GLIN: A (G)eneric (L)earned (In)dexing Mechanism for Complex Geometries

Congying Wang*
University at Buffalo
cwang39@buffalo.edu

Jia Yu
WSU, Wherobots Inc.
jia.yu1@wsu.edu

Zhuoyue Zhao
University at Buffalo
zzhao35@buffalo.edu

ABSTRACT

Although spatial indexes shorten the query response time, they rely on complex tree structures to narrow down the search space. Such structures in turn yield additional storage overhead and take a toll on index maintenance. Recently, there have been a flurry of efforts attempting to leverage Machine-Learning (ML) models to simplify the index structures. However, existing geospatial indexes can only index point data rather than complex geometries such as polygons and trajectories that are widely available in geospatial data. As a result, they cannot efficiently and correctly answer geometry relationship queries. This paper introduces GLIN, an indexing mechanism for spatial relationship queries on complex geometries. To achieve that, GLIN transforms geometries to Z-address intervals, and then harnesses an existing order-preserving learned index to model the cumulative distribution function between these intervals and the record positions. The lightweight learned index greatly reduces indexing overhead and provides faster or comparable query latency. Most importantly, GLIN augments spatial query windows to support queries exactly for common spatial relationships. Our experiments on real-world and synthetic datasets show that GLIN has 80%-90% lower storage overhead than Quad-Tree and 60% - 80% than R-tree and 30% - 70% faster query on medium selectivity. Moreover, GLIN's maintenance throughput is 1.5 times higher on insertion and 3 - 5 times higher on deletion.

CCS CONCEPTS

• **Information systems** → **Multidimensional range search**; *Query operators*; • **Theory of computation** → **Data structures design and analysis**.

KEYWORDS

learned index, spatial relationship query

ACM Reference Format:
Congying Wang, Jia Yu, and Zhuoyue Zhao. 2023. GLIN: A (G)eneric (L)earned (In)dexing Mechanism for Complex Geometries. In *11th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '23), November 13, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3615833.3628590

*Part of this work was conducted while the author was affiliated with WSU.

BigSpatial '23, November 13, 2023, Hamburg, Germany

ACM ISBN 979-8-4007-0345-4/23/11…$15.00
https://doi.org/10.1145/3615833.3628590

1 INTRODUCTION

Database Management Systems (DBMSs) often create spatial indexes such as R-Tree [10], JED-Tree [3], and Quad-Tree [23] to accelerate queries on geospatial data. Although spatial index structures shorten query response time, they rely on complex tree structures to narrow down the search space. Such structures in turn yield additional storage overhead and take a toll on index maintenance [30]. Recent works on spatial indices [19, 26] mostly focus on accelerating query speed at the cost of even higher storage overhead. As depicted in Table 1a, R-Tree and Quad-Tree usually cost 10% - 20% additional storage overhead. This leads to significant dollar cost especially now that most enterprises move their data to cloud storage for better security and stability. Table 1b shows the cloud storage cost we collect from Amazon Web Services (AWS) EC2, the most popular cloud vendor. Such storage services are typically charged on a monthly or even an hourly basis, with additional fees for data transfer, which can result in unexpected bills for the users.

Table 1: Index storage overhead and storage dollar cost
(a) Index storage overhead on geospatial data (described in Table 4)

	Boost R-Tree	GEOS Quad-Tree
ROADS (8.3GB)	1.57GB	1.98GB
LinearWater (6.4GB)	461.1 MB	685 MB
Parks (8.5GB)	783 MB	1.01GB

(b) Cloud storage cost on Amazon Web Services EC2 instances

RAM (no. CPU)	SSD storage	Data transfer
32GB (2): $0.16/hour	L1: $0.08/GB/month	Internal 1: $0.01/GB
64GB (4): $0.33/hour	L2: $0.10/GB/month	Internal 2: $0.02/GB
128GB (8): $0.66/hour	L3: $0.12/GB/month	External: $0.09/GB

On the other hand, as open data lake formats such as Apache Parquet [21] and GeoParquet [9], become widely adopted, and as the trend to separate storage and computation layers in the cloud continues, researchers and practitioners are increasingly focusing their efforts on leveraging lightweight index structures. This strategy is aimed at enhancing data skipping efficiency at the storage level. Several works leverage data synopses such as min-max, bounding box, and histograms [11, 24, 29, 30] from the indexed data to navigate queries. They adopt much simpler data structure, which bring down the cost of storing and maintaining the index. However, they compromise on query response time and cannot be easily tailored to geospatial data (e.g., polygons, trajectories, etc.).

Recently, there have been a flurry of works [7, 15, 28] attempting to leverage Machine-Learning (ML) models to simplify the index structures. An index, denoted as $y = f(x)$, can be viewed as an ML model, where x is the lookup key and y is the physical position of the complete record in an array. Theoretically, this ML model learns the Cumulative Distribution Function (CDF) between keys and their positions in a sorted array. Although learned index structures demonstrate promising results on space saving and query speedup as opposed to the traditional B+ Tree index, these approaches only work for 1-dimensional (1-D) sortable values

To remedy that, follow-up works extend the idea to support geospatial points. These approaches [17, 18, 27] partition the multidimensional space to cells and assign IDs to these cells using space-filling curve (e.g., Z-order curve [27]) or mathematical equations [17]. They can reduce data dimension to 1-D and thus can be indexed by learned indexes. They work well for geospatial points but are incapable of handling complex geometries such as polygons and trajectories which are widely available in geospatial data. One reason is that complex geometries can intersect multiple cells and thus have more than one IDs. This leads to duplicates in the final result [31] and introduces additional challenges in index maintenance. In addition, the user must hand-tune the partitioning resolution to find an appropriate cell size that is small enough but also does not introduce too many duplicates. Finding such a sweet spot can be prohibitively expensive especially when indexed geometries go across large regions by nature (e.g., trajectories).

Designing a learned index structure for complex geometries presents several major challenges, stated as follows: (1) *Shapes.* Geometries are collections of various complex shapes including polygons and trajectories, which have been standardized to 7 categories [12]. Geometries are not 1-D or 2-D point values. Thus we cannot establish a CDF from such data to their positions for an existing learned indexes. (2) *Spatial distribution.* Geometries often show skewed spatial distributions in the space. For example, most landmarks, such as parks, hospitals, and government buildings, cluster at major metropolitan regions. The index structures should adapt to such distributions for better prediction performance. (3) *Spatial relationship.* Geometries may have various spatial relationships such as $Contains$, $Intersects$, $Touches$, and $Disjoint$. Given a spatial query, the learned index structures must return all geometries that satisfy the spatial relationship to the query geometry.

This paper proposes GLIN [1], a lightweight learned indexing mechanism for spatial range queries on complex geometries such as points, polygons, and trajectories. GLIN by design produces low storage and maintenance overhead while achieving competitive query performance in common cases, as opposed to Quad-Tree and R-Tree. Moreover, GLIN can work in conjunction with existing regular learned indexes to enable geospatial data support. Our contributions in this paper are summarized as follows:

- GLIN transforms geometries to 1-D sortable values using Z-order curve. We prove that GLIN can always deliver correct results for both *Contains* and *Intersects* spatial relationships.
- For any existing order-preserving learned indexes (Section 4), GLIN can extend it to index Z-address values and further improves the search performance by introducing additional information in leaf models. To the best of our knowledge, it is the first indexing mechanism that enables learned indexes on non-point data.
- GLIN equips efficient algorithms to update its structure for data insertion and deletion while offering the query accuracy guarantees.
- Our experimental analysis on real-world dataset shows that GLIN has 80% - 90% lower storage overhead than Quad-Tree and 60% - 80% than R-tree. Meanwhile, GLIN has faster query response time on medium selectivity. Its update throughput is 1.5 times higher on insertion and 3 5 times higher on deletion.

