \ldots, p_k\}$ = the k nearest points in B sorted in descending order of their $Dist_E$ to q;
($\{p_m, p_{m+1} \ldots, p_k\}$ may be ⊘ if B contains $< k$ points)
11: Q = <(p_1, $Dist_E(q, p_1)$),…,(p_j, $Dist_E(q, p_j)$)> ($m \leq j \leq k$)

12: **if** ($|Q| < k$) **then**
13: Obtain the remaining top ($k - |B|$) points into Q from B-trees spatially close to CT[a] at disk level CT[a].dl;
14: $r = Dist_E(q, p_1)$;
15: With the distance r, perform a range query at at disk level CT[a].dl to validate that Q holds the top-k nearest indexes;
16: If Q changes, $r = Dist_E(q, p_1)$;

17: Let N be the last row number of CT
18: **for** ($i = N - 4^{MDL}$; $i \geq 1$; $i - -$) **do**
19: **if** ($CT[i].dl < lvl$) **then** ▷ new level
20: Construct a MBR M for the circle centered at q with a radius r;
21: $Q\_filter \leftarrow \oslash$
22: $Q\_filter = create\_query\_filter(M)$;
23: $lvl = lvl - 1$;
24: **if** ($CT[i].num == 0$) **then**
25: continue;
26: Let M' be the quadrant corresponding to CT[i];
27: **if** ($M' \cap M = \emptyset$) **then**
28: continue;
29: Q = $kNN\_comp\_access(i)$;

30: Let B_M be the B-tree of the memory component;
31: **for** (each search key data point e in B_M) **do**
32: **if** ($Dist_E(q, e) \leq r$) **then**
33: De-queue <(p_1, $Dist_E(q, p_1)$)>;
34: En-queue<(e, $Dist_E(q, e)$)>;
35: $r = Dist_E(q, p_1)$;
36: **return** Q

as displayed in line 15 of algorithm 6. This is to assure that the obtained candidate set actually contains the top-k nearest indexes. There may be a case where the query point is located very near a boundary of a component, and the nearest neighboring indexes are actually located in the spatially close components rather than the disk component that contains the point q. If a change in the candidate set occurs, the query range r is updated again to the first element in the queue. After this step, the HQ-sLSM tree can be traversed upwards with the candidate set Q. Since the last level has already been searched with the process above, the HQ-sLSM tree is traversed from ($MAX\_LEVEL - 1$) level to the top after creating a query filter with the current distance. Line 18 of algorithm 6 shows that the component table is searched for levels ($MAX\_LEVEL - 1$) to 1.

The rest of the process is the same as the Top-down method. The query filter is initialized and re-created at a new level, and the accessing of a component is executed by the function *knn_comp_access* given in algorithm 5. The transition of levels is determined by the *dl* attribute of the component table as described in line 19 of algorithm 6. After searching level 1, the memory component of the HQ-sLSM tree is searched before the termination of the algorithm.

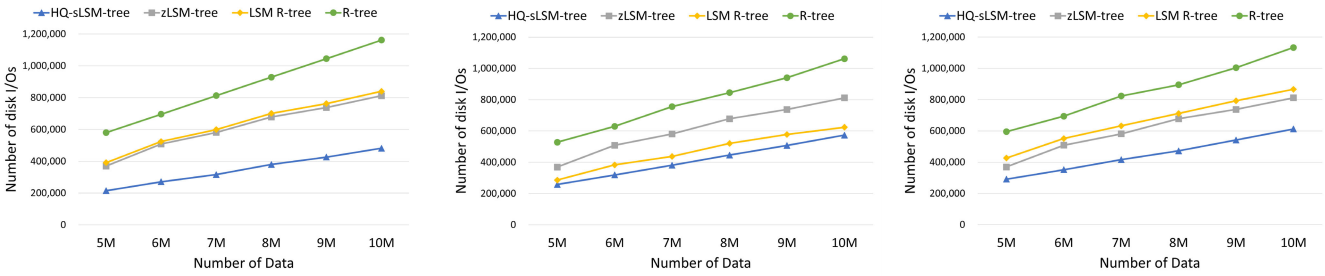
By using the bottom-up method, we start the search of the tree with a very small range compared to the top-down method. Therefore as we traverse the tree, less components become a subject of access resulting in less disk I/Os. Given a query with the same number of k, the bottom-up method tends to perform better than the top-down method. A more detailed comparison of the two kNN query processing algorithm is analyzed in section VI-F.

