A Fast Hybrid Spatial Index with External Memory Support

Xinyu Su
The University of Melbourne
Melbourne, Australia
suxs3@student.unimelb.edu.au

Jianzhong Qi
The University of Melbourne
Melbourne, Australia
jianzhong.qi@unimelb.edu.au

Egemen Tanin
The University of Melbourne
Melbourne, Australia
etanin@unimelb.edu.au

***Abstract*—Despite claiming to support external memory access, very few learned indices implement their indices on secondary storage. It is challenging to extend learned models to secondary storage while preserving high levels of index performance. In this paper, we propose a *Fast Hybrid Spatial Index with External Memory Support* (FHSIE). Its core idea is to learn a model which can group spatial objects in a top-down manner with few parameters. We use a height-balanced hierarchical structure, which recursively uses simple unsupervised models to group (i.e., cluster) spatial objects. We associate the learned structure with a grid for accurate query processing, and we utilize the relationship among clusters and grid cells to estimate the grid layout and optimize grid performance. Extensive experiments on real and synthetic data sets with more than 200 million points show that FHSIE is highly efficient and precise no matter where it works, i.e., in memory or on a disk.**

***Index Terms*—spatial index, learned index, optimized grid layout, spatial query**

I. Introduction

Spatial indices have become ubiquitous due to an increase in location-based services, such as digital mapping, the next Points of Interest (POIs) recommendation, etc. Traditional approaches like R-trees [1] may incur high storage and query costs when dealing with large data sets.

Recent studies [2]–[4] propose *learned indices* that apply machine learning to construct indices. They formulate an index structure as a *learned function* (i.e., an index model) f which enables mapping the search keys to corresponding physical positions. To construct learned indices for spatial objects (i.e., *learned spatial indices*), studies [4]–[6] map spatial points to one-dimensional values which are then indexed with a learned one-dimensional index. Another study [7] learns the *cumulative distribution function* (CDF) of the object coordinates in each dimension respectively to form a learned spatial index.

Generally, learned spatial indices allocate objects to blocks first, and then they regard the objects' positions (e.g., block IDs) as labels to train learned index models. Due to the inherent inaccuracy of supervised machine learning models, query processing with such learned indices needs to scan the storage addresses around an area of prediction. These indices may miss positive objects for window or kNN queries. To alleviate the inaccuracy problem, RSMI [5] re-labels the objects (i.e., allocates the objects according to model prediction) except the bottom-level, and LISA [4] uses monotonic functions in model learning. The key disadvantages of these indices are that they either have inaccuracy prediction at the bottom level which increases the search range or allocate all objects into the same subset which incurs high scanning costs for query processing.

Further, most learned spatial indices evaluate their query algorithms in memory only. LISA [4] puts data blocks on a hard disk while its learned index models are still in memory. The in-memory storage limits the index models' size and capability which in turn constrains the size of data sets that can be indexed by the index models.

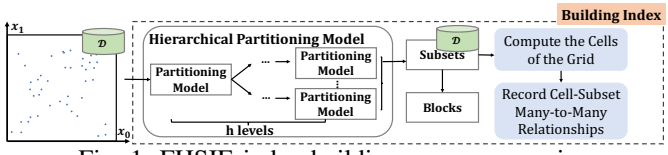


Fig. 1: FHSIE index building process overview

Motivated by these limitations, we propose FHSIE. It associates a machine learning-based structure with a grid to enable accurate and fast point and non-point (window and kNN) queries. FHSIE extends to external memory easily because of fewer models' parameters. As Fig. 1 shows, FHSIE adopts a partitioning model to recursively allocate objects into sub-data sets, which produces a height-balanced structure. Then, it splits each bottom-level subset into (one or more) blocks by the first-dimensional coordinates of the data objects in the subset. We introduce a simple yet efficient method to estimate the layout of the grid by fitting the radii of the bottom-level subsets, which does not require a foreknown query workload as in existing learned spatial indices [7]–[9]. We then build many-to-many relationships between cells and bottom-level subsets for query processing. Note that the grid resolution is determined by the bounding circles of the bottom-level subsets which are learned. Thus, our eventual structure is a hybrid learned and grid-based structure.

Once built, FHSIE processes point queries simply by invoking the models that were learned to allocate objects in subsets in index building. FHSIE adopts a grid-based method for window query processing. For kNN queries, FHSIE combines the point query algorithm and the window query algorithm to enable efficient query processing.