[1]GLIN GitHub repository: https://github.com/DataOceanLab/GLIN

2 BACKGROUND

Spatial range query. Given a query window Q, a spatial dataset R and a predefined spatial relationship SR, a range query denoted as $range(Q, R, SR)$ finds the geometries in R such that each geometry (denoted as GM) has SR relationship with Q. GM and Q can have any shapes including polygons and lines.

Spatial relationship. SQL/MM3 standard [12] lists a number of possible spatial relationships between two geometries. This includes but is not limited to: Contains, Intersects, Touches, and Disjoint. In this paper, we focus on the two most common spatial relationships: (1) *Contains* (Figure 2 Case 1): given two geometries Q and GM, "Q contains GM" is true if and only if no points of GM lie in the exterior of Q, and at least one point of the interior of GM lies in the interior of Q. (2) *Intersects* (Figure 2 Case 1, 2, 3): given two geometries Q and GM, "Q intersects GM" is true if Q and GM share any portion of space. *Contains* is a special case of *Intersects*. If "Q contains GM" is true, then "Q intersects GM" must be true as well.

Minimum Bounding Rectangle (MBR). An MBR describes the maximum extents of a 2-dimensional geometry in an (x, y) coordinate system. An MBR consists of four values, the minimum and maximum values of x and y coordinates of a geometry, and are represented as two points, $p_{min}(x_{min}, y_{min})$ and $p_{max}(x_{max}, y_{max})$ (see Figure 2 Case 1). The coordinates of the MBR can be easily obtained by iterating every coordinate of a geometry. MBR is often used to approximate geometries since it is a much simpler shape.

Probing and Refinement steps of spatial index search. Most existing spatial indexing mechanisms, such as R-Tree, Quad-Tree, and KD-Tree, approximate complex geometries to their MBR and then build index structures on MBRs. A spatial range query is processed mostly in two steps. (1) *Index probing*: the MBR of the query window is processed against the spatial index. It returns a set of candidate geometries whose MBRs possibly *intersect* the query window MBR. This set of candidates is not the exact answer of this query but is a super set of the answer. (2) *Refinement*: the candidate geometries are checked against the query window using their actual shapes with a spatial relationship such as *Contains* or *Intersects*. The refinement step is computationally expensive due to the complexity of shapes and usually takes more time than the probing step [4]. GLIN also follows this two-step process which can significantly reduce computation cost and index storage overhead. Existing learned spatial indexes [17, 18, 27] only perform the probing step and their results might only be a subset of the exact answer if the underlying data is not points.

3 OVERVIEW

The index structure of GLIN is depicted in Figure 1.

Z-address interval. To establish this CDF and enable the model training process, GLIN assigns each geometry a Z-address interval, an one-dimensional sortable interval (serve as keys), by using a well-known space-filling curve called Z-order curve. In this paper, we also study that how *Contains* and *Intersects* relationships are reflected on Z-address intervals and prove that GLIN can guarantee the query accuracy in both cases.

Index structure. Once the CDF is made available between Z-address intervals and record positions, GLIN can harness an existing order-preserving learned indexes, called the base index, such as

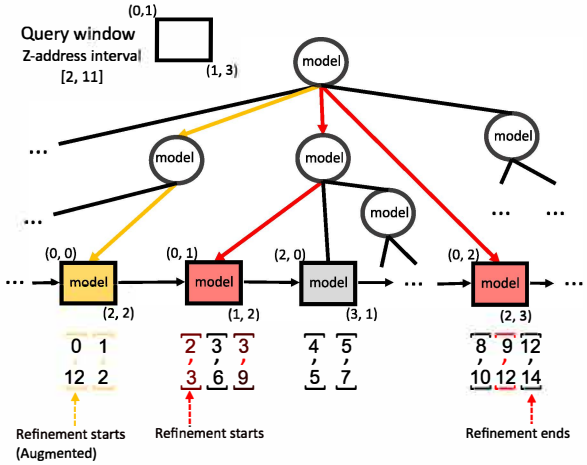


Figure 1: GLIN index structure. White nodes are internal nodes and colored nodes are leaves. Index search: (1) *Contains*: follow the red paths and return red records. The gray node is skipped as its MBR does not intersect the query. (2) *Intersects*: follow the yellow path and the second red path. Red and yellow records will be returned.

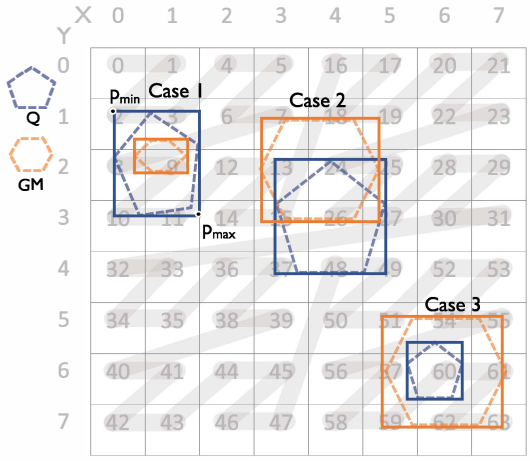


Figure 2: Spatial relationship. Case 1: Q *contains* GM; Case 1, 2, 3: Q *intersects* GM. Z-address(p_{min}) = 2, Z-address(p_{max}) = 11, Z-address interval of Q in Case 1 = $[\![2, 11]\!]$.

Table 2: Notations used in this paper

Term	Definition
Q, GM	Q - a spatial range query window. GM - an indexed geometry. Both can be in any shapes.
MBR	Minimum Bounding Rectangle of a geometry, represented as two points $p_{min}(x_{min}, y_{min})$ and $p_{max}(x_{max}, y_{max})$ MBR_Q - MBR of Q. MBR_{GM} - MBR of GM.
Zmin	Z-address for p_{min} of a MBR $Zmin_Q$ - Zmin of Q, $Zmin_{GM}$ - Zmin of GM.
Zmax	Z-address for p_{max} of a MBR $Zmax_Q$ - Zmax of Q, $Zmax_{GM}$ - Zmax of GM.
Zitvl	Z-address interval described by $[\![Zmin, Zmax]\!]$ $Zitvl_Q$ - Zitvl of Q, $Zitvl_{GM}$ - Zitvl of GM.

ALEX [7] and RadixSpline [14] to model the CDF. Since Z-address intervals cannot 100% preserve the original shape information and spatial proximity, the index probing will return some false positive results. In response, GLIN introduces a refinement phase to prune out all false positives. In addition, it creates a MBR on each leaf node of the hierarchical model to accelerate the refinement phase.

Query augmentation. The basic indexing mechanism in GLIN is designed to handle *Contains* relationship and may produce true negatives for *Intersects* relationship. To remedy that, GLIN employs a piecewise function with outlier handling to augment the query window. More precisely, GLIN will enlarge the Z-address interval of the query window to make sure that it covers all correct results at the cost of additional pruning time.

4 Z-ADDRESS INTERVALS AND ORDER PRESERVING INDEXES

Notations used in this section are summarized in Table 2.

Z-address. Z-order curve is a Z-shape curve (see Figure 2) that connects all 2-dimensional positive integer coordinates in the space. Each coordinate (x, y) will then have a Z-address. The Z-addresses of two nearby coordinates will likely be close to each other. For example, $p_{min}(0, 1)$ in Figure 2 will have a Z-address 2. GLIN rounds down geospatial coordinates to their nearest integers: x = $\frac{longitude-(-180°)}{cell\ size}$ and y = $\frac{latitude-(-90°)}{cell\ size}$. Then it calculates the Z-address using libmorton [2] which interleaves the binary representation of x and y coordinates [16]. GLIN sets the cell size as 5×10^{-7} to represent centimeter-level precision [6]. Detailed discussion can be found in full paper 11.3.