VI. PERFORMANCE ANALYSIS

A. EXPERIMENTAL DATA

The proposed techniques for the HQ-sLSM tree were experimented and compared with the existing LSM-tree, and the R-tree which is primarily used for indexing data in spatial database systems. Furthermore, comparisons with the LSM R-tree was made as it showed the best performance amongst other expanded LSM structures such as the DHB-tree, DHVB-tree, SHB-tree and SIF [17]. To compare the existing LSM-tree with the HQ-sLSM tree, we linearized the coordinate information of the point (x,y) using a 32-bit geohash value and indexed it using the LSM- tree. We name this tree as the zLSM-tree. For the zLSM-tree used in the following experiments, the size of the disk component at each level was increased to 2 times the size of the disk component at the previous disk level.

Two synthetically generated datasets and one real dataset covering the territory of South Korea were used for the experiments in this paper. All data points were either generated or sampled within the geographic location. The first synthetic dataset contains uniformly distributed point data, and the second synthetic dataset contains point data following a gaussian distribution. For the real dataset, we used the real GPS point data provided by Open-StreetMap [35]. Up to 10 million data points within the territory of South Korea was sampled to create the dataset. A blockchain storing the geospatial point data from the datasets were created. The number of disk I/Os was measured for the HQ-sLSM tree, the R-tree, the zLSM-tree, and the LSM R-tree when indexing geospatial point data from the blockchain. The



(a) **Synthetic-Uniform distribution** (b) **Synthetic-Gaussian distribution** (c) **OSM Real Dataset**

FIGURE 10. **Performance comparisons for the index construction with an increasing size of blockchain data.**

disk I/O was measured blockwise. A movement of a block from memory to disk was counted as 1 disk I/O, and a disk to disk movement was counted as 2 disk I/Os.

Our experiments were performed on a 64-bit Linux operating system with 8 processors. Each processor was a Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz model, having a spec of 128GB RAM and 1TB SSD.

B. THEORETICAL ANALYSIS

A disk component is made up of multiple blocks, which are referred to as disk blocks. Any actions(read or write) that involve accessing a disk component is done in disk block units. If the total number of data entries is N and B entries fit into a disk block, the cost for inserting the data in the zLSM-tree is $O(1/B \times log_T(N/B))$ I/Os where T represents the ratio of the disk component size between adjacent levels. In the zLSM-tree, each entry gets copied $O(log_T(N/B))$ times. In order to copy an entry, a block has to be accessed resulting in 1 disk I/O because all operations are processed blockwise. Therefore, the cost of copying one entry can be thought of as $O(1/B)$. The multiplication of the two results gives the insertion cost of the zLSM-tree.

In the worst case where an entire level is flushed, the HQ-sLSM tree approximates the zLSM-tree. Due to the different ratio of disk component size at levels, the HQ-sLSM tree can be viewed as a mix of two zLSM-trees with a different value T. Thus to calculate the disk I/O cost of the HQ-sLSM tree for insertion and searching a data point, we analyze the tree in two parts.

First, we calculate the disk I/O cost for insertion. The space is divided into four until the MDL, which increases the total disk component size of a level by a multiple of 4. In the worst case, this approximates a zLSM-tree with the value $T = 4$. Naming this part of the HQ-sLSM tree $t1$, its disk I/O cost becomes $O(1/B \times log_4(N/B))$. The threshold is doubled for components after the MDL level which increases the total disk component size of a level by a multiple of 2. Again in the worst case, this approximates a zLSM-tree with the value $T = 2$. By naming this part of the tree $t2$, we can represent the whole HQ-sLSM tree as $t1 + t2$. The disk I/O cost for $t2$ becomes $O(1/B \times log_2(N/B))$.

The lower bound disk I/O cost of insertion for the HQ-sLSM tree becomes $O(1/B \times log_4(N/B))$ when only the $t1$ part exists in the tree. On the contrary, the upper bound disk I/O cost of insertion becomes $O(1/B \times log_2(N/B))$ when only the $t2$ part exists in the tree. To summarize, if the HQ-sLSM tree consists of part $t1$ and $t2$, its disk I/O cost is less than $O(1/B \times log_2(N/B))$ and greater than $O(1/B \times log_4(N/B))$. In general, the HQ-sLSM tree tends to have a better disk I/O cost than the worst case calculated above due to the fact that a level is not very often flushed entirely in the HQ-sLSM tree.