To move FHSIE onto external memory, we decomposed the model into three main components: (1) partitioning models of internal levels, (2) bottom-level partitioning models with block information (e.g., block IDs and block boundary points), and

(3) cells of the grid. Each internal partitioning model is given its own block for storing its meta data (e.g., partition center). Each bottom-level model has a block for storing the meta data and the data block IDs, plus additional blocks for storing block boundary points. Each cell stores a list of bottom-level subset IDs overlapping with the cell, and multiple cells can share a block if their lists of bottom-level subsets are not too long.

Overall, this paper makes the following contributions:

- We propose an external-memory friendly hybrid spatial index named FHSIE that works with a grid to learn spatial partitioning and provide efficient query processing. We analyze the time benefits and space costs of adding more levels to help determine the height of the structure.
- We design a simple and efficient grid resolution estimating algorithm to enhance the query performance of our grid-based structure.
- We extend FHSIE to external memory to achieve a scalable structure.
- We design algorithms for point, window and kNN queries and update processing with FHSIE. Our extensive experiments on real and synthetic data sets, under both in-memory and external-memory settings, show that FHSIE can easily index more than 200 million objects.

II. Related Work

We review studies on traditional and learned spatial indices.

Traditional spatial indices: Traditional spatial indices can be classified into three categories: space-partitioning indices, data-partitioning indices and mapping-based indices. *Space-partitioning indices* recursively partition the data space until the spatial objects in each region can fit into an index node. The *k-d tree* [10] and the *quadtree* [11] are two classic space-partitioning indices. Grid-based space partitioning (e.g., *Grid File* [12] and its variants [13], [14]) is another simple and highly effective approach. *Data-partitioning indices* partition a data set into subsets such that each subset fits into an index node. The *R-tree* [15] and its variants are arguably the most commonly used data partitioning indices. *Mapping-based indices* transform multi-dimensional objects into one-dimensional values. The mapped values are then indexed using a one-dimensional index such as the B^+-tree [16]. *Space-filling curves* that overlay the data space with a grid and associate each grid cell with a number (i.e., a curve value) are a typical mapping technique.

Learned spatial indices: Recent studies [6], [8], [17]–[21] take advantage of machine learning techniques to learn data/query distribution and optimize the partitioning structures of traditional spatial indices (e.g., the *Qd-tree* [17] is an optimized version of k-d trees using reinforcement learning), or they directly predict the storage location of multidimensional (spatial) objects based on their coordinates. Our study is more relevant to the second group of studies, i.e., we also use machine learning techniques to help locate a spatial object given its coordinates.

The inherent inaccurate nature of prediction-based machine learning models may result in a large data scan range at query processing. Besides, existing learned spatial indices [5], [6] struggle to offer fully accurate query results for non-point (e.g., window or kNN) queries, because the search space for a non-point query is not foreknown and is difficult to define. Prior studies combine learned methods with additional modules to address this limitation. For example, *LISA* [4], *Flood* [8] and follow-up studies [7], [9] use grid structures. The issues with these indices are that they either map spatial objects to one-dimension values via a fixed grid resolution which may allocate a large subset of points into a partition or require known query workloads for index structure optimization. Our technique avoids such issues by an unsupervised learning approach to learn data partitions from the data distribution directly and a grid resolution estimation method to optimize grid layout based on the cluster radii distribution.

III. FHSIE Structure

Given a set of n points $P = \{p_1, \cdots, p_n\}$ in a d-dimensional space $\boldsymbol{V} = [x^0_{min}, x^0_{max}] \times \cdots \times [x^{d-1}_{min}, x^{d-1}_{max}] \subseteq R^d$, where $[x^i_{min}, x^i_{max}]$ denotes the data domain in dimension i, we aim to construct a structure over P for efficient and accuracy point, window and kNN query processing. For ease of presentation, we use $d = 2$ in the following discussion, although our techniques also apply to any $d > 2$.

A. Learned Partitioning

FHSIE provides highly efficient and fully accurate queries and supports external memory-based data storage. To avoid prediction errors and scanning of multiple data partitions at query processing, we use unsupervised (i.e., clustering) models to pack points into subsets (i.e., clusters) based on their coordinates, although supervised models such as neural networks (e.g., following RSMI [5]) can also be plugged into our structure for partition learning.

Clustering model: We use *K-means* to partition the points according to their coordinates. The target number of clusters K_1 is determined by the data set size n and the data block size B which is a system parameter (i.e., $K_1 = \lceil n/B \rceil$). This induces a clustering time complexity of $O(T_1) = O(ndK_1) = O(nd\lceil n/B \rceil)$, if a single-level data partitioning structure is constructed, and it suggests a high running time when n is large. To address this issue, we design a hierarchical structure.