Z-address interval. GLIN assigns a Z-address interval, Zitvl $[\![Zmin, Zmax]\!]$, for every geometry including the indexed geometries and the range query window. It is computed in two steps: (1) find the MBR of a geometry, represented as $p_{min}(x_{min}, y_{min})$ and $p_{max}(x_{max}, y_{max})$; (2) compute the minimum and maximum Z-addresses $Zmin, Zmax$ from p_{min}, p_{max} respectively. In Figure 2 Case 1, Q has $Zitvl_Q[\![2, 11]\!]$.

Notes. (1) When calculating $Zitvl$, GLIN must use p_{min} and p_{max} rather than vertices of the geometry because $Zitvl$ from the latter might not cover all Z-addresses touched by the geometry. For example, vertices of Q in Figure 2 Case 1 indicate $Zitvl$ = $[\![3, 11]\!]$, which misses Z-address 2. (2) We choose Z-order curve due to its monotonic ordering property [16] which guarantees that $Zitvl$ from p_{min} and p_{max} covers the Z-address of any point that falls inside this geometry [27]. In Hilbert curve (or other space filling curves), the $Hmin$ and $Hmax$ of the desired $Hitvl$ are actually on the boundary of the MBR (details omitted due to page limit). To obtain such $Hitvl$, we have to calculate every H-address touched by the MBR. This will significantly slow down the queries and require to tune the cell size which cannot be too large or too small.

Spatial relationship between intervals. Since GLIN is built on Z-address intervals, when we examine the spatial relationship between Q and GM, we also need to understand how this relationship translates to Z-address intervals of Q and GM, denoted as $Zitvl_Q$ and $Zitvl_{GM}$, respectively. We show two important lemmas below which allow us to prune the search space for *Contains* and

Intersects relationship using range scans. The proof is in Appendix 11.

LEMMA 1. **Z-address interval Contains.** *If Q contains GM, then* $Zitvl_Q$ *contains* $Zitvl_{GM}$

where $Zitvl_Q$ *contains* $Zitvl_{GM} \iff Zmin_Q \leq Zmin_{GM} \leq Zmax_Q \wedge Zmin_Q \leq Zmax_{GM} \leq Zmax_Q$.

LEMMA 2. **Z-address interval Intersects.** *If Q Intersects GM, then* $Zitvl_Q$ *intersects* $Zitvl_{GM}$

where $Zitvl_Q$ *intersects* $Zitvl_{GM} \iff Zmin_Q \leq Zmax_{GM} \wedge Zmax_Q \geq Zmax_{GM}$. In other words, $Zitvl_Q$ and $Zitvl_{GM}$ share some portion of the intervals

Order-preserving learned index. GLIN is designed as a general framework to adapt 1-D learned indexes as spatial indexes for polygon relationship queries. One important property that the underlying index must satisfy is *order-preserving*. To explain why, we first define the order-preserving property.

Definition 4.1. Let I be a 1-D range index where the leaves store item keys. The keys in the leaves can be conceptually concatenated into a list in the pre-order traversal of the index. We say a 1-D range index is *order-preserving* if the list is in sorted order.

Order-preserving is crucial for us to correctly prune the search search space in tree probing. Suppose we index the geometries with an order-preserving range index using $Zmin$ of the geometries. As we will show later, Lemma 1 and 2 allow us to search for geometrices with respect to a query geometry Q using a sufficiently large $Zitvl$. Taking *Contains* as an example, the range can be $[\![Zmin_Q, Zmax_Q]\!]$. Since the index is order preserving, we can simply probe the index for the first geometry with $Zmin_{GM} \geq Zmin_Q$ and scan the leaf levels until $Zmin_{GM} > Zmax_Q$. However, we cannot do so if an index is not order preserving as there might be some $Zmin_{GM} > Zmin_Q$ that appears before the first $Zmin_Q$ (if it exists) in the index.

In traditional range indexes such as B-tree (which technically can also be used in GLIN), the list consists of all keys in the leaf level from left to right. It is order-preserving because the tree strictly divides sub-trees into disjoint and increasing key ranges from left to right. Many learned indexes are also order-preserving. For instance, ALEX [7] also divides its key into disjoint and increasing key ranges based on linear regression models, and thus it can be used in GLIN. RMI [15] is an example of non-order-preserving index because an implementation can choose not to enforce the monotonicity of assignment of keys in the internal models when they are complex neural networks. Consequently, we cannot use RMI (or adapt any RMI based spatial indexes such as RSMI [22] to support exact spatial relationship queries).

5 INDEX INITIALIZATION

To create an index, GLIN reads a set of geometry records and initializes the index structure based on the geometries. The mechanism depicted in this section only handles the spatial range query with the *Contains* spatial relationship, which is "the query window contains geometries".

Sort geometries by Z-address intervals. To establish the CDF between keys and record positions, the first step is to put the geometries in a sequential order. GLIN sorts geometries based on their Z-address intervals (see Section 4). The reason is two-fold: (1) Z-addresses can partially preserve the spatial proximity of geometries. This is critical because a spatial range query looks for geometries that lie in the same region. This query will inevitably scan a large portion of the table if the trained ML model does not respect any spatial proximity. (2) A Z-address interval can partially preserve the shapes of geometries. This will make it possible for GLIN to employ different strategies for *Contains* and *Intersects* relationships for the sake of query performance.

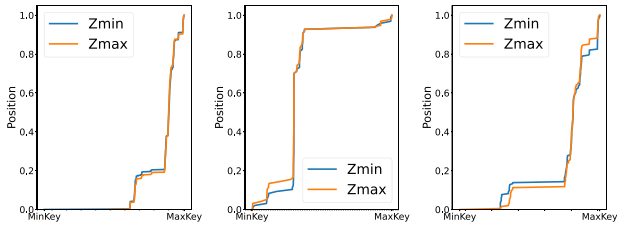


Figure 3: CDFs of different datasets based on $Zmin$ **and** $Zmax$

GLIN iterates through every geometry and calculates the Z-address interval (i.e., $Zitvl = [\![Zmin, Zmax]\!]$) for this geometry. GLIN then sorts these geometries by the $Zmin$ of their intervals. In the rest of the index initialization phase, $Zmax$ of intervals will be completely discarded. In other words, GLIN actually indexes $< Zmin, geometry\ key >$ pairs while traditional spatial indices index $< MBR, geometry\ key >$ pairs. Later, in the index probing phase of the index search, GLIN only checks if the Z-address interval of the query window contains $Zmin$ of geometries (i.e., $Zmin_Q \leq Zmin_{GM} \leq Zmax_Q$). According to Lemma 1, this introduces some false positive results but does not have true-negatives. $Zmax$ will be used in Section 7 for query augmentation. It is worth noting that GLIN could also sort geometries by the $Zmax$ instead of $Zmin$. However, this will not make much difference in the learned CDF models since $Zmax$ will follow the same data distribution of $Zmin$ (see Figure 3). In Section 7, both $Zmin$ and $Zmax$ will be needed in order to handle *Intersects* relationship.

Build the base learned index. Once geometries are sorted by the $Zmin$ of their intervals, GLIN will train a base index to learn the CDF between these $Zmin$ addresses and the record positions. GLIN works in conjunction with any order-preserving learned index and extend it to uphold geometries. These indexes usually possess a hierarchical structure to build models for different regions. In this paper, we adopt the method in ALEX [7] because it supports index updates by design.

Create MBRs in leaf nodes. Since GLIN trains and queries models based on Z-addresses of geometries, the model prediction may introduce more false positives that need to be pruned during the refinement step.

