

Enhancing In-Memory Spatial Indexing with Learned Search

Varun Pandey ¹, Alexander van Renen ³, Eleni Tzirita Zacharatou ⁴, Andreas Kipf ³, Ibrahim Sabek ⁵, Jialin Ding ⁶, Volker Markl ^{1,2} und Alfons Kemper ⁷

Abstract: Spatial data is generated daily from numerous sources such as GPS-enabled devices, consumer applications (e.g., Uber, Strava), and social media (e.g., location-tagged posts). This exponential growth in spatial data is driving the development of efficient spatial data processing systems.

In this study, we enhance spatial indexing with a machine-learned search technique developed for single-dimensional sorted data. Specifically, we partition spatial data using six traditional spatial partitioning techniques and employ machine-learned search within each partition to support point, range, distance, and spatial join queries. By instance-optimizing each partitioning technique, we demonstrate that: (i) grid-based index structures outperform tree-based ones (from $1.23\times$ to $2.27\times$), (ii) learning-enhanced spatial index structures are faster than their original counterparts (from $1.44\times$ to $53.34\times$), (iii) machine-learned search within a partition is $11.79\% - 39.51\%$ faster than binary search when filtering on one dimension, (iv) the benefit of machine-learned search decreases in the presence of other compute-intensive operations (e.g. *scan* costs in higher selectivity queries, *Haversine* distance computation, and *point-in-polygon* tests), and (v) index lookup is the bottleneck for tree-based structures, which could be mitigated by linearizing the indexed partitions.

Keywords: spatial data, indexing, machine-learning, spatial, geospatial

1 Introduction

In today's data-driven world, the volume of spatial data is rapidly increasing. For example, the NYC Taxi Rides dataset [NYC19] includes over 2.7 billion rides since 2009, corresponding to more than 650,000 rides per day. However, this is only a fraction of the location data captured by many applications today. Uber, a popular ride-hailing service, operates on a global scale and completed 10 billion rides in 2018 [Ub18]. To meet the increasing demands of spatial applications today, there are numerous research efforts on scale-out systems [Aj13, AN20, EM15, El17, HGS17, Ta16, Th19, To20, Xi16, YZG15, YWS15, El21], databases [Go19, Ma19, Mon13, Ora19, Pa16], improving spatial query processing [Ga20, Ki20a, Ki18, Si18, Ts19, Tz19, Wi21, Tz21, Cu22, Vu21b, SE22, GTM23], or leveraging modern hardware and compiling techniques [DF20, TR20, Tz17, DF22b, DF22a].

¹ Technische Universität Berlin, varun.pandey@tu-berlin.de,  <https://orcid.org/0000-0002-1314-9061>; volker.markl@tu-berlin.de,  <https://orcid.org/0009-0009-0964-026X>

² Berlin Institute for the Foundations of Learning and Data (BIFOLD), volker.markl@tu-berlin.de,  <https://orcid.org/0009-0009-0964-026X>

³ Technische Universität Nürnberg, alexander.van.renen@utn.de,  <https://orcid.org/0000-0002-6365-4592>; andreas.kipf@utn.de,  <https://orcid.org/0000-0003-3463-0564>

⁴ IT University of Copenhagen, elza@itu.dk,  <https://orcid.org/0000-0001-8873-5455>

⁵ University of Southern California, sabek@usc.edu,  <https://orcid.org/0009-0006-2102-5241>

⁶ Amazon Web Services, jialind@amazon.com,  <https://orcid.org/0009-0002-1772-450X>

⁷ Technische Universität München, kemper@in.tum.de,  <https://orcid.org/0009-0003-9066-271X>

Recently, Kraska et al. [Kr18] proposed using learned models instead of traditional database indexes to predict the location of a key in a sorted dataset and demonstrated that they are typically faster than binary searches. Kester et al. [KAI17] showed that index scans are more efficient than optimized sequential scans in main-memory analytical engines for queries that select a small portion of the data. In this paper, we build on these recent research results and thoroughly investigate the impact of applying ideas from learned index structures (e.g., Flood [Na20]) on classical multidimensional indexes. Specifically, we focus on six core spatial indexing techniques, namely linearization using Hilbert space-filling curve, fixed-grid [BF79], adaptive-grid [NHS84], Kd-tree [Be75], Quadtree [FB74] and STRtree [LEL97] for *read-only* datasets. Query processing using these indexing techniques typically consists of three phases: index lookup, boundary refinement, and scanning. The index lookup phase identifies the intersecting partitions, the boundary refinement phase locates the lower bound of the query on the sorted dimension within the partition, and the scan phase scans the partition to find the qualifying points. Section 2.3 provides more details on these phases. In this paper, *we propose using learned models, such as RadixSpline [Ki20b], to replace the traditional search techniques (e.g., binary search) used in the boundary refinement phase.*