Hierarchical structure: Consider a height-balance structure where we apply K-Means recursively to partition a data set with a single K_2 value. To obtain n/B bottom-level clusters with an h-level structure, K_2 needs to be $\sqrt[h]{\lceil n/B \rceil}$, assuming that the data points are clustered evenly into the clusters in each level of the structure. Then, the time complexity to build such a hierarchical structure is:

$$O(T_2) = \sum_{j=0}^{h-1} O(\sum_{i=0}^{m} n_i^j dK_2) = \sum_{j=0}^{h-1} O(ndK_2) = O(hndK_2) \tag{1}$$

Here, j represents the level number of the hierarchical structure and i represents the ith partitioning model of the jth level. Each level has n objects, hence, $\sum_{i=0}^{m} n_i^j dK_2 = ndK_2$. Since $h\sqrt[h]{\lceil n/B \rceil} < \lceil n/B \rceil$ when $h > 1$, $hndK_2 = hnd\sqrt[h]{\lceil n/B \rceil} <$

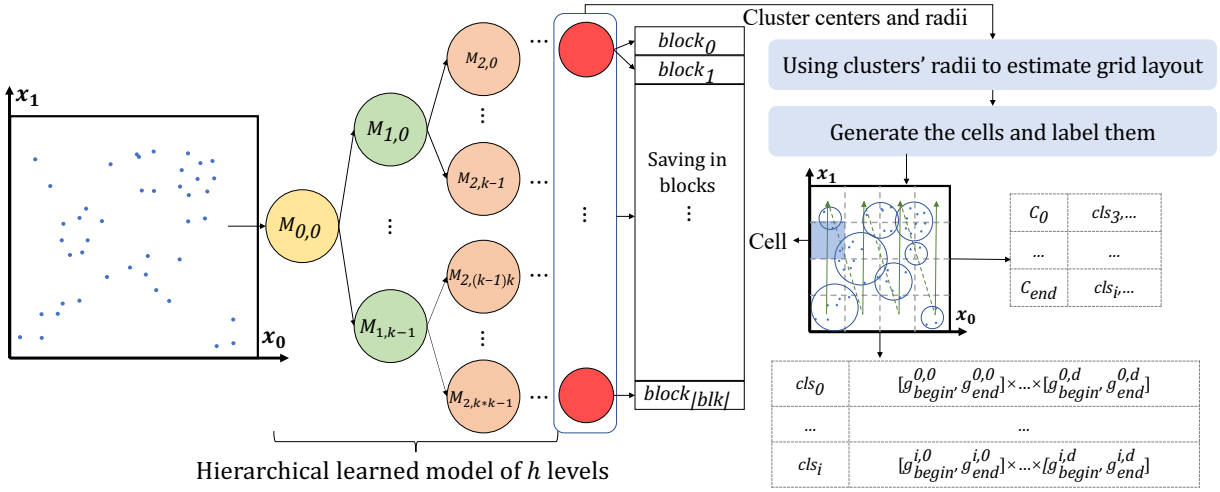


Fig. 2: FHSIE index structure.

$nd\lceil n/B\rceil = ndK_1$. This means that using a hierarchical structure can reduce the index construction cost. However, the hierarchical structure also has a higher space cost (i.e., with extra upper levels in the structure). We aim to make balance the storage space and the index build time costs by analyzing the time benefit of adding an extra level to the structure. In particular, we compute $O(T_2)$ for h and $h+1$, denoted by $O(T_2)_h$ and $O(T_2)_{h+1}$. We heuristically compute $O(T_2)_h/O(T_2)_{h+1} = (h\sqrt[h]{\lceil n/B\rceil})/((h+1)\sqrt[h+1]{\lceil n/B\rceil})$ and find the maximum value of h such that $(h\sqrt[h]{\lceil n/B\rceil})/((h+1)\sqrt[h+1]{\lceil n/B\rceil}) = \frac{h}{h+1}\lceil n/B\rceil^{\frac{1}{h(h+1)}} < 2$. We add another level when the level can bring at least a 2-time speedup for index building heuristically. The resultant value of h is used as the target height of our structure.

After choosing the optimal height, we recursively partition points into subsets until the target height is reached, where each data point p is then assigned to a bottom-level subset.

The unsupervised models cannot guarantee that each bottom-level subset fits the size of a data block (i.e., B). When a bottom-level subset has more than B points, we sort the points in the subset by their coordinates in the first dimension and split them into blocks of size B. We record the coordinate of the first point in the first dimension for each block as the split point of the block.