Second, the disk I/O cost for searching a point data in the zLSM-tree is $O((log_T(N/B))^2)$ where T represents the ratio of the disk component size between adjacent levels. The cost above is obtained by multiplying the cost of traversing the tree from the top $O(log_T(N/B))$, and the cost of using a binary search to search a point in a component $O(log_T(N/B))$.

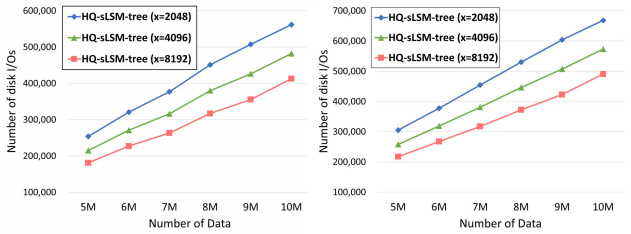
In the case of the HQ-sLSM tree, we can narrow down the location of the point being searched. Using the component table allows us to easily identify the disk component that needs to be searched at each level. Thus, although the number of components increase, only one component needs to be searched per level unlike the zLSM-tree. This allows for the traverse cost of the HQ-sLSM tree to be ignored. As a result, the cost for searching a point data in the HQ-sLSM tree is the binary search cost of a component $O(log_T(N/B))$.

For $t1$ and $t2$ parts of the HQ-sLSM tree, the disk I/O cost for searching a data point becomes $O(log_4(N/B))$ and $O(log_2(N/B))$ respectively. Combining part $t1$ and $t2$, the disk I/O cost for searching a point data in the HQ-sLSM tree can said to be less than $O(log_2(N/B))$ and greater than $O(log_4(N/B))$.

C. INSERTION PERFORMANCE

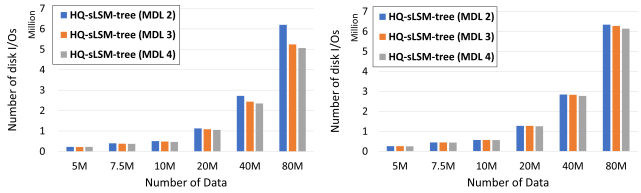
For this experiment, the two synthetic datasets and the real dataset was used with their size each increasing from 5 million to 10 million points. The MDL of the HQ-sLSM tree was set to 3, and the threshold size of the component was fixed to 4096. Figure 10 shows the results for the data insertion performance of the HQ-sLSM tree, zLSM-tree, LSM R-tree, and the R-tree using three datasets. The number of data in each dataset started at 5 million and increased until 10 million for this experiment.

Regardless of the dataset, the disk I/Os for all the baseline index trees increased as a larger number of data was inserted. The R-tree showed the worse performance of all trees while



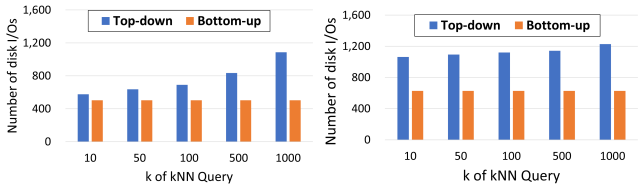
(a) **Uniform distribution** (b) **Gaussian distribution**

FIGURE 11. Performance comparisons by varying component threshold in the HQ-sLSM tree.



(a) **Uniform distribution** (b) **Gaussian distribution**

FIGURE 12. Performance comparisons of MDL change in the HQ-sLSM tree.



(a) **Uniform distribution** (b) **Gaussian distribution**

FIGURE 13. Performance Comparison of Top-Down and Bottom-up *k*NN query processing methods.

the HQ-sLSM tree showed the best performance. A lot of random disk I/Os occur when inserting data in the R-tree, which gives a poor result for data insertion. Increasing the capacity of each level by the decomposition of space before the MDL, and increasing the threshold after the MDL reduces disk I/Os for data insertion in the HQ-sLSM tree. Furthermore, dividing the space into quadrants prevent the flushing of an entire level, which also contributes to the lesser disk I/Os.

One noticeable fact from Figure 10 was that the disk I/Os of the zLSM- tree for all datasets were the same. In the zLSM-tree, the entire space is indexed by one *B*-tree at each level. Thus, the number of I/Os for the same number of data will always be the same regardless of the distribution of data points. In other words, the data distribution does not affect the number of disk I/Os in the zLSM-tree.