Interestingly, we discovered that using a learned model as the search technique for boundary refinement can significantly improve query runtime, particularly for low-selectivity⁸ range queries. This applies to various queries, but the benefit decreases when other dominant costs such as scans, Haversine distance computations, and point-in-polygon tests are present. Figure 1 shows the average running time of a range query using adaptive-grid on a Tweets dataset, which consists of 83 million records (Section 3.2 provides more details about the dataset), with and without learning. As shown in the figure, for a low-selectivity query (selecting 0.00001% of the data, i.e., 8 records), the index and boundary refinement times dominate. In contrast, for a high-selectivity query (selecting 0.1% of the data, i.e., 83 thousand records), the scan time dominates.

Additionally, our study found that one-dimensional grid partitioning techniques (e.g., fixed-grid) benefit more from the use of learned models than two-dimensional techniques

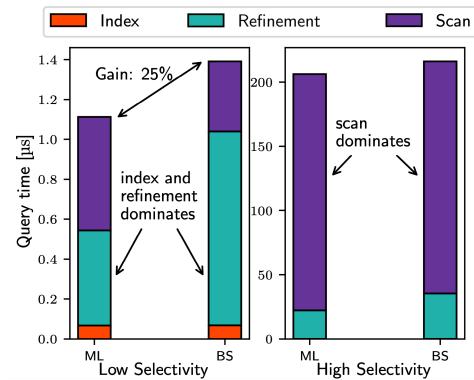


Fig. 1: Machine Learning vs. Binary Search (Spatial Range Query). For low selectivity (0.00001%), the index and boundary refinement phases dominate. For high selectivity (0.1%), the scan phase dominates. Parameters are tuned to favor Binary Search.

⁸ We adopt the definition of “selectivity” used by Pat Selinger et al. [Se79]. Therefore, low selectivity indicates that the result set of a query has few qualifying tuples, while high selectivity indicates the opposite.

(e.g., Quadtree). We have also discovered that, contrary to conventional wisdom, grid-based indexes, which filter on one dimension and index on the other, are *consistently* faster than tree-based indexes. This is because grid-based indexes typically have very large partitions for optimal performance, allowing fast searches based on learned models within each partition. This advantage might not extend to disk-based index structures due to the confinement of partition size by page dimensions. We also note that another advantage of grid-based indexes is that they are the *simpler* to implement.

In this paper, we extend the work presented in our previous publication [Pa20a]. In that previous work, we showed *preliminary* results *only for range queries* using three datasets and five learned indexes. In this study, we expand on our previous research by adding *three more query types* (i.e., point, distance, and spatial join), *a new learning-enhanced index* based on the linearization using the Hilbert curve, and *two competitive methods* commonly used in various applications and systems [Pa20b, Pa21] (i.e., JTS STRtree and S2PointIndex). Our study highlights the effectiveness of different spatial index structures when combined with learned models. By evaluating multiple query types and index structures, we aim to guide researchers and practitioners in selecting the best approach for their needs. Furthermore, our findings contribute to the design of improved spatial indexing methods using learned models.

Outline. The remainder of this paper is structured as follows. Section 2 presents the spatial indexing techniques that we implemented in our work, their learned variants, as well as the implementation of the different query types. Then, Section 3 presents our experimental study. Finally, we discuss related work in Section 4 before concluding in Section 5.

2 Approach

In this section, we describe the spatial indexing techniques we implemented (Section 2.1), explain how we built the learned indexes (Section 2.2), and detail the implementation of each query type (range, point, distance, and spatial join) in Sections 2.3- 2.6.

2.1 Indexing Techniques

Multidimensional access methods are classified into Point Access Methods (PAMs) and Spatial Access Methods (SAMs) [GG98]. PAMs handle point data without spatial extent, while SAMs handle extended objects such as linestrings and polygons. This work focuses on PAMs. *Spatial partitioning* splits a spatial dataset into partitions, or cells, where objects within the same partition are close in space. There are two types of spatial partitioning: space partitioning, which divides the embedded space, and data partitioning, which divides the data space. In this work, we use three spatial partitioning techniques: linearization using the Hilbert curve, fixed-grid [BF79], and Quadtree [FB74], and three data partitioning techniques: adaptive-grid [NHS84], K-d tree [Be75], and Sort-Tile-Recursive (STR) [LEL97]. Figure 2 illustrates these techniques on a sample of the Tweets dataset used in our experiments (cf. Section 3.2), with points and partition boundaries shown as dots and grid axes, respectively.