To mitigate this, GLIN employs a simple yet efficient method to reduce the computation cost. When constructing each leaf node of the hierarchical model, GLIN also creates a MBR of all geometries in this node. This can be done by traversing all geometries and finding the overall $p_{min}(x_{min}, y_{min})$ and $p_{max}(x_{max}, y_{max})$. When refining the query results, GLIN will directly skip a leaf node unless the MBR of this node intersects the query window's MBR.

Index maintenance. For insertion, GLIN takes as input a geometry key and inserts it to the index. To insert a new record, GLIN first obtains the $Zmin$ address for the geometry key in this record using the approach described in Section 4, then inserts the record to the base index. Once GLIN puts the new record in a leaf node, it expands the MBR of the leaf node to include the new geometry.

For deletion, the input is a geometry key and GLIN deletes records that have the same key. Similar to the insertion, the first step to delete a geometry key is to get the $Zmin$ address of this geometry. It is possible that several different geometries share the same $Zmin$ addresses. In that case, GLIN only erases records that have the same geometry key. The MBR of the involved leaf node will not be shrunk after the deletion because it requires a scan of all records in this leaf node to get the latest MBR. However, this does not affect the correctness of GLIN as the out-of-date MBR only introduces more false positives instead of true negatives.

6 INDEX SEARCH

Algorithm 1: GLIN Index Search

```
   Input  : A query window Q, a spatial relationship SR
   Output : A set of records Result that satisfy SR with Q
1  /* Step 1: index probing                                   */
2  Zitvl_Q⟦Zmin, Zmax⟧ = calculate_zitvl(Q);
3  if SR is "Intersects" relationship then
4      // Augment the query window
5      Zmin = augment(Zitvl_Q, piecewise function).Zmin;
6  start_position = model_traversal(GLIN.root, Zmin);
7  /* Step 2: refinement                                      */
8  Result = new List();
9  iterator = start_position;
10 while iterator.key ≤ Zmax do
11     geom = Get_Record(iterator);
12     if Q has SR with geom then
13         Result.add(geom);
14     node = iterator.node();
15     if iterator ≥ node.last_position() then
16         while MBR_Q does not intersect node.MBR do
17             node = node.next_node();// Skip this node
18         iterator = node.first_position();
19     else
20         iterator++;
21 Return Result;
```

6.1 Probe the base index

GLIN leverages an existing learned index to build the hierarchical model based on the $Zmin$ addresses of geometries. Therefore, to search the index, this algorithm must first obtain the Z-address interval of the query window (see Section 4). The Z-address interval ⟦Zmin,Zmax⟧ of the query window will then serve as the actual input for the index probing which finds geometries whose $Zmin$ is within the interval. According to Lemma 1, geometries whose $Zmin$ is not within this interval are guaranteed not to be contained by the query window.

The process of searching the hierarchy model is identical to the base learned index on which GLIN is built. As described in Algorithm 1, the index probing step requires a tree-like traversal of the hierarchical model (i.e., *model_traversal*). It uses $Zmin$ address of the query window as the lookup key (see the first red pathin Figure 1). This traversal runs in a top-down fashion starting from the root. The model inside the root node will predict a position in the pointer array using the lookup key. Once the algorithm reaches the leaf node, it will first perform the model prediction to find an approximate position in the record array and then run an exponential search from this position to locate the correct result.

Table 3: Number of records checked during the refinement

	Query selectivity	W/o leaf MBR	W/ leaf MBR
ROADS	1%	3339990	369184
	0.1%	1173710	67474
	0.01%	632839	18244
PARKS	1%	1126520	154685
	0.1%	291197	21700
	0.01%	105076	4180

6.2 Refine the results

The records returned by the index probing have some false positives due to the following reasons: (1) GLIN uses the MBR of each geometry to produce the Z-addresses rather than the actual shape. (2) The Z address interval includes additional Z-addresses. For example, in Case 1 of Figure 2, the MBR_Q only contains 6 addresses (2, 3, 8, 9, 10, 11) but all records whose $Zmin$ values are in⟦2, 11⟧ (the Zitvl of Q, 10 Z-addresses in total) will be returned by the index probing step. (3) the hierarchical model is built upon the $Zmin$ of geometries without considering $Zmax$ at all.

As given in Algorithm 1, GLIN conducts a refinement step to filter out these false positive results and hence offer accuracy guarantee. As illustrated in Figure 1, this refinement starts from the position returned by the $Zmin$-based model traversal and keeps checking if every geometry satisfies the spatial relationship with the query window using their exact shapes, until it arrives at the position whose key is no larger than the query window's $Zmax$. The MBRs in leaf nodes will help GLIN skip non-intersecting nodes directly.

7 QUERY AUGMENTATION FOR INTERSECTS

7.1 The lowest intersecting Z-address

Given a query window Q, Algorithm 1 in Section 6 finds all geometries GM such that $Zmin_{GM} \geq Zmin_Q$. However, an intersecting geometry GM may have $Zmin_{GM} < Zmin_Q$ and $Zmax_{GM} \geq Zmin_Q$ (Lemma 2). Therefore, the algorithm does not return a superset of all the intersecting geometries. For example, Case 2 in Figure 2 shows two intersecting polygons whose Z-address intervals are ⟦7, 27⟧ and ⟦13, 49⟧. Let the first one be an indexed geometry GM and the second one be the query window Q. ⟦7, 27⟧ will be missing from the final results returned by Algorithm 1.

A simple solution is, for all geometries that have $Zmax_{GM} \geq Zmin_Q$, we find the smallest $Zmin_{GM}$, namely the *lowest intersecting Z-address*. Then we augment the query window by taking $min(Zmin_Q, lowest\ intersecting\ Z-address)$. Unfortunately, since the underlying records are sorted by their $Zmin_{GM}$ instead of $Zmax_{GM}$, finding such value for each query requires a full scan, which is prohibitively expensive. Hence, we need a data structure to help GLIN quickly find $lowest\ intersecting\ Z-address$.

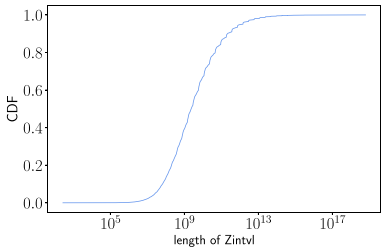


Figure 4: CDF base on $Zintvl$ length

7.2 The piecewise function

The intuition is that we divide the domain range of $Zmax$ to a few sub-domains, and precompute the lowest intersecting Z-address for each sub-domain. Consider a set of N Z-address intervals, let Z be the maximum $Zmax$ among all intervals. Note that all Z-addresses are non-negative and thus the $Zmax$ of all intervals $\in [\![0, Z]\!]$. If we divide the range $[0, Z]$ into k disjoint domains: $[\![0, Z_1]\!]$, $(\!|Z_1, Z_2]\!], \ldots (\!|Z_{k-1}, Z]\!]$, we can define a piecewise-constant function with these k pieces and compute the lowest intersecting Z-address for each. To simplify notation, we denote $Z_0 = -1$ and $Z_k = Z$, and thus the i^{th} piece can be denoted as $(\!|Z_i, Z_{i+1}]\!]$ for $0 \le i < k$. Given a query window Q, if $Zmin_Q \in (\!|Z_i, Z_{i+1}]\!]$, we can find such value on the fly by binary searching the piecewise function, and then augment the query window. It is worth noting that the lowest intersecting Z-address monotonically increases over i. A proof sketch is shown in Appendix 11.