D. INSERTION PERFORMANCE - VARYING THRESHOLD

For this experiment, the two synthetic datasets were used with their size each increasing from 5 million to 10 million points. The MDL of the HQ-sLSM tree was fixed to 3. Figure 11 shows the effect of varying the threshold value x of a component in the HQ-sLSM tree. Three threshold values were tested for each size of the two datasets. Starting with a threshold value of 2048, the value was doubled twice. The order of the three threshold values for the two distributions showed no change while the size of the dataset increased. The threshold value of 2048 always performed worse and the threshold value of 8192 performed the best.

A larger threshold enables for more indexes to be stored in a level of the HQ-sLSM tree. This lead to a fewer flushing and merging of components which eventually resulted in a smaller number of disk I/Os. Again, the Gaussian dataset resulted with a little higher number of disk I/Os due to the fact that the data-dense area requried a deeper level in the HQ-sLSM tree.

E. INSERTION PERFORMANCE - VARYING MDL

For this experiment, the two synthetic datasets were used with their size each increasing from 5 million to 80 million points. The threshold size of the component was fixed to 4096. Figure 12 shows the results of measuring the number of disk I/O by varying the MDL in the HQ-sLSM tree. The size of the synthetic dataset was increased to a very large size for this experiment because a distinct pattern could not be found with a small dataset size. As the size of the dataset increased drastically, the HQ-sLSM tree with a higher MDL turned out to perform better for both distributions.

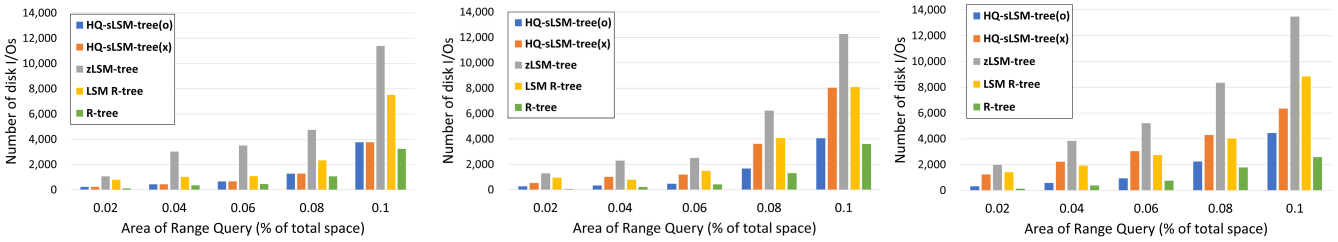
The reason for the marginal difference in the disk I/Os with the small number of data was that the three HQ-sLSM trees were at different stages of indexing data. Some were still decomposing space into quadrants while others were finished with the decomposition and were doubling threshold values. A mix of the two made it unclear to determine the best performing HQ-sLSM tree.

To index a large number of data, all HQ-sLSM trees need a fairly deep level, all well past their MDL. Eventually, a HQ-sLSM tree with a higher MDL creates much more components in the tree, which allows for more indexes to be stored in a level and ends up reducing the number of disk I/Os. However, the increase of MDL did not show a significant difference in the decrease of disk I/Os after MDL 3. Moreover, there exists a tradeoff between the maintenance overhead of the components. This is to be discussed in section VI-I. The Gaussian distribution required more flushes in the data-dense areas which resulted in a higher number of disk I/Os compared to the uniform distribution.

F. TOP-DOWN AND BOTTOM-UP PERFORMANCE COMPARISON FOR THE kNN QUERY

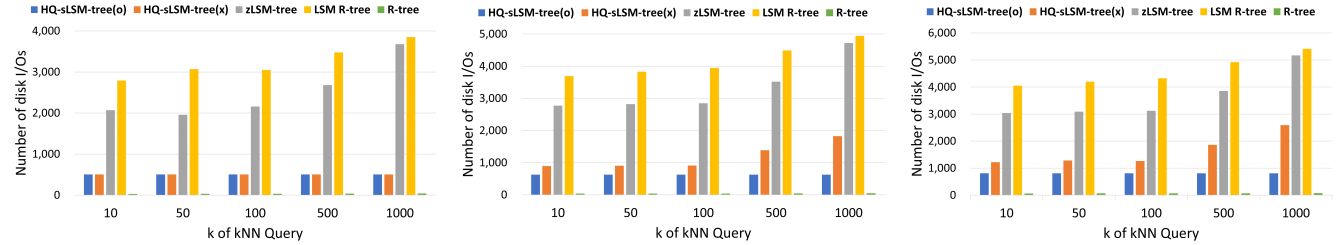
For this experiment, the two synthetic datasets with a size of 5 million points were used. The MDL of the HQ-sLSM tree was set to 3, and the threshold of a component was fixed to 4096.