Grid: We further associate a regular grid to our structure to build many-to-many relationships between cells of the grid (i.e., cells) and bottom-level subsets such that it is more efficient to find the subsets intersecting with a given query. For each cell, we maintain a list of bottom-level subsets overlapping with the cell. Fig. 2 shows the overall structure of FHSIE.

Overall, each internal node of FHSIE contains a list of cluster centers (i.e., sub-models' parameters for locating the child nodes), while each bottom-level node contains a list of cluster centers and the corresponding cluster radii. Further, the IDs and split points of the blocks of each bottom-level cluster are stored with the bottom-level cluster together with the IDs of the cells overlapped by each such cluster.

B. Estimating the Optimal Grid Layout

The grid layout has a substantial impact on the query performance of FHSIE because our window and kNN query algorithms access the cells that overlap with the query windows and then scan the clusters overlapping with the cells to compute the final query results. Small cells may require processing a lot of cells in a query window, while large cells may each overlaps with many clusters which also can incur high cluster scanning costs.

We determine the side length of the cell heuristically. We represent each cluster as a circle. When a cell's side length is greater than a cluster's radius, a cluster can overlap with at most three cells in one dimension. Hence, we use the radii of the bottom-level clusters to estimate the side length of the cells to reduce cluster accesses for query processing. To avoid the negative impact of extremely large clusters, we set the side length of the cells, denoted by cw, as the top 5% largest of cluster radius. Once the cw is determined, we generate the cells by dividing each dimension of the data space X^w into $\theta_w = \lfloor (X^w_{max} - X_{min})^w/cw \rfloor$ columns.

Construction cost: Note that FHSIE has three cost components. The first is the sub-model training (i.e., partition learning) which we discussed earlier. Suppose that in training, at most E epochs (i.e., E K-Means runs) are executed for each sub-model. The total training time is $O(ndh\sqrt[h]{\lceil n/B\rceil}E)$. The second is the sorting cost for all points in each bottom-level cluster which is bounded by $O(n\log n)$. The third is grid preparation cost. This step needs to scan $\lceil n/B\rceil$ clusters. Thus, the time cost is $O(\lceil n/B\rceil)$. Overall, the time cost for FHSIE construction is $O(ndh\sqrt[h]{\lceil n/B\rceil}E + n\log n + \lceil n/B\rceil)$

IV. Query Processing

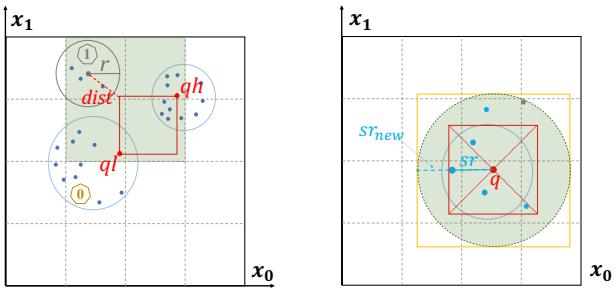
This section presents algorithms to process point, window and kNN queries using our structure.

Point query. Given a query point q, we recursively call the partitioning models from the root model $\mathcal{M}_{0,0}$ until the bottom-level model to locate the cluster containing q, denoted by cls_j. Then, we locate the blocks (denoted by lower and upper bounds of their IDs) of cls_j that correspond to the range

that encloses the dimension-0 coordinate of q (i.e., $q.coor^0$, the coordinate of q in the first dimension), which is done by a binary search over the split points of cls_j. We scan these blocks (usually only one block), and we return point p in such blocks if it has the same coordinates as q.

Window query. For window queries, we use the grid structure of FHSIE to guide the search instead of using the learned models. This is because the models may not have learned the query window boundaries and hence cannot accurately predict the clusters that overlap with a query window.

Given a window query q, as illustrated by the red rectangle in Fig. 3a, we first compute two bounding points, i.e., the bottom-left corner ql and the top-right corner qh. Since we use a regular gird, it is easy to compute the grid cells overlapped by q, using the coordinates of ql and qh, and the side length of the cells cw (Section III-B). We store such cells in a list $list_{ce}$ and take all unique clusters from the cells in $list_{ce}$, which form a list of cluster candidates $list_{cl}$. Then, for every cluster candidate $cls \in list_{cl}$, we filter it by the (Euclidean) distance between its centre $cls.c$ and the query q. There are three relationships, including non-overlapping, partially overlapping and enclosing. We (1) ignore the non-overlapping clusters (cf. circle 1 in Fig. 3a); (2) add all points in enclosing clusters into the result set S and (3) scan partially overlapping clusters to identify qualified points and add them into S. (cf. circle 0 in Fig. 3a). We return S as the query result when all clusters in $list_{cl}$ have been processed.