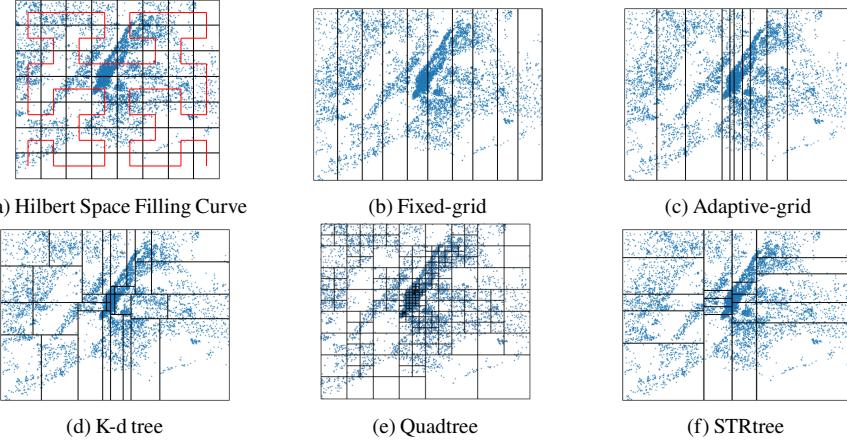


Fig. 2: An illustration of the different partitioning techniques.

2.1.1 Linearization using Hilbert Curve

Applying learned models to a spatial (multidimensional) index is challenging due to the lack of inherent sort order. We address this using the Hilbert space-filling curve (SFC) [Hi35] to map the multidimensional space to one dimension. As shown in Figure 2(a), linearization with the Hilbert SFC involves dividing the two-dimensional space into a uniform grid and using the Hilbert curve to enumerate the grid’s cells. Once enumerated, we can sort their identifiers and learn an index on this sorted order. This approach is similar to the recently proposed Z-order Model (ZM) index [Wa19], which uses the Z-curve for cell enumeration. We chose the Hilbert curve as it performs better for multi-dimensional indexing [LK01, LK00, Mo01, Dat21]. However, we show that linearization-based techniques can suffer from skewed cases, where queries cover a large portion of the curve, as shown in Section 3.4.2. For example, if a query rectangle covers all partitions at the bottom of Figure 2(a), the curve would lie entirely within the query, causing many irrelevant points to be scanned and leading to poor performance.

2.1.2 Fixed and Adaptive Grid

Grid-based indexing optimizes record retrieval by dividing the d-dimensional attribute space into cells, each pointing to a data page (or bucket). Data points that fall within the boundaries of a cell are stored on the corresponding data page, allowing quick navigation to the specific data page containing the desired records, rather than having to search through the entire dataset. The fixed-grid [BF79] enforces equidistant grid lines, while the adaptive-grid (or grid file [NHS84]) relaxes this constraint. Instead, to define the partition boundaries of the d-dimensions, the adaptive-grid introduces an auxiliary data structure containing a set of d-dimensional arrays called linear scales. In our implementation, we divide the space along one dimension and use the other dimension as the sort dimension. We also note that grid-based indexes are the *simplest* to implement since they only require maintaining a vector of grid lines and computing the intersection between the vector and a given query using offset computation and binary search for fixed-grid and adaptive-grid, respectively.

2.1.3 K-d tree

The K-d tree [Be75] is a binary search tree that recursively subdivides space into equal subspaces using rectilinear (or iso-oriented) hyperplanes. The splitting hyperplanes, called discriminators, alternate between the k dimensions at each tree level. In a 2-dimensional space, the splitting hyperplanes alternate between being perpendicular to the x- and y-axes. The original K-d tree splits *space* into equal halves but becomes unbalanced with skewed data. To ensure balanced partitions, we can instead use a data-aware approach, dividing the data at each level based on the median point. We implemented this data-aware K-d tree in our work.

2.1.4 Quadtree

The Quadtree [FB74] partitions the space similarly to the K-d tree but is not binary. For d dimensions, internal nodes have 2^d children. In 2D, each internal node has four children representing rectangles. The search space is recursively divided into four quadrants until each contains fewer objects than a predefined threshold, typically the page size. Quadtrees are generally not balanced, as the tree goes deeper in areas of higher density.

2.1.5 Sort-Tile-Recursive packed R-tree

An R-tree [Gu84] is a hierarchical data structure for efficient range queries. It approximates geometric objects with minimum bounding rectangles (MBRs). Each node stores up to N entries containing a rectangle R and a pointer P . R is the MBR of an object (at the leaf level) or a subtree (in internal nodes), and P points to that object or subtree.

The Sort-Tile-Recursive (STR) packing algorithm [LEL97] efficiently fills R-trees by tiling the data space into an $S \times S$ grid, where $S = \sqrt{P/N}$ (with P as the number of points and N the node capacity). It first sorts data by the x-dimension, divides it into S vertical slices, then sorts within each slice by the y-dimension, and packs nodes in runs of length N . This process continues recursively, filling all nodes except the last, which may have fewer than N elements.