Figure 13 shows the results of measuring disk I/O while processing a *k*NN query in the HQ-sLSM tree with the



(a) **Uniform distribution** (b) **Gaussian distribution** (c) **OSM Real Dataset**

FIGURE 14. **The effect of the radius of the range query on disk I/O with 3 datasets.**



(a) **Uniform distribution** (b) **Gaussian distribution** (c) **OSM Real Dataset**

FIGURE 15. **The effect of the *k* of the *k*NN query on disk I/O with 3 datasets.**

number of neighbors ranging from 10 to 1000. The disk I/O was measured for the two different algorithms explained in section V.

The Bottom-up method performed better compared to the Top-down method in the both distributions. In the top-down method, a candidate set is made from the memory first which leads to a relatively large distance(query range). However, in the Bottom-up method, the candidate set is obtained from the component at the last level where the query point is located. Using a HQ-sLSM tree with MDL 3, a tree having an average depth of level 8 was created to index 5 million point data. A component at the last level has a threshold size of about 128,000. Thus, it was very likely to obtain k neighbors in one component of the last level. This implies that the query range would be created inside a component, and from the last level to the MDL, only one component would be accessed per level.

This caused the big reduce of disk I/O with the bottom-up method. The components in the levels after MDL contains much more data compared to the lower levels in the HQ-sLSM tree. Therefore, filtering out one or two components create a meaningful diffference in the number of disk I/Os. To summarize, the difference in the initial query range of the two methods produced the difference in the number of disk I/Os.

As the number of k increases, the candidate set also grew larger creating a bigger query range. Thus to find more nearest neighbors, more components became a subject of access at each level, increasing the total number of disk I/Os while processing the query. Furthermore, the Gaussian distribution showed more disk I/Os because more data had to be searched in the data-dense areas.

G. RANGE QUERY AND kNN QUERY PERFORMANCE

For this experiment, two synthetic datasets and the real dataset was used, each with a size of 5 million points. The MDL of the HQ-sLSM tree was set to 3, and the threshold of the component was fixed to 4096. Figure 14 and 15 shows the experimental results of comparing the HQ-sLSM tree, the zLSM-tree, the LSM R-tree, and the R-tree for the range query, and the kNN query for the three datasets. In Figure 14 and 15, there are two versions of the HQ-sLSM tree, marked with (o) and (x) respectively. The HQ-sLSM tree(o) refers to a HQ-sLSM tree using the spatial filters described in section III-B when processing queries. The HQ-sLSM tree(x) is a HQ-sLSM tree which does not use the spatial filter and process queries only based on the query filter.

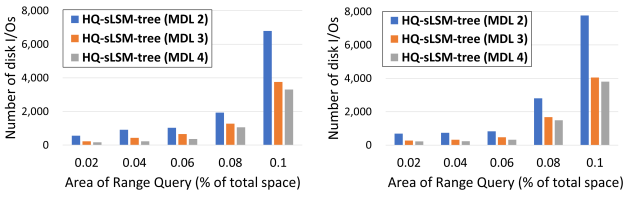
The area of the query range was set to 2 to 10 percent of the total data space for the range query, and the number of k was set to 10 to 1000 for the kNN query. As described in section III-C, comparing query filters with the spatial filters stored in the component table allows us to avoid accessing unnecessary components while processing queries. For both the range query and the kNN query, we were able to see that the HQ-sLSM tree(o) performed the second best, slightly worse than the R-tree. However, for queries based on uniformly distrubuted data (i.e., Figure 14a, Figure 15a) the effect of the spatial filter was very marginal. Due to the uniform distribution, the all the spatial filters were very likely to have a value of 1.

Thus regardless of the location of the query filter, all components overlapping with the query filter was likely to be a subject of access. With or without the spatial filter, the number of component access was similar. The difference

between the HQ-sLSM tree(o) and HQ-sLSM tree(x) in the Gaussian and real dataset confirmed that the spatial filter we proposed worked significantly. For the HQ-sLSM tree(o) in Figure 15, we used the result of the better performing Bottom-up method for processing the kNN query.