(a) Window query (b) kNN query

Fig. 3: Window and kNN query examples

***K*NN Query.** Our kNN query algorithm combines a point query with window queries of increasing sizes to compute the query answer. Given a query point q and a query parameter k, we first run a point query using q as the query point to fast initiate a kNN candidate set (i.e., all points in the predicted block). We take the distance between q and the kth nearest neighbor in the predicted block as our initial search radius sr to form a window query centered at q. A boundary case is when the predicted block contains only one point, which shares the same coordinates with q. In this case, we estimate sr by k/n, i.e., $sr = (k/n)^{1/d} \cdot (x^0_{max} - x^0_{min})$. We then run window queries repeatedly and increase the search radius by a factor of $(k/|\hat{S}| + 0.5)^{1/d}$ iteratively to ensure no false kNN dismissals (cf. Fig. 3b).

Since our window query algorithm is based on the grid structure which offers an accurate result, our kNN query algorithm also returns an accurate result.

V. Update Handling

Our FHSIE structure supports insertions and deletions. Given a point q to be inserted, we first run a point query to locate the cluster cls_j and the block $cls_j.block_b$ to host q. If the block is not full, we simply insert q into the block, and we update the split point of $cls_j.block_b$ if necessary. Otherwise, we split $cls_j.block_b$ into two blocks each containing $B/2$ points, insert q into one of the two blocks based on its dimension-0 coordinate, and update the split points of cls_j accordingly. After either case, we need to check if adding q to cls_j has enlarged the radius of the bottom-level cluster. If so, we update the cluster radius $cls_j.r$ and add the cluster to the cluster lists of the cells overlapping with the enlarged cluster. During this process, the internal nodes' parameters would not be updated.

For point deletion, we run a point query to locate the point and remove it from its block (and hence cluster). We shrink the bottom-level cluster radius and update the cluster lists of the overlapping cells accordingly.

VI. External Memory Implementation

We next extend FHSIE to external memory. There are three main components in FHSIE: (1) internal partitioning models, (2) bottom-level partitioning models (with cluster and block information), and (3) cells of the grid. Each component can be saved as blocks in external memory, such that query algorithms in Section IV apply directly.

To support updates, in particular, insertions, takes further consideration. Since the updates mainly impact the bottom-level clusters, we focus on such clusters and present three block storage modes for different storage spaces and update efficiency (Fig. 4).

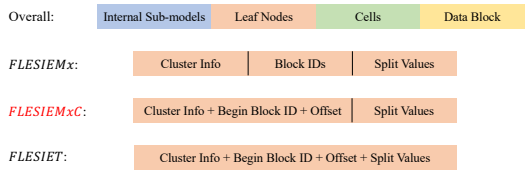


Fig. 4: FHSIE external memory implementation

FHSIEMx: The first storage mode computes the maximum number of blocks needed for any bottom-level cluster and allocates blocks for cluster information (e.g., centers and radii), block IDs and block split points, respectively. This storage mode has a high space overhead, but it is insertion friendly as there are pre-allocated empty blocks.

FHSIEMxC: The second storage mode uses data block offset values to reduce the storage required for the block IDs. It stores the first data block ID and an offset value of all data blocks belonging to the cluster (all such blocks are stored consecutively). During updates, the offset values need to be updated after data insertions or deletions.

FHSIET: The third storage mode stores all meta data of a bottom-level cluster compactly in a few pre-allocated blocks. Only any remaining space of the pre-allocated blocks is used to record insertions of new data blocks. This approach has the smallest index size but suffers from higher update costs.

When a large volume of data needs to be inserted, FHSIEMx is the best choice. When there are no updates, FHSIET is more suitable. FHSIEMxC aims to balance the update and storage costs and may be used for scenarios with a moderate update frequency. In all three cases, an index overhaul is needed when the pre-allocated blocks to store the points or data block IDs are exhausted.

VII. Experiments

All experiments are run on a desktop computer with a 3.20 GHz Inter i9 CPU, 64 GB memory, a 512 GB solid-state drive and a 1 TB SATA hard drive.