2.2 Index Building

In this section, we outline how we can turn the above indexing techniques into learned indexes for a location dataset D containing points in latitude/longitude format (referred to as the x- and y-dimensions, respectively, for ease of understanding).

First, we partition D using one of the techniques described in Section 2.1 into partitions of size l points. We then sort the points within each partition on the y-dimension and build a learned index on the y-dimension for each partition. Algorithm 1 outlines the index building process.

Algorithm 1: Generic method for building learning-enhanced indexes

Input : D : the input location dataset; l : the partition size
Output : D' : the partitioned and indexed input dataset

```
1  $D' \leftarrow \{\}$ 
2  $P \leftarrow \text{Partition}(\text{some approach from the techniques described in Section 2.1}, l)$ 
3 for  $p \in P$  do
4   | Sort( $p, y$ )
5   | BuildLearnedIndex( $p, y$ )
6   |  $D' \leftarrow D' \cup \{p\}$ 
7 end
8 return  $D'$ 
```

2.3 Range Query Processing

A two-dimensional range query takes as input a query range q with a lower $((q_{xl}, q_{yl}))$ and an upper $((q_{xh}, q_{yh}))$ bound in both dimensions and a location dataset D , containing two-dimensional points represented by (p_x, p_y) . It returns all points in D contained in the query range q . Formally:

$$\text{Range}(q, D) = \{ p | p \in D : (q_{xl} \leq p_x) \wedge (q_{yl} \leq p_y) \wedge (q_{xh} \geq p_x) \wedge (q_{yh} \geq p_y) \}.$$

To accelerate query processing, we use the partitioned and indexed input dataset D' generated by Algorithm 1. Given D' , range query processing works in three phases, as shown in Algorithm 2.

Phase I: Index Lookup. The index lookup phase identifies partitions that intersect with the given range query using the index directory, i.e., the grid directories or trees. These intersected partitions, denoted as IP , are shown in line 2 of Algorithm 2. Note that the specific method used for this step depends on the partitioning technique.

Phase II: Boundary Refinement. After identifying intersected partitions, the next step is to locate query bounds on the sorted dimension within each partition. If a partition is fully within the query range, all its points are returned immediately (Algorithm 2, line 6). For partial intersections, there are two cases: (1) If the partition is fully within the x-dimension range, we employ a search technique to compute the lower and upper bounds and then copy all points within these bounds (lines 8–12 of the algorithm). (2) If the partition is not fully within the x-dimension range, we compute the lower and upper bounds on the sorted y-dimension, then switch to the scan phase.

Typically, binary search is used as the search technique. In this paper, we propose replacing it with a learned model, specifically the RadixSpline index [Ki20b, Ki20c], to search the sorted dimension more efficiently. RadixSpline has two components: spline points and a radix

Algorithm 2: Range Query Algorithm

```

Input   :  $D'$ : a partitioned and indexed input dataset;  $q$ : a query range
Output  :  $RQ$ : a set of all points in  $D'$  within  $q$ 

1  $RQ \leftarrow \{\}$ 
  /* find intersected partitions (IP) */
2  $IP \leftarrow \text{IndexLookup}(D', q)$ 
3 for  $ip \in IP$  do
  /* if completely inside x-dim. range */
  /*
  4 if  $q_{xl} \leq ip_{xl}$  and  $ip_{xh} \leq q_{xh}$  then
    /* if completely inside y-dim. range, copy entire partition */
    /*
    5 if  $q_{yl} \leq ip_{yl}$  and  $ip_{yh} \leq q_{yh}$  then
      /* copy points in partition */
      /*
      6  $RQ \leftarrow RQ \cup ip$ 
    else
      /* lower bound */
      7  $lb \leftarrow \text{EstimateFrom}(ip, q_{yl})$ 
      /* get exact lower bound */
      8  $lb \leftarrow \text{LocalSearch}(ip, lb, q_{yl})$ 
      /* upper bound */
      9  $ub \leftarrow \text{EstimateTo}(ip, q_{yh})$ 
      /* get exact upper bound */
     10  $ub \leftarrow \text{LocalSearch}(ip, ub, q_{yh})$ 
      /* copy points between lower and upper bound */
     11  $RQ \leftarrow RQ \cup ip.\text{range}(lb, ub)$ 
    end
  else
    /* lower bound */
    12  $lb \leftarrow \text{EstimateFrom}(ip, q_{yl})$ 
    13  $lb \leftarrow \text{SearchPoint}(ip, lb, q_{yl})$ 
    /* upper bound */
    14  $ub \leftarrow \text{EstimateTo}(ip, q_{yh})$ 
    15  $ub \leftarrow \text{LocalSearch}(ip, ub, q_{yh})$ 
    /* scan */
    16 for  $i \in [lb, ub]$  do
      /*  $i$ th point in partition  $ip$  */
      /*
      17  $p \leftarrow ip_i$ 
      if  $p$  within  $q$  then
        18  $RQ \leftarrow RQ \cup \{p\}$ 
      end
    end
  end
end
return  $RQ$ 