Below is a possible piecewise function for Z-intervals listed in Figure 5. If we want to search for geometries that intersect a query window Q, whose z-address interval is $[\![2, 5]\!]$, then we can use $f(2) = 1$ as the new $Zmin_Q$ to search the base index. Without this function, we will miss a potentially intersecting geometry $[\![1, 2]\!]$ during the index probing phase.

$$f(Zmin_Q) = \begin{cases} 1 & -1 < Zmin_Q \le 3 \\ 3 & 3 < Zmin_Q \le 6 \\ 5 & 6 < Zmin_Q \le 14 \end{cases}$$

Issues with long intervals. Long Z-address intervals could jeopardize the effectiveness of the piecewise function. If we insert interval $[\![0, 14]\!]$ in the dataset, then the piecewise function becomes a contant function with value 0 everywhere. The reason is that the new interval has to be considered in each lower bound of the lowest intersecting z-address because $Zmax_{GM} = 14$ is greater than the lower ends of all intervals. Fortunately, we observe that such long intervals rarely appear in real-world datasets (Figure 4). If we treat them as outliers and separately index them in a different structure, the piecewise function will still produce close lower bounds. In GLIN, we first find out the 99% percentile (an adjustable threshold) of the lengths of the Z-address intervals as the outlier length threshold. We treat all intervals with length longer than that as outliers. They are separately maintained in a smaller outlier index using the base index. For any intersects query, we additionally probe and search the index from $Zmin_Q = 0$ using Algorithm 1.

Greedy construction of the piece-wise function. There is a trade-off of how many intervals we partition the range $[\![0, Z]\!]$ into. On one hand, we can create one interval per distinct $Zmax$ value in the dataset, which provides the exact lowest intersecting z-address for any query window. However, it takes $O(\log N)$ time to augment the query window. On the other hand, we can create one single interval $[\![0, Z]\!]$, which maps to the smallest Zmin for all possible query windows. Augmenting the query is quick with $O(1)$ time, but the number of geometries to go through the refinement step will be very large. Hence, we balance the trade-off by constructing the piecewise function using the following greedy algorithm (Algorithm 2): we sort all geometries' intervals by $Zmin_{GM}$ (which can be done using a leaf level traversal in GLIN without an additional sorting operation), and combine every m intervals (called *piece granularity*) into a single combined interval that exactly covers the m intervals. We treat all combined intervals as the input dataset, denoted as $\{[\![Zmin'_1, Zmax'_1]\!], \ldots [\![Zmin'_{\lceil N/m \rceil}, Zmax'_{\lceil N/m \rceil}]\!]\}$. Then we scan these intervals and create a new piece if the $Zmax$ of an interval is greater than that of the previous one. The higher m is, the less accurate the lower bounds are. A less accurate lower bound leads to more records to refine.

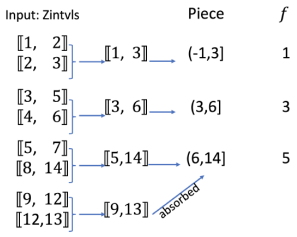


Figure 5: A piecewise funtion with piece granularity $m = 2$.

Figure 5 shows a concrete example of how to build the piecewise function using Algorithm 2 with piece granularity of 2. The input $Zintvl$ is first sorted by $Zmin_{GM}$ (and ties are broken using $Zmax_{GM}$) and we combine every two intervals into a larger interval. For example, the first two $[\![1, 2]\!]$ and $[\![2, 3]\!]$ are combined into $[\![1, 3]\!]$. Then we construct a new piece starting from the previous $Zmax = -1$ (exclusive) until the $Zmax = 3$ (inclusive) of the combined interval, and record the function value $f = Zmin = 1$. Note that, if the $Zmax$ of the combined interval is not larger than the previous $Zmax$ value (e.g, the last combined interval $[\![9, 13]\!]$), it should be absorbed by the previous pieces without any update due to the monotonicity of the piecewise function.

Updating piecewise function. To handle an update, we may also need to update the piecewise functions. For insertion, suppose the inserted geometry is GM. We first check its z-address interval length against the outlier length threshold. If the length is greater than the threshold, we do not need to update the piecewise function. Otherwise, we use binary search to find the first interval $(Z_i, Z_{i+1}]$ such that $Zmin_{GM}$ is in that interval. For all intervals $j \ge i$, we update the function value of range j to $Zmin_{GM}$ if $Zmax_{GM} > Zmin_j$ with $\min\{Zmin_{GM}, f(Zmax_j)\}$. For example, if we insert a non-outlier geometry with its z-addres interval being $[\![6, 8]\!]$, we first find the first piece that contains its $Zmin_{GM} = 6$ and scan forward until it no longer overlaps with the piece. In this case, both the $(\!|3, 6]\!]$ and $(\!|6, 14]\!]$ need to be updated. However, since $Zmin_{GM} = 6$ is already larger than the recorded function values 3 and 5, we will keep the original function values. A subsequent Intersects query will start from either $Zmin = 3$ or $Zmin = 5$, which will be able to find the inserted geometry $[\![6, 8]\!]$. If an inserted geometry has

Algorithm 2: Initialize the piecewise Function

```
Input  : Zintvls sorted by Zmin I, Piece_granularity m
Output : piecewise function PW
c = 0;
prev_zmax = -1;
foreach Zintvl in I do
    if c == 0 then
        current_zmin = Zintvl.Zmin;
        current_zmax = Zintvl.Zmax;
    else
        current_zmax =max(Zintvl.Zmax, current_zmax) ;
    c ++ ;
    if c == m then
        if current_zmax > prev_zmax then
            PW.pushback(current_zmax, current_zmin);
            prev_zmax = current_zmax;
        counter = 0 ;
if c> 0 then
    if current_zmax > prev_zmax then
        PW.pushback(current_zmax, current_zmin);
Return PW ;
```

Table 4: Dataset description

Name	Size	Cardinality	Type	Description
LINEARWATER (LW) [25]	6.44GB	5.8M	LineString	Paths of rivers in the USA
Roads [25]	8.29GB	19M	LineString	Paths of roads in the USA
Parks [20]	8.53GB	9.8M	Polygon	Boundaries of parks and green areas on the planet

z-value larger than the maximum value in the piecewise function, we will append a new piece at the end of the function.

For deletion, we do not perform any update because the piecewise function would still provide lower bounds of the lowest intersecting Z-addresses. However, we can end up with unnecessary refinements if there are many deletions. Hence, we periodically rebuild the piecewise function using GLIN if the lower bounds are too loose and the number of refinements on average becomes too large. Since deletion is less frequent than insertion in a typical workload, it is not unreasonable to amortize the rebuild cost across a large number of deletions.

8 EXPERIMENTS

This section presents the result of an experimental analysis on GLIN on query-only workloads, maintenance-only workloads and hybrid workloads (depicted in Section 11.4 in the interest of space). All experiments are done in the main memory of a machine with 12th Gen Intel Core i9 CPU, 128GB memory, 1TB SSD storage.

8.1 Experiment setup

Implementation details. We implement GLIN on top of ALEX using C++ since ALEX is open-source and supports data updates. Our implementation re-uses the hierarchical model and gapped arrays from ALEX and keeps the corresponding ALEX parameters unchanged. With that being said, GLIN can be easily migrated to other learned indexes too. Furthermore, our idea is also compatible with other one-dimensional index structures, such as B-trees.We illustrate this by effortlessly implementing GLIN on top of a B-tree, which incurs a minimal overhead with promising performance.

Compared approaches. (1) Boost-Rtree: from Boost C++ v1.73.0, with default settings. (2) Quad-Tree: from GEOS v3.9.0. Quad-Tree.(3) GLIN-ALEX: This is the approach proposed in this paper. When querying for the *Contains* relationship, no query augmentation is needed. However, for the *Intersection* relationship, query augmentation comes into play.(4) GLIN-BTREE: This uses the same approach as GLIN-ALEX but with TLX-BTree as the base index, demonstrating the adaptability of GLIN and the benefits derived from a one-dimensional index structure. **Datasets.** We test our approaches on 3 real-world datasets(see Table 4), including polygon and line string data. Real-world datasets are obtained from the US Census Bureau TIGER project [25] . These datasets are cleaned by SpatialHadoop [8].