Datasets. We use two real-world data sets **Tiger** [22] and **OSM** [23] and synthetic data sets of three different distributions. Tiger consists of rectangles that represent geographical features in 18 Eastern states of the USA. We use the centroid of the rectangles and remove duplicate points, where 16,832,907 points remain in the data set. OSM is extracted from OpenStreetMap covering the USA. We randomly sample 100 million points to form the data set.

We generate synthetic data sets with up to 256 million points following **Uniform**, **Normal** and **Skewed** distributions in a unit square, following HRR [24], [25].

Competitors. We compare FHSIE with the **Grid** [12], **KDB Tree** [26], **HRR** [24], **RSMI** [5] and **LISA** [4]. Except for **RSMI** and **LISA**, other competitors are traditional spatial indices. **RSMI** and **LISA** are two learned spatial indices with released source code. For **Grid**, We build a $\sqrt{n/B} \times \sqrt{n/B}$ regular grid, allocate points to grid cells, and store them in blocks of size B.

The traditional indices are implemented in C++. We use RSMI's released code based on C++ and PyTorch 1.4. We use LISA's released code which is based on Python. We implement FHSIE in C++ and Python separately to compare with these baseline methods. We run the algorithms on the CPU.

We first compare using neural networks (following RSMI [5]) with using K-Means for data partitioning. The results show that K-Means suits our structure better and yields lower query times. Hence, we use K-Means in the rest of the experiments. We set the block size $B = 100$ by default.

TABLE I: Impact of height

Height	h=2	h=3	h=4	h=5
Construction time (s)	4,033	733	350	341
Point query (μs)	1.50	0.29	0.18	0.16

Table I shows that each additional layer brings decreased benefits as the height of FHSIE increases (on Skewed dataset under default setting), which is consistent with the analysis of Section III. Hence, we use the analysis to compute tree heights, which results in use 3 and 4 when $n \leq 8$ million and $n \geq 16$ million, respectively.

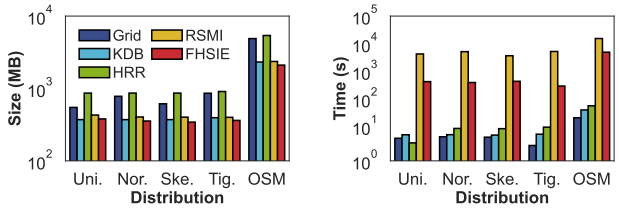
Other parameter settings are summarized in Table II where the default settings are in boldface.

A. Experiment Results

We start with the results under in-memory settings, including the index construction times, point, window and kNN query times, and update times. Then, we will show query times on external memory. Here, due to space limits, we focus on the results of the indices implemented with C++, i.e., Grid, KDB, HRR, RSMI and our FHSIE.

TABLE II: Parameters

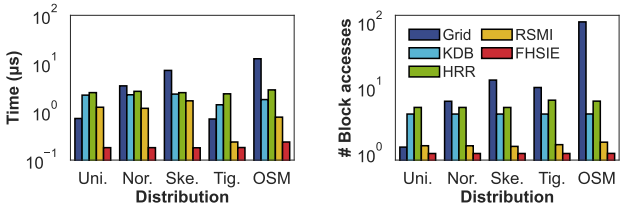
Parameter	Values
Distribution	Uniform, Normal, **Skewed**
Data set size (million)	2, 4, 8, **16**, 32, 64, 128, 256
Query window size (%)	**0.01**, 0.04, 0.09, 0.16, 0.25
k (for kNN queries)	1, 5, **25**, 50, 75
Data insertion ratio (%)	**0**, 10, 25, 50, 75, 100



(a) Index Size (b) Construction Time

Fig. 5: Index size and construction time vs. distribution

1) Index Size and Construction Time: As Fig. 5a shows, our FHSIE index has the smallest size across data sets of different distributions, which attributes to its fewer model parameters (i.e., just the cluster centres). Both learned indices RSMI and our FHSIE are slower than the traditional indices to build. Importantly, FHSIE takes an unsupervised approach which makes it almost one order of magnitude faster than RSMI (cf. Fig. 5b).



(a) Query time (b) Block access

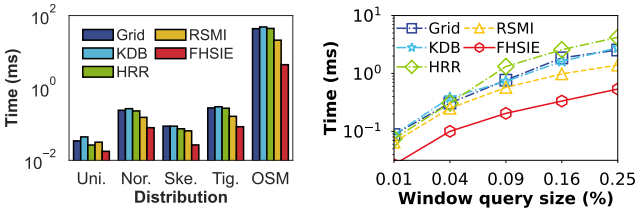
Fig. 6: Point query vs. distribution

2) Point query: We query every data point in a data set and report the average response time and the number of data block accesses per query. The strength of FHSIE on point queries comes from its hierarchical structure and inherent error-free predictions. It takes only 3 or 4 model invocations and a binary search to locate the block address. Since the number of blocks per bottom-level cluster of FHSIE is very small both on average and in the worst case (e.g., 2.04 and 29 on OSM), FHSIE achieves the lowest response time and the number of block accesses as shown in Fig. 6 (e.g., 0.24 μs and 1.01 block accesses vs. 0.78 μs and 1.49 block accesses on OSM comparing with RSMI).