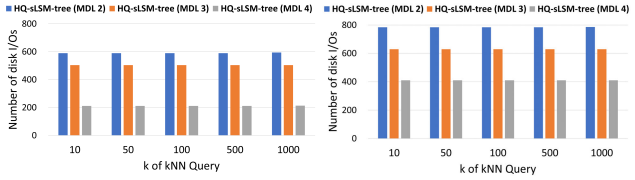
H. RANGE QUERY AND kNN QUERY PERFORMANCE - VARYING MDL

For this experiment, the two synthetic datasets with a size of 5 million points were used. The MDL of the HQ-sLSM tree was set to 3, and the threshold of a component was fixed to 4096. Figure 16 and Figure 17 show effect the MDL in the HQ-sLSM tree has on the range query and kNN query. This results show that a higher MDL results in less disk I/Os for range and kNN queries.



(a) **Uniform distribution** (b) **Gaussian distribution**

FIGURE 16. Range query in the HQ-sLSM tree - varying MDL.



(a) **Uniform distribution** (b) **Gaussian distribution**

FIGURE 17. kNN query in the HQ-sLSM tree - varying MDL.

More space decomposition with the higher MDL improves the filtering effect when using the the query filter and the spatial filter together. Going back to the results from section VI-E, a higher MDL in the HQ-sLSM tree tends to perform better in terms of insertion and processing range and kNN queries. By these two experiments, we can confirm that a certain high MDL is needed for a well perfoming HQ-sLSM tree in general. We used the results of the better performing bottom-up method in Figure 17.

I. MAINTENANCE OVERHEAD OF DISK COMPONENTS

For this analysis, a HQ-sLSM tree having 15 levels was assumed to be created. The number of components generated in the tree was calculated while varying the MDL of the tree.

Following the results of the previous experiments, it seemed logical to increase the MDL. However as the MDL increases, the total number of created components increase radically after a certain MDL as shown in Figure 18. The overhead of opening and the maintenance of disk components are the factors to consider when increasing components.

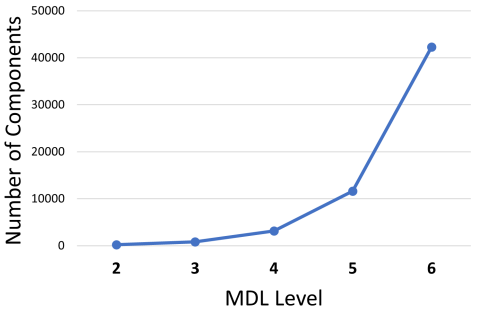


FIGURE 18. Number of components created for different MDL.

Along with this, the size of the component table holding metadata for all components also increase, which causes a tradeoff with performance results.

While a higher MDL might seem to perform better for insertion and processing range and kNN queries, the underlying problem of maintenence overhead cannot be ignored. Comparing the marginal difference of improved performance for insertion and processing queries as shown in Figures 12 and 16 to the increase of components, it is reasonable to select a value of 3 or 4 for the MDL in the HQ-sLSM tree.

J. DISK SPACE FOR THE HQ-sLSM TREE

For this analysis, a HQ-sLSM tree having the parameters of MDL value 3 and component threshold 4096 was used. The uniform and gaussian synthetic datasets each ranging from 1M to 10M were inserted into the HQ-sLSM tree. The total disk space used by the disk components created in the HQ-sLSM tree from the insertion was measured each time and the average disk space from the two datasets is displayed in Figure 19. The disk space for the R-tree and the LSM R-tree was measured as well when the same datasets were inserted.

The HQ-sLSM tree requires about 120MB of disk space for a 10M dataset, whereas the R-tree and the LSM R-tree requires twice as much. The increase of the disk space as the size of the data gets larger is smaller for the HQ-sLSM tree. Using Geohash to store the coordinate point data contributed to shrinking the size of the spatial data when the HQ-sLSM tree was used.

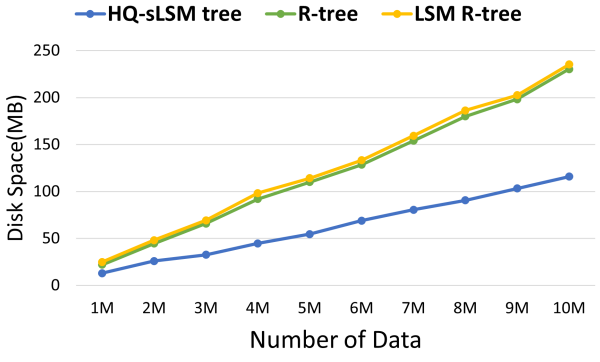


FIGURE 19. Disk space analysis for the HQ-sLSM tree, R-tree and LSM R-tree.