Query selectivity. We test GLIN on 3 range query selectivities: 1%, 0.1%, 0.01%. To generate a range query with the required selectivity, we randomly take a geometry from the dataset and do a K Nearest Neighbor query around this geometry (K = selectivity * dataset cardinality) using JTS STR-Tree The MBR of the KNN query results then becomes the query window at this selectivity. We generate 100 such query windows per selectivity per dataset.

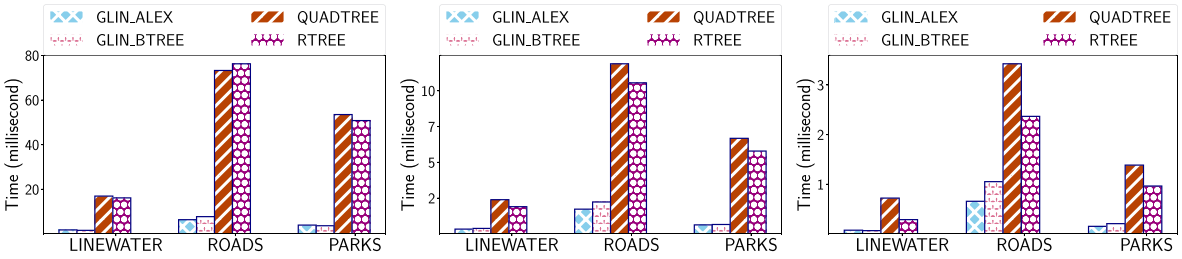
Query response time. The measured query response time consists of two parts: (1) index prob time. For all compared approaches, this is the time spent on searching the index structure. For GLIN-ALEX and GLIN-BTREE, this also includes the query augmentation time when check *Intersects*. (2) Refinement time. For all compared approaches, this is the time spent on refining the query results using the exact shapes of query windows and geometries. For the *Contains* relationship, the results are refined using the *Contains* check in GEOS while for the *Intersects* relationship, the refinement uses the *Intersects* check.

8.2 Query response time

Query response time for *Contains*. As shown in Figure 6, on 1% - 0.01% selectivity, the index probing time of GLIN-ALEX and GLIN-BTREE is nearly 30% - 80%shorter than Quad-Tree and R-Tree. On 0.01% selectivity, GLIN-ALEX and GLIN-BTREE are still 1 times to 3 times faster than R-Tree and Quad-Tree. This makes sense because GLIN-ALEX uses the model prediction-based traversal and GLIN-BTREE uses pointer to traverse while Quad-Tree and R-Tree do the comparison-based tree traversal. When check *Contains*, there is no need to perform the query augmentation.

Query response time for *Intersects*. Figure 7 demonstrates the query performance of both GLIN-ALEX and GLIN-BTREE when they incorporate query augmentation to perform query with the *Intersects* relationship. Performing query augmentation in GLIN introduces additional overheads during index construction and querying. These overheads include: an additional traversal of the leaf level to construct the piecewise function, the piecewise function to augment the query window, a small auxiliary index (either ALEX or BTREE) to handle outliers, and an extra search on the piecewise function when augmenting the query window. Despite these overheads, as evidenced by Figure 7, they do not significantly impact the query performance. Both GLIN-ALEX and GLIN-BTREE can achieve a query performance comparable to that of GLIN when it does not utilize query augmentation.

False positives. Figure 10 illustrates the number of records checked during the refinement. A lower value indicates less false positives which eventually leads to less refinement time and overall



(a) 1% Selectivity (b) 0.1% Selectivity (c) 0.01% Selectivity

Figure 6: Query response time on different query selectivities with *Contains* relationship

(a) 1% Selectivity (b) 0.1% Selectivity (c) 0.01% Selectivity

Figure 7: Query response time on different query selectivities with *Intersects* relationship

Figure 8: Index size with the piecewise function

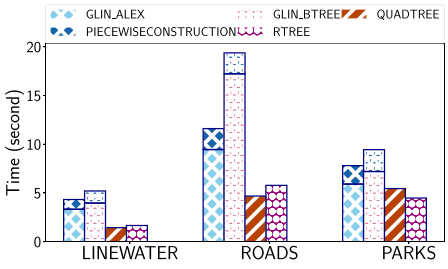


Figure 9: Index initialization time

query response time. GLIN has 10% - 50% more false positives than R-Tree on 1% to 0.1% selectivity. But the fast index probing of GLIN makes up the difference so its query response time still outstanding to others. As expected, GLIN with query augmentation introduces more false positives. the larger the selectivity, the more number of record to check during the refinement.

8.3 Indexing overhead

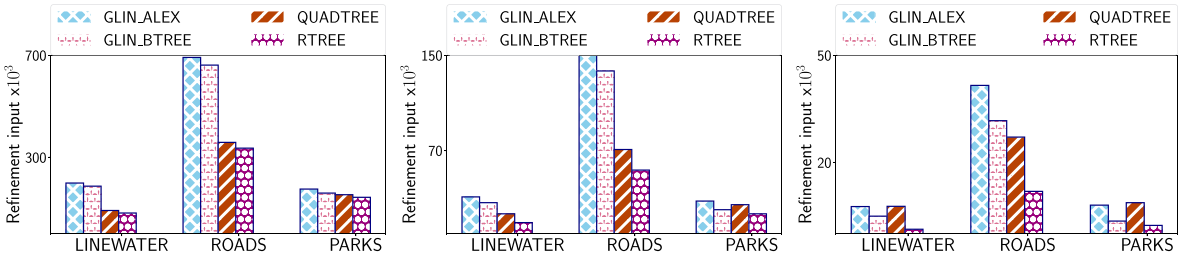
Index size. We measure the sizes by combining the size of internal nodes and the size of leaf node metadata. As demonstrated in Figure 8, the index size of GLIN is 80% - 90% smaller than that of the Quad-Tree, and 60% - 80%times smaller than the R-Tree when tested on real-world datasets. This is reasonable considering that GLIN has far fewer nodes, and each internal node employs a simple linear regression model comprised only of two parameters. Additionally, we calculated the sizes of GLIN-ALEX and GLIN-BTREE, including the piecewise function, and found that this part is very small, so it does not alter our conclusion.

Index initialization time. As depicted in Figure 9, GLIN requires 10% - 50% more initialization time compared to Quad-Tree and R-Tree on real-world datasets. This is understandable as, during index initialization, GLIN needs to sort geometries by their *Zmin* values and train models. GLIN with query augmentation takes approximately 10% more time than GLIN because it needs to generate the piecewise function and handle outliers. The top part of GLIN-ALEX and GLIN-BTREE, depicted in a deeper color, represents the construction time of the piecewise function and additional auxiliary index for query augmentation, which is not significantly more than the original construction time of GLIN.

8.4 Tuning GLIN parameters

This section studies the impact of the piece_ granularity (m) parameter. This parameter defines the number of records summarized by each piece of the piecewise function.