3) Window query: We generate 1,000 window queries following the distribution of each data set and report the average query performance. As Fig. 7a shows, FHSIE is also the fastest for window queries, with at least 1.8 times ($0.018ms$ vs. $0.034ms$ for FHSIE and Grid on Uniform) and up to 10.94 times speedup ($4.39ms$ vs. $48.01ms$ for FHSIE and HRR on OSM). This advantage attributes to the use of a grid structure

to efficiently identify data clusters that overlap with the query window. Note that FHSIE not only outperforms RSMI but also offers fully accurate answers as traditional indices do, while RSMI only offers approximate answers.

Results where we vary the data set size, the query window size, and the number of data updates as well as on kNN queries show similarity comparison patterns. Such results are omitted.



(a) Query time (b) Query time

Fig. 7: Window query vs. distribution vs. window size

4) Comparison against LISA: To compare with LISA, we generate 1,000 window queries for each aspect ratio (i.e., from 0.25 to 4) following the data set distribution, respectively (i.e., 5,000 window queries in total).

Table III summarizes the results. We see that FHSIE is faster than LISA on both synthetic and real data. Meanwhile, FHSIE offers exact query results while LISA may miss a few points.

TABLE III: Window Query Performance against LISA

	Skewed		OSM	
Model	Response time (ms)	Recall	Response time (ms)	Recall
LISA	0.26	99.99%	30.75	99.88%
FHSIE	**0.17**	**100%**	**12.37**	**100%**

5) External Memory Queries: Next, we compare external-memory based implementation of Grid, KDB, HRR and FHSIE. RSMI is omitted since it only has an in-memory implementation. We run experiments with 256 million points.

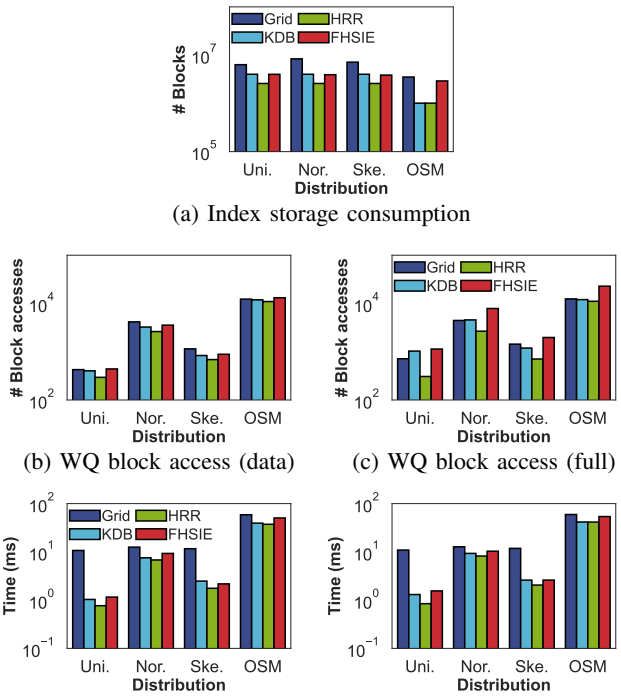
We consider all three storage modes of FHSIE (Section VI). Table IV shows the number of blocks taken by FHSIE using the three storage modes, respectively. In what follows, we use FHSIEMxC for comparisons with the baselines.

TABLE IV: Storage Consumption on OSM

Storage mode	FHSIEMx	FHSIEMxC	FHSIET
Number of blocks	2,947,198	2,914,430	2,883,518

Index storage costs: As Fig. 8a shows, the storage cost of FHSIE is consistently smaller than Grid and smaller than those of KDB on most data sets except for OSM which is quite skewed, where FHSIE forms more blocks.