```

table. The radix table quickly identifies the spline points for a given lookup key (in our case, the sorted dimension). At lookup time, the radix table is first used to determine the range of spline points. Then, these spline points are searched to find those surrounding the lookup key, and linear interpolation is applied to predict the lookup key’s position in the sorted array.

Given the inherent error introduced by the RadixSpline (and generally, learned indexes), a local search (*LocalSearch()* in Algorithm 2) is needed to find the exact query bound. Without loss of generality, we describe the local search procedure for the computation of the lower query bound. For range scans, there are two cases: (1) If the estimated value is lower than the true lower bound, scan upward to the lower bound. (2) If the estimated value is higher, scan downward to the lower bound, materializing all encountered points. This search incurs no extra materialization costs unless the estimate exceeds the query’s upper bound, as points within query bounds are materialized anyway.

Phase III: Scan. When the partition partially intersects the x-dimension range, then after determining the bounds of the query on the sorted dimension in the boundary refinement phase, the final step is to scan the partition to find the qualifying points on the x-dimension. During this scan phase, we iterate through the partition starting from the determined lower bound and continue until we reach either the upper bound of the query on the sorted y-dimension or the end of the partition. This process is reflected in Algorithm 2 from line 14 onward.

2.4 Point Query Processing

A point query takes a query point q_p and a set of geometric objects D as input. It returns true if q_p is found within D , and false if it is not. Formally:

$$\text{Point}(q_p, D) = \exists p \in D. q_p.x = p.x \wedge q_p.y = p.y.$$

We use the partitioned and indexed dataset D' from Algorithm 1 to speed up point queries, as outlined in Algorithm 3. First, we perform *IndexLookup()* using a degenerate rectangle from query point q_p . If no intersecting partition is found, we return false. Otherwise, we search the partition in two steps: (i) estimate the point’s location in the y-dimension using the learned search method, and (ii) refine the result with *SearchPoint()* to correct errors introduced by the learned search technique, similar to *LocalSearch()* in range query processing.

For the RadixSpline, there are three cases for the *SearchPoint()* procedure. First (second), if the estimated value is lower (higher) than the lower (upper) bound on the sorted dimension, we scan upward (downward) and compare the elements on the sorted dimension until reaching the lower (upper) bound. We then continue scanning and comparing elements on both dimensions until finding the query point or reaching the partition’s upper (lower) bound. Third, if the estimated value matches the query point’s value on the search dimension, we scan upward, comparing on both dimensions, to locate the query point. If the partition’s upper bound is reached without finding the point, we then scan downward, again comparing on both dimensions, until we locate the query point or reach the partition’s lower bound.

Algorithm 3: Point Query

Input : D' : a partitioned and indexed input dataset; q_p : a query point
Output :*true* if the point q_p is in D' , *false* otherwise

```

/* find intersected partition (IP) */
1 IP ← IndexLookup( $D'$ ,  $q_p$ )
/* search within the partition */
2 if  $IP \neq \emptyset$  then
    /* get estimate */
    3   est ← EstimateFrom(IP,  $q_p.y$ )
    4   found ← SearchPoint(ip, est,  $q_p$ )
    5   return found
6 else
7   return false
8 end

```

Algorithm 4: Distance Query

Input : D' : partitioned and indexed input dataset; q_p : query point; d : distance
Output : DQ : set of points in D' within distance d of q_p

```

1  $DQ \leftarrow \{\}$ 
/* Get the circle's MBR */
2 mbr ← GetMBR( $q_p, d$ )
/* Filter using mbr */
3 RQ ← RangeQuery( $D'$ , mbr)
/* Refine */
4 for  $p \in RQ$  do
5   if WithinDistance( $p, q_p, d$ ) then
6     |  $DQ \leftarrow DQ \cup \{p\}$ 
7   end
8 end
9 return  $DQ$ 

```

In the case of a binary search, the process is simpler. We first use the value of the query point on the sorted dimension to find the lower bound. Then, we scan upward, comparing on both dimensions, until we find the query point.