Index probing time. As shown in Figure 11, piece_granularity has a significant impact on the index probing time. The probing time for piece_granularity = 9 million is up to twice as high as that



(a) 1% selectivity (b) 0.1% selectivity (c) 0.01% selectivity

Figure 10: Number of records checked during the refinement

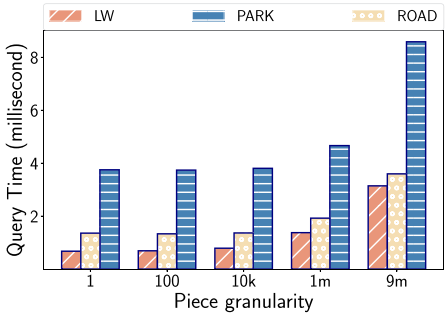


Figure 11: GLIN query time on piece granularity (*Intersects*)

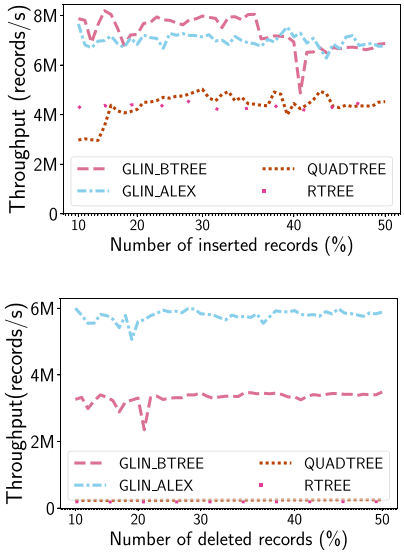


Figure 12: Index maintenance performance on ROADS

for piece_granularity = 10000. A larger piece_granularity results in higher index probing time because it can augment every query window to a small *Zmin*. This may potentially cause every query to start from the beginning of the leaf level, resulting in a time-consuming leaf level scan and more false positive refinement. The condition for piece_granularity = 9 million illustrates this scenario. Conversely, a smaller piece_granularity doesn't significantly reduce the query response time. Even though the query window might not be augmented to a small *Zmin*, leading to more refinement, the search within the piecewise function will take longer for each augmented query window. As a result, the query response time for piece_granularity = 1 is not significantly smaller than the time for larger granularities, such as 10000.

Index size. Compared to the index size of GLIN (see Figure 8), the storage overhead of the piecewise function is negligible. Each piece contains a *Zmax*, indicating where the current piece ends, and a *Zmin*, used to augment a query window if it falls within this piece. As such, the storage overhead of the piecewise function is insubstantial. Moreover, Figure 8 also incorporates the auxiliary index, yet the overall size remains significantly smaller than those of QUADTREE and RTREE. Therefore, we can conclude that the storage overhead of GLIN is indeed negligible.

Therefore, we use piece_granularity = 10000 as the default parameter as it shows the good performance in Figure 11 and its piecewise function size is very small compared to GLIN index size.

8.5 Maintenance Overhead

Insertion. For each dataset, we first bulk-load a random 50% of the data into all indexes and then insert the remaining 50% into the indexes record by record. As shown in Figure 12(a), on larger datasets, the throughput of GLIN is around 1.5 times higher than R-Tree and 1.2 times higher than Quad-Tree. This makes sense because GLIN's index probing is orders of magnitude faster than others and no refinement is needed for insertion. GLIN occasionally show performance downgrade because of node expansion or splitting.

Deletion. For each dataset, we first bulk-load the entire dataset and then randomly delete 50% of the data record by record. As shown in Figure 12(b), the throughput of GLIN is around 3 - 5 times higher than R-Tree and Quad-Tree as GLIN's index probing is orders of magnitude faster than others. The throughput ofGLIN occasionally show performance downgrade because of node merging.

9 RELATED WORK

Learned indexes. Recursive model index(RMI) [15], a tree-like hierarchical model, to learn the cumulative distribution function(CDF) between keys and their position. RMI takes as input a lookup key and predicts the corresponding position by using the models level by level. Compared to B+ Tree, RMI possesses low storage overhead with an outperforming lookup performance but only supports read-only workload. Hermit [28] is a learned secondary index that leverages a hierarchical machine learning model to learn the correlation between two columns. ALEX [7] is an updatable learned

index which adopts RMI's hierarchy structure but adds gapped arrays and node splitting to absorb data updates.

Learned spatial indexes. Researchers have been working on extending learned indexes to uphold spatial and multi-dimentional point data. ZM-index[27] leverages the Z-order space-filling curve to sort the data and then builds RMI on them. Given a spatial range query, it first maps a range query to two Z addresses, then uses the prebuilt machine learning model to find an approximate range for further investigation. Although both GLIN and ZM-Index make use of Z order curve, ZM-index cannot handle non-point data and only works with read-only workload. Flood [18] also employs the RMI to support multidimensional data. It proposes an in-memory read optimized index that partitions a d-dimensional space with a d-1 dimensional grid. The model will predict the grid cell that contains the lookup key. The ML-Index[5] utilizes the iDistance [13] to map data points to the one-dimensional value and also employs the RMI to index the values further. Qi et al. [22] come up with a recursive spatial model index called RSMI to improve the ZM-index. Their work mitigates the uneven gap problem by using a rank space-based transformation. However, this work provides approximate answers. Li et al. propose LISA [17], a disk-based learned spatial index that can reach a low storage consumption and I/O cost. LISA partitions the space to grids and assigned each grid an ID by applying the partially monotonic function. All of the works mentioned above focus on making learned indexes work for 2 or multi-dimensional point data. Unfortunately, in the real world, geospatial data is more than just points. GLIN handles all types of geometries and hence is a practical alternative to R-Tree or Quad-Tree.

Lightweight index structures. Some other studies focus on succinct index structures which take advantage of data synopses from the indexed data and quickly skip irrelevant data. Column imprints [24] utilizes the idea of cache conscious bitmap indexing to create a bit map for each zone. Block Range Indexes (BRIN) in Postgres stores min/max values for each range of disk blocks. Hippo [29] extends BRIN's idea but implements partial histograms in each range to decrease the query response time. Hentschel et al. propose Column Sketch[11] that makes use of lossy compression to generate data synopses and hence accelerates table scan. BF-tree[1] applies bloom filter in the leaf node of a B-Tree and hence reduces the storage overhead. However, these succinct index structures reduce the index storage overhead at the cost of additional query response time and cannot be easily tailored to complex geometries.

10 CONCLUSION

This paper introduces GLIN, a lightweight learned index for spatial range queries on complex geometries. In terms of storage overhead, GLIN is 80% - 90% less than Quad-Tree and 60% - 80% times less than R-Tree. Moreover, GLIN's maintenance speed is around 1.5 times higher on insertion and 3 -5 times higher on deletion as opposed to R-Tree and Quad-Tree. If the application only needs the *Contains* relationship, the user can opt to use GLIN without query augmentation the query response time is 30% - 80% shorter than Quad-Tree and R-Tree on medium selectivity. GLIN with query augmentation deals with both spatial relationships still showing a 30%-70% faster than Quad-Tree and R-Tree query response time on medium selectivity. In a nutshell, GLIN is a lightweight indexing mechanism for medium selectivity queries which are commonly used in spatial analytic applications.