Window queries: On external memory, FHSIE performs similarly to traditional indices for window queries ("**WQ**") under both settings of placing only data blocks on external memory ("**data**") and placing both data and index blocks on external memory ("**full**"), as Figs. 8b to 8e show. This is because FHSIE uses only dimension-0 coordinates to prune the data blocks in the final pruning step, which suffers in the number of block accesses compared with using MBRs by the traditional indices. Since block access on external memory has a higher cost, the overall query response time of FHSIE now also gets impacted. However, we argue that FHSIE still offers competitive performance in this case, which confirms



(a) Index storage consumption

(b) WQ block access (data) (c) WQ block access (full)

(d) WQ time (data) (e) WQ time (full)

Fig. 8: External-memory index performance

the general applicability of FHSIE under both in-memory and external-memory settings.

Like above, we omit results on the other query settings.

VIII. Conclusions

We proposed FHSIE – a fast hybrid spatial index with external memory support. To construct FHSIE, we recursively invoke clustering models to group spatial objects in a top-down manner and construct a height-balanced hierarchical structure. Then, we associate a grid structure to the bottom-level data clusters, which results in a hybrid learned-and-grid structure for accurate query processing. We further extend FHSIE to an external-memory based implementation. Extensive experiments on real and synthetic data sets with more than 200 million points show that FHSIE is highly efficient and precise no matter where it runs, i.e., in memory or on a disk. It outperforms state-of-the-art learned spatial indices by up to an order of magnitude in query times.

Though FHSIE is faster than traditional indices at the query processing, it is more time-consuming to build. When a large number of indices needs to be built in a short time, traditional indices may still be preferable. In the future, we plan to extend FHSIE to more types of queries, such as continuous spatial queries [27], e.g., to help report users or points of interest nearby continuously.

Acknowledgment

This work is partially supported by Australian Research Council (ARC) Discovery Project DP23010153.

References

[1] N. Beckmann and B. Seeger, "A revised R*-tree in comparison with related index structures," in *SIGMOD*, 2009, pp. 799–812.

[2] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.

[3] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *PVLDB*, vol. 13, no. 8, pp. 1162–1175, 2020.

[4] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *SIGMOD*, 2020, pp. 2119–2133.

[5] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *PVLDB*, vol. 13, no. 12, pp. 2341–2354, 2020.

[6] A. Davitkova, E. Milchevski, and S. Michel, "The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries." in *EDBT*, 2020, pp. 407–410.

[7] S. Zhang, S. Ray, R. Lu, and Y. Zheng, "SPRIG: A learned spatial index for range and *k*NN Queries," in *SSTD*, 2021, p. 96–105.

[8] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *SIGMOD*, 2020, pp. 985–1000.

[9] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," *PVLDB*, vol. 14, no. 2, p. 74–86, 2020.

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

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

[12] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems*, vol. 9, no. 1, pp. 38–71, 1984.

[13] M. Freeston, "The BANG file: A new kind of grid file," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 260–269, 1987.

[14] A. Hutflesz, H.-W. Six, and P. Widmayer, "Twin grid files: Space optimizing access schemes," *ACM SIGMOD Record*, vol. 17, no. 3, pp. 183–190, 1988.

[15] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[16] D. Comer, ""ubiquitous b-tree"," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[17] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *SIGMOD*, 2020, pp. 193–208.

[18] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "XIndex: A scalable learned index for multicore data storage," in *PPoPP*, 2020, pp. 308–320.

[19] A. Llaveshi, U. Sirin, A. Ailamaki, and R. West, "Accelerating B+ tree search by using simple machine learning techniques," in *International Workshop on Applied AI for Database Systems and Applications*, 2019.

[20] X. Li, J. Li, and X. Wang, "ASLM: Adaptive single layer model for learned index," in *DASFAA*, 2019, pp. 80–95.

[21] G. Liu, J. Qi, C. S. Jensen, J. Bailey, and L. Kulik, "Efficiently Learning Spatial Indices," *ICDE,* 2023.

[22] TIGER/Line Shapefiles. (2006) https://www.census.gov/geo/maps-data/data/tiger-line.html. Accessed: 2020-06-10.

[23] OpenStreetMap US Northeast data dump. (2018) https://download.geofabrik.de/. Accessed: 2020-06-10.

[24] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Theoretically optimal and empirically efficient R-trees with strong parallelizability," *PVLDB*, vol. 11, no. 5, pp. 621–634, 2018.

[25] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 3, pp. 1–47, 2020.

[26] J. T. Robinson, "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," in *SIGMOD*, 1981, pp. 10–18.

[27] J. Qi, R. Zhang, C. S. Jensen, K. Ramamohanarao, and J. HE, "Continuous Spatial Query Processing: A Survey of Safe Region Based Techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 64:1–64:39, may 2018.