2.5 Distance Query Processing

A distance query takes a query point q_p , a distance d , and a set of geometric objects D . It returns all objects in D that lie within the distance d from the query point q_p . Formally:

$$Distance(q_p, d, D) = \{ p | p \in D \wedge dist(q_p, p) \leq d \}.$$

As in the case of Range query processing (Section 2.3), we use the partitioned and indexed input dataset D' from Algorithm 1 for faster query processing. The implementation of the distance query employs the *filter and refine* [Or89] approach, which is commonly used in state-of-the-art research where various spatial queries (e.g., kNN, distance, and join queries) are first decomposed to range queries as a preliminary filter, followed by query-specific refinement [Qi20, Li20, Gu23, Li23a, YC23] as well as popular database systems such as Oracle Spatial [KRA02].

Algorithm 5: Join Query

Input : D' : partitioned and indexed input dataset; $polygons$: a set of polygons
Output : JQ : a set of sets, a set of points within each polygon in $polygons$

```

1  $JQ \leftarrow \{\}$ 
2 for  $polygon \in polygons$  do
    /* Get minimum bounding rectangle (mbr) of the polygon */
    3   mbr ← GetMBR( $polygon$ )
    /* Filter using MBR */
    4   RQ ← RangeQuery( $D'$ , mbr)
    contained ← {}
    /* Refine */
    5   for  $p \in RQ$  do
    6     if Contains( $polygon, p$ ) then
    7       | contained ←
    8         | contained  $\cup \{p\}$ 
    9     end
    10   end
    11    $JQ \leftarrow JQ \cup \{contained\}$ 
12 end
13 return  $JQ$ 

```

Algorithm 4 shows the algorithm for distance query processing. We first filter using a rectangle (reflected in line 1 of Algorithm 4), whose corner vertices are at a distance of d from the query point q . We issue a range query using this rectangle, and then refine the resulting candidate set of points using a *withinDistance* predicate. Note that we are using GPS coordinates (i.e., a Geographic coordinate system). Therefore, special attention must be given if either of the poles or the 180th meridian is within the query distance d . We compute the coordinates of the minimum bounding rectangle by moving along the geodesic arc as described in [BS13] and then handle the edge cases of the poles and the 180th meridian⁹.

2.6 Join Query Processing

A spatial join combines two input spatial datasets, R and S , using a specified join predicate θ (such as overlap, intersect, contains, within, or withindistance). It returns a set of pairs (r,s) where $r \in R, s \in S$ that meet the join predicate θ . Formally:

$$R \bowtie_{\theta} S = \{ (r,s) | r \in R, s \in S, \theta(r,s) \text{ holds} \}.$$

We implemented a join query between a set of polygons and the partitioned and indexed input location dataset D' . The join algorithm is outlined in Algorithm 5 and is based on the *filter and refine* [Or89] approach.

This involves using the minimum bounding rectangle of each polygon to perform a range query. We then refine the candidate set of points using *contains* as the predicate θ , thus computing all points contained in each polygon. We implemented the contains predicate using the ray-casting algorithm, where a ray is cast from the candidate point to a point outside the polygon, and then the number of intersections with polygon edges is counted. Some polygons could potentially contain hundreds or thousands of edges. Therefore, to facilitate a quick lookup of the edges intersected with the ray, we index the polygon edges in an interval tree. We implemented the interval tree using a binary search tree. Future work will explore the indexing of polygons with learned structures [Tz21, WY22].

3 Evaluation

3.1 Experimental Setup

Hardware Configuration. Experiments were run single-threaded on an Ubuntu 18.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 10 cores, 3.00 GHz turbo)¹⁰ and 256 GB DDR3 RAM. To avoid NUMA effects, we use the *numactl* command to bind the thread and memory to one node. CPU scaling was also disabled using the *cpupower* command.

⁹ We currently use only one bounding box. This approach is not optimal, as it can result in materializing a large number of unnecessary points when the 180th meridian falls within the query distance. To improve efficiency, we could break the bounding box into two parts, one on either side of the 180th meridian. We leave this optimization for future work. Furthermore, our query workload does not include these edge cases.

¹⁰ CPU: <https://ark.intel.com/content/www/us/en/ark/products/75272/intel-xeon-processor-e5-2660-v2-25m-cache-2-20-ghz.html>

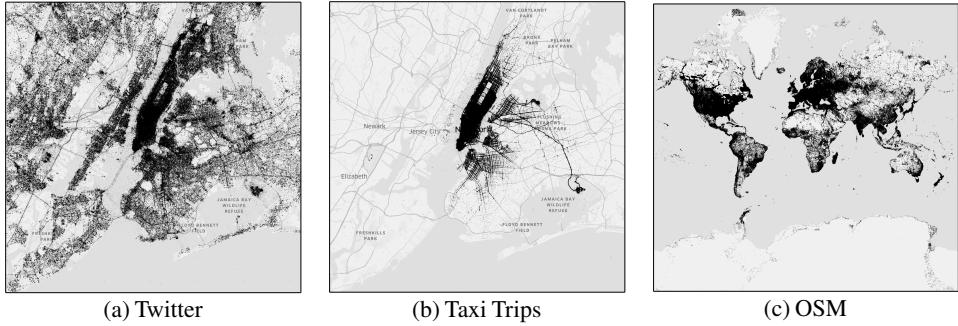


Fig. 3: Datasets: (a) Tweets are spread across New York, (b) NYC Taxi trips are clustered in central New York, and (c) All Nodes dataset from OSM.