Likewise, the zLSM-tree also uses the geohash values of the point data as keys. Although not shown on the graph below, the space perfomance of the zLSM-tree will not be very different to the HQ-sLSM tree in terms of disk space. We successfully concluded that our proposed HQ-sLSM tree performs much better in insertion and is more space-efficient in the process compared to the R-tree and the LSM R-tree.

VII. CONCLUSION

In this paper, we proposed a hierarchical quadrant spatial LSM tree (i.e., HQ-sLSM tree) which reduces disk I/O costs when inserting geospatial data from a blockchain environment. Disk components in the HQ-sLSM tree have a hierarchical structure by decomposing space into four quadrants as they go down the level. The experiments we conducted confirmed that the HQ-sLSM tree performed the best amongst other existing baseline index trees (i.e., the R-tree, the zLSM-tree, and the LSM R-tree) for the insertion of data, and showed a superior performance over the zLSM-tree and the LSM R-tree when processing range queries and kNN queries. Using the spatial filter and the query filter together is the key to minimizing disk I/Os by avoiding unnecessary component access. Although the HQ-sLSM tree falls short in processing range and kNN queries compared to the R-tree, the efficiency of inserting data is the more important factor in a write intensive blockchain environment. Thus, the overall performace of our HQ-sLSM tree provides sufficient competitiveness against existing methods.

Furthermore, we identified the effects of changing the thresholds of the component and changing MDL have on the number of disk I/Os for inserting data and processing range and kNN queries in the HQ-sLSM tree. A HQ-sLSM tree with a higher MDL eventually showed better results in both insertion and processing queries. However, the fact that the difference in disk I/Os become very marginal for insertion of data at a certain MDL level, and the increasing maintenance overhead that comes along with a higher MDL gives a tradeoff between a higher MDL and the maintenance overhead of the tree. Currently, we are developing an HQ-sLSM tree to index line and polygon data stored in the blockchain.

REFERENCES

[1] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage management in AsterixDB," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 841–852, Jun. 2014.

[2] *Apache Apache Hbase: The Hadoop Database, a Distributed, Scalable, Big Data Store*, Apache Softw. Found., Wilmington, DE, USA, 2023.

[3] M. Bal, "Securing property rights in India through distributed ledger technology," Observer Res. Found., New Delhi, India, Tech. Rep., Jan. 2017.

[4] M. N. Kamel Boulos, J. T. Wilson, and K. A. Clauson, "Geospatial blockchain: Promises, challenges, and scenarios in health and healthcare," *Int. J. Health Geographics*, vol. 17, no. 1, Dec. 2018.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[6] K. Dorling, J. Heinrichs, G. G. Messier, and S. Magierowski, "Vehicle routing problems for drone delivery," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 1, pp. 70–85, Jan. 2017.

[7] T. M. Fernández-Caramés and P. Fraga-Lamas, "A review on the use of blockchain for the Internet of Things," *IEEE Access*, vol. 6, pp. 32979–33001, 2018.

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

[9] *FOAM Location*, Brooklyn, New York, NY, USA, 2018.

[10] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, Jun. 1998.

[11] S. Ghemawat and J. Dean. *LevelDB*. Accessed: Oct. 26, 2023. [Online]. Available: http://code.google.com/p/leveldb

[12] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984.

[13] J. He and H. Chen, "An LSM-tree index for spatial data," *Algorithms*, vol. 15, no. 4, p. 113, Mar. 2022.

[14] *IBM Supply Chain Intelligence Suite: Food Trust*, IBM, Armonk, NY, USA, 2022.

[15] H. V. Jagadish, "Linear clustering of objects with multiple attributes," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, May 1990, pp. 332–342.

[16] Y. S. Kim, "Transactional and spatial query processing in the big data era," Ph.D. thesis, UC Irvine Electron., Univ. California, Irvine, CA, USA, 2016.

[17] Y.-S. Kim, T. Kim, M. J. Carey, and C. Li, "A comparative study of log-structured merge-tree-based spatial indexes for big data," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 147–150.