REFERENCES

[1] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *PVLDB* 7, 14 (2014), 1881–1892.
[2] Jeroen Baert. 2018. Libmorton: C++ Morton Encoding/Decoding Library. https://github.com/Forceflow/libmorton.
[3] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *CACM* 18, 9 (1975), 509–517.
[4] Panagiotis Bouros and Nikos Mamoulis. 2019. Spatial joins: what's next? *ACM SIGSPATIAL Special* 11, 1 (2019), 13–21.
[5] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *EDBT*. 407–410.
[6] Degree-pricision. 2011. Accuracy versus decimal places. http://wiki.gis.com/wiki/index.php/Decimal_degrees.
[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984.
[8] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*. 1352–1363.
[9] Geoparquet. 2019. GeoParquet. https://github.com/opengeospatial/geoparquet.
[10] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
[11] Brian Hentschel, Michael S. Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *SIGMOD*. 857–872.
[12] ISO. 2021. ISO/IEC 13249-3:2016 Information technology — Database languages — SQL multimedia and application packages — Part 3: Spatial. https://www.iso.org/standard/60343.html.
[13] H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2005. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *TODS* 30, 2 (2005), 364–397.
[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *SIGMOD*. 5:1–5:5.
[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.
[16] Ken C. K. Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. 2007. Approaching the Skyline in Z Order. In *VLDB*. 279–290.
[17] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*. 2119–2133.
[18] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.
[19] Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2017. BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. In *SSDBM*. 15:1–15:12.
[20] OSM 2007. OpenStreetMap.
[21] Parquet. 2019. Apache Parquet.
[22] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 11 (2020), 2341–2354.
[23] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *CSUR* 16, 2 (1984), 187–260.
[24] Lefteris Sidirourgos and Martin L. Kersten. 2013. Column imprints: a secondary index structure. In *SIGMOD*. 893–904.
[25] Tiger. 2017. TIGER/Line files. https://gisgeography.com/tiger-gis-data-topologically-integrated-geographic-encoding-referencing/.
[26] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A Two-layer Partitioning for Non-point Spatial Data. In *ICDE*. 1787–1798.
[27] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. 569–574.
[28] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *SIGMOD*. 1223–1240.
[29] Jia Yu and Mohamed Sarwat. 2016. Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems. *PVLDB* 10, 4 (2016), 385–396.
[30] Jia Yu and Mohamed Sarwat. 2017. Indexing the Pickup and Drop-Off Locations of NYC Taxi Trips in PostgreSQL - Lessons from the Road. In *SSTD (Lecture Notes in Computer Science, Vol. 10411)*. 145–162.
[31] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial data management in apache spark: the GeoSpark perspective and beyond. *GeoInformatica* 23, 1 (2019), 37–78.

11 APPENDIX

11.1 Proofs of Lemma 1 and Lemma 2

To prove the lemmas, we first show a well-known result in Theorem 1 and additionally prove Theorem 2.

THEOREM 1. **Monotonic ordering** *[16]: Data points ordered by non-descending Z-addresses are monotonic in a way that a dominating point is placed before its dominated points.*

where *dominance* is defined as: given two points p and p', if p is no larger than p' in any dimension, then we say p dominates p'.

THEOREM 2. *If Q contains GM, then MBR_Q contains MBR_{GM}*

where MBR_Q contains $\text{MBR}_{GM} \iff$ Q's p_{min} dominates GM's p_{min} and GM's p_{max} dominates Q's p_{max}.

PROOF. Since Q *contains* GM, in a 2D space, if there is a line, it is obvious that the geometrical projection of Q on this line must contain the geometrical projection of GM on this line. In other words, when Q *contains* GM, then if a person stands somewhere outside Q, he or she should never see GM because GM is completely inside Q assuming Q is a closed geometry. The projection of a geometry on X axis and Y axis are $[x_{min}, x_{max}]$ and $[y_{min}, y_{max}]$, respectively. We have (1) Q's $x_{min} \leq$ GM's x_{min} & Q's $y_{min} \leq$ GM's y_{min}, so Q's p_{min} dominates GM's p_{min} (2) GM's $x_{max} \leq$ Q's x_{max} & GM's $y_{max} \leq$ Q's y_{max}, so GM's p_{max} dominates Q's p_{max}. □

Proof of Lemma 1:

PROOF. $Zmin_{GM} \leq Zmax_{GM}$ is known. Since Q *contains* GM, we have (1) p_{min} of Q dominates p_{min} of GM, so $Zmin_Q \leq Zmin_{GM}$ (2) p_{max} of GM dominates p_{max} of Q, so $Zmax_{GM} \leq Zmax_Q$. □

Proof of Lemma 2:

PROOF. Since Q *Intersects* GM, Q and GM must share some portion of the space. On the other hand, Q and GM's *Zitvl* guarantee to cover any point that falls inside Q and GM, respectively. Therefore, $Zitvl_Q$ and $Zitvl_{GM}$ share some portion of the intervals. □

11.2 Monotonicity of piecewise functions

LEMMA 11.1. *The piecewise function for query augmentation is a non-strict monotonically increasing function.*

PROOF. To show that by contradition, suppose there are two $Zmax$ values $Zmax_1 < Zmax_2$ but the lowest intersecting Z-addresses $f(Zmax_1) > f(Zmax_2)$. Let the pieces containing $Zmax_1$ and $Zmax_2$ be the i_1 and i_2. Since the computed f values are different and $Zmax_1 < Zmax_2$, we must have $i_1 < i_2$. For the second piece, there must be a z-address interval $[\![f(Zmax_2), Zmax'_2]\!]$ such that $Zmax'_2 \in [\![Z_{i_2}, Z_{i_2+1}]\!]$. Then we can show that the z-address interval $[\![f(Zmax_2), Zmax'_2]\!]$ intersects with the first piece:

$$\begin{aligned} f(Zmax_2) &< f(Zmax_1) && \text{(assumption)} \\ &\leq Z_{i_1+1} && \text{(definition of } f\text{)} \\ &\leq Z_{i_2} && \text{(since } i_1 < i_2\text{)} \\ &\leq Zmax'_2 && \text{(definition of } f\text{)} \end{aligned}$$

Therefore, $f(Zmax_1)$ should have been smaller than or equal to $f(Zmax_2)$, which is a contradition. □

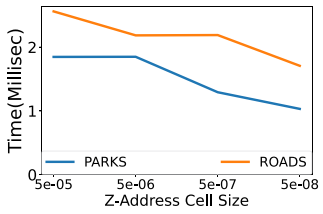
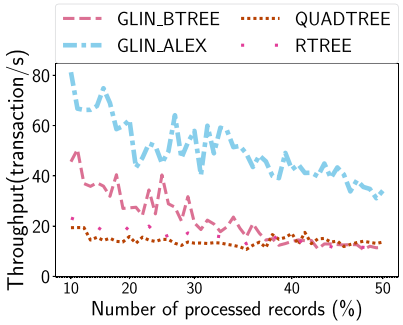
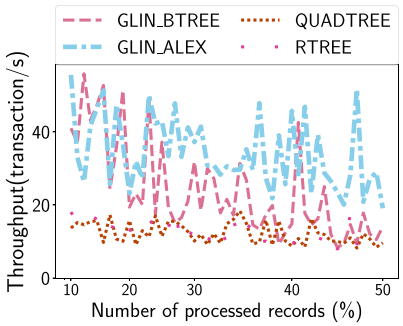


Figure 13: Query response time per cell size



(a) Read-intensive on Roads



(b) Write-intensive on Roads

Figure 14: Index performance on hybrid workloads

11.3 Z-address cell size

The cell size can be any small number with 7 to 8 decimal places (see Figure 13). This way, we can prevent too many geospatial coordinates from having the same Z-address, which would otherwise make indexe unable to prune the refinement candidates.

11.4 GLIN's performance on hybrid workload

We define a transaction as (1) query: a spatial range query with *Intersects* relationship at 1% selectivity, or (2) insertion: insert 1% new records into the indexes. We have two hybrid workloads: (1) read-intensive: 90% of the transactions are queries and the other 10% are insertion. (2) Write-intensive: 50% of the transactions are queries and the rest are insertion. For each dataset, we first bulk-load 50% of the entire dataset, and then start the workloads. We stop when the remaining 50% data are inserted.

Read-intensive workload. As depicted in Figure 14a, the throughput of GLIN-ALEX and GLIN-BTREE initially surpasses that of

Quad-Tree and R-Tree, but eventually aligns with the throughput of R-Tree and Quad-Tree towards the end. This behavior is expected, as GLIN-BTREE requires more rebalancing as more data is inserted into the tree. Meanwhile, GLIN-ALEX is able to consistently maintain higher throughput due to its learned index structure.

Write-intensive workload. As shown in Figure 14b, GLIN outperforms Quad-Tree and R-Tree almost all the time. This matches our expectation because the insertion speed of GLIN is much higher than that of Quad-Tree and R-Tree (see Figure 12). When we have a write-intensive workload, the overall performance of GLIN is proven to be better.