Software Configuration. In all our experiments, we sort on the longitude value of the location within each partition. The currently available open-source implementation of RadixSpline only supports integer values. However, most spatial datasets contain floating-point values. To address this issue, we adapted the RadixSpline implementation to work with floating-point values. We set the spline error to 32 for all experiments in our RadixSpline implementation. Furthermore, we do not address updates in this work, but focus only on *read-only* datasets.

3.2 Datasets and Queries

For evaluation, we used three datasets, the New York City Taxi Rides dataset [NYC19] (NYC Taxi Rides), geo-tagged tweets in the New York City area (NYC Tweets), and Open Streets Maps (OSM). NYC Taxi Rides contains 305 million taxi rides from 2014 and 2015. NYC Tweets data was collected using Twitter’s Developer API [Twe20] and contains 83 million tweets. The OSM dataset is taken from [Pa18] and contains 200M records from the All Nodes (Points) dataset. Figure 3 shows the spatial distribution of the three datasets. We further generated two types of query workloads for each of the three datasets: skewed queries, which follow the distribution of the underlying data, and uniform queries. For each type of query workload, we generated six different workloads ranging from 0.00001% to 1.0% selectivity. For example, for the Taxi Rides dataset (305M records), these queries would materialize from 30 to 3 million records. The query workloads consist of one million queries each. To generate skewed queries, we select a record from the data and expand its boundaries (using a random ratio in both dimensions) until the selectivity requirement of the query is met. For uniform queries, we generate points uniformly in the embedding space of the dataset and expand the boundaries similarly until the selectivity requirement of the query is met. The query selectivity and the type of query are mostly application-dependent. For example, consider a user issuing a query to find a popular pizzeria nearby on Google Maps. The expected output for this query should be a handful of records, i.e., the query selectivity is low (a list of 20-30 restaurants near the user). On the other hand, a query on an analytical system would materialize many more records (e.g., find the average cost of all taxi rides originating in Manhattan).

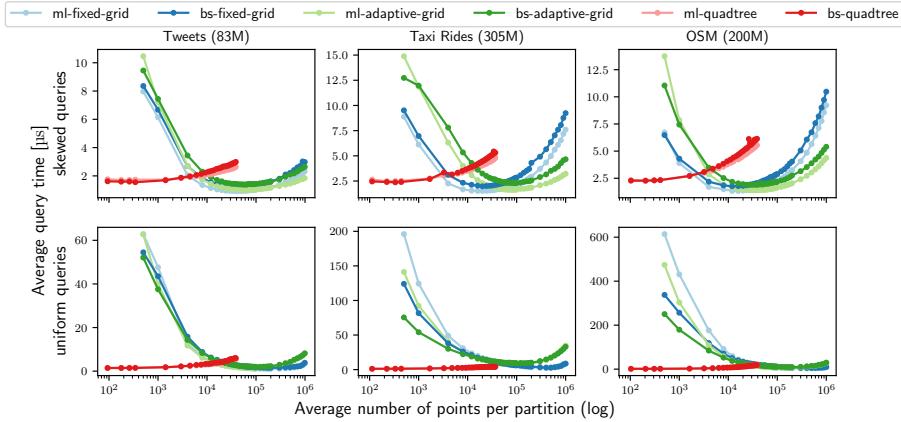


Fig. 4: Range Query Configuration - ML vs. BS for low selectivity (0.00001%).

3.3 Baselines

All our learning-enhanced indexes are implemented in C++. We evaluate their performance for search within partitions against binary search. Furthermore, we compare our learning-enhanced indexes with the two best-performing indexes from prior studies [Pa20b, Pa21], which compared state-of-the-art spatial libraries. More specifically, for range and distance queries, we compare our implementation with the STRtree implementation from the Java Topology Suite (JTS) and the S2PointIndex from Google S2. For join queries, we use the S2ShapeIndex provided by Google S2. JTS is written in Java, whereas Google S2 is implemented in C++. The source code used in this work is available on GitHub¹¹.

3.4 Range Query Performance

In this section, we first explore the tuning of partition sizes and why the tuning is crucial to obtain optimal performance. Next, we present the total query runtime when the partition size for each index is tuned for optimal performance.