[18] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[19] J. K. Lawder, "The application of space-filling curves to the storage and retrieval of multi-dimensional data," Doctoral dissertation, Birkbeck, Univ. London, U.K., 2000.

[20] H. Leinonen, *Decentralised Blockchained and Centralised Real-Time Payment Ledgers: Development Trends and Basic Requirements*. London, U.K.: Palgrave Macmillan, 2016, pp. 236–261.

[21] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-infused streams: Enabling authentication of sliding window queries on streams," in *Proc. Very Large Data Bases Conf.*, 2007, pp. 147–158.

[22] J. Liu, H. Li, Y. Gao, H. Yu, and D. Jiang, "A geohash-based index for spatial data management in distributed memory," in *Proc. 22nd Int. Conf. Geoinformatics*, Jun. 2014, pp. 1–4.

[23] L. I. U. Zi-Hao, H. U. Hui-Qi, X. U. Rui, and Z. H. O. U. Xuan, "Implementation of LevelDB-based secondary index on two-dimensional data," *J. East China Normal Univ.*, vol. 2019, no. 5, p. 159, 2019.

[24] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, Jul. 2019.

[25] Q. Mao, S. Jacobs, W. Amjad, V. Hristidis, V. J. Tsotras, and N. E. Young, "Comparison and evaluation of state-of-the-art LSM merge policies," *VLDB J.*, vol. 30, no. 3, pp. 361–378, Feb. 2021.

[26] Q. Mao, M. A. Qader, and V. Hristidis, "Comprehensive comparison of LSM architectures for spatial data," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2020, pp. 455–460.

[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[28] Q. Qu, I. Nurgaliev, M. Muzammal, C. S. Jensen, and J. Fan, "On spatio-temporal blockchain query processing," *Future Gener. Comput. Syst.*, vol. 98, pp. 208–218, Sep. 2019.

[29] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Gener. Comput. Syst.*, vol. 88, pp. 173–190, Nov. 2018.

[30] H. Samet, *Foundations of Multidimensional and Metric Data Structures* (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). San Francisco, CA, USA: Morgan Kaufmann, 2005.

[31] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, May 2012, pp. 217–228.

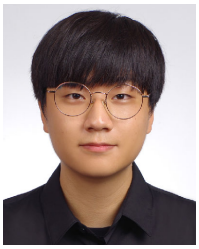
[32] M. Seo, T. Kwon, and S. Jung, "Spatial LSM tree for indexing blockchain-based geospatial point data," *J. KIISE*, vol. 49, no. 10, pp. 898–905, Oct. 2022.

[33] J. Shin, J. Wang, and W. G. Aref, "The LSM RUM-tree: A log structured merge R-tree for update-intensive spatial workloads," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, Apr. 2021, pp. 2285–2290.

[34] M. Skrodzki, "The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time," 2019, *arXiv:1903.04936*.

[35] R. Weait, "Openstreetmap's bulk GPS point data," OpenStreetMap Found., Cambridge, U.K., Tech. Rep., 2014.

[36] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.
[37] R. Xu, Z. Liu, H. Hu, W. Qian, and A. Zhou, "An efficient secondary index for spatial data based on LevelDB," in *Database Systems for Advanced Applications*, Y. Nah, B. Cui, S.-W. Lee, J. X. Yu, Y.-S. Moon, and S. E. Whang, Eds. Cham, Switzerland: Springer, 2020, pp. 750–754.



JUNGHYUN LEE is currently pursuing the bachelor's degree with the Computer Science and Engineering Department, Sogang University. His research interests include spatial databases, blockchain-based geospatial databases, and data analysis.

TAEHYEON KWON received the B.S. degree in computer science from Shinhan University, South Korea, in 2021, and the M.S. degree from the Computer Science and Engineering Department, Sogang University, in 2023. His research interests include spatial databases, blockchain-based geospatial databases, and data analysis.

MINJUN SEO received the B.S. degree in electrical engineering from Hongik University, Seoul, South Korea, in 2018, and the M.S. degree from the Computer Science and Engineering Department, Sogang University, in 2022. His research interests include spatial databases, blockchain-based geospatial databases, and data analysis.

SUNGWON JUNG received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 1988, and the M.S. and Ph.D. degrees in computer science from Michigan State University, East Lansing, MI, USA, in 1990 and 1995, respectively. He is currently a Professor with the Computer Science and Engineering Department, Sogang University. His research interests include data mining, graph neural networks, spatial and temporal databases, and blockchain.