3.4.1 Tuning Indexing Techniques

Recent work in learned multidimensional and spatial indexes focuses on instance-optimizing based on data and query workload [Kr21, Di21]. To study this effect, we conducted multiple experiments on the three datasets by varying the sizes of the partitions, tuning them on two workloads with different selectivities, for both skewed and uniform range queries. We omit the results for tuning the indexing techniques for other queries (point, distance, and join) as they are similar to the ones for range queries.

Figure 4 shows the impact of tuning for the lowest selectivity workload across two query types. It can be seen that tuning the grid indexing techniques to the workload is crucial, as they

¹¹ <https://github.com/varpande/learnedspatial>

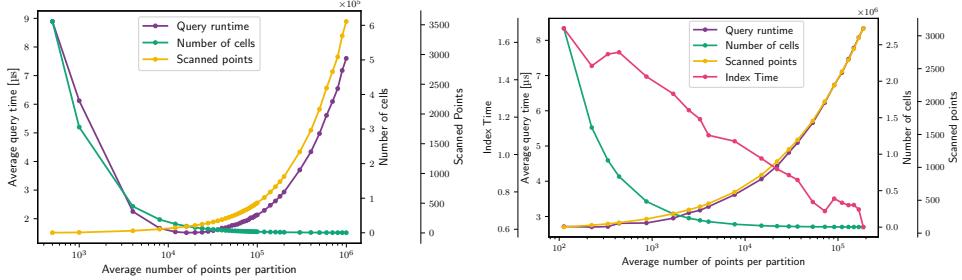


Fig. 5: Effect of the number of cells and scanned points for fixed-grid on Taxi Trip dataset for skewed queries (0.00001% selectivity).

Fig. 6: Effect of the number of cells and scanned points for Quadtree on Taxi Trip dataset for skewed queries (0.00001% selectivity).

Selectivity (%)	Taxi Trips (Skewed Queries)						Taxi Trips (Uniform Queries)					
	Fixed		Adaptive		Quadtree		Fixed		Adaptive		Quadtree	
	ML	BS	ML	BS	ML	BS	ML	BS	ML	BS	ML	BS
0.00001	1.78	2.35	1.86	2.40	2.77	2.51	2.02	2.58	81.4	10.54	1.48	1.31
0.0001	4.54	5.82	4.67	6.12	6.12	5.82	5.85	6.91	228.1	27.69	3.69	3.42
0.001	14.97	18.83	15.32	19.49	20.84	19.47	22.87	24.34	708.8	87.49	13.59	12.98
0.01	90.13	97.04	89.48	95.96	117.01	104.37	141.24	151.47	2634.4	309.62	98.85	112.77
0.1	678.12	698.39	675.14	696.49	922.67	793.96	988.35	922.96	9609.9	1174.79	891.24	1101.95
1.0	8333.94	8408.15	8301.56	8399.69	10678.04	9512.29	8843.71	8753.68	8574.84	8836.28	10647.97	12377.14

Tab. 1: Total query runtime (in microseconds) for RadixSpline (ML) and binary search (BS) for Taxi Rides dataset on skewed and uniform query workloads (parameters tuned for selectivity 0.00001%).

are *highly* sensitive to partition size. Performance improves as partition size increases, reaching an optimal point, after which further increases degrade performance. For instance, using a fixed-grid with 100 points per partition (a common default in many spatial libraries) leads to up to 300× worse performance compared to the optimal size. For grid (single-dimension) indexing techniques, optimal partition sizes are much larger than for multi-dimensional indexing techniques (only Quadtree is shown in the figure but the same holds for the other indexing techniques covered in this work). This results in significant performance gains when using learned indexes, especially for skewed queries, with speedups ranging from 11.79% to 39.51% over binary search. Even when we tuned a learned index to a partition size that corresponds to the optimal performance for binary search, we found that the learned index frequently outperformed the binary search. However, learned indexes offer little benefit for techniques that filter on both dimensions (cf. Table 1), such as Quadtree and STRtree, where optimal partition sizes are low (fewer than 1,000 points). In such cases, refinement costs become an overhead. In contrast, the k-d tree, with optimal partition sizes ranging from 1,200 to 7,400 points for the Taxi Trips and OSM datasets, sees a performance boost of 2.43% to 9.17% with learned indexes. For the Twitter dataset, where the optimal partition size is under 1,200 points, we observed a similar drop in performance with learned indexes.

Figure 5 shows how the number of cells and points scanned in each partition impact the query runtime for fixed-grid on Taxi Trips dataset with the lowest selectivity. As the number of points per partition increases (i.e., there are fewer partitions), the number of cells decreases, but more points need to be scanned. The point where these curves meet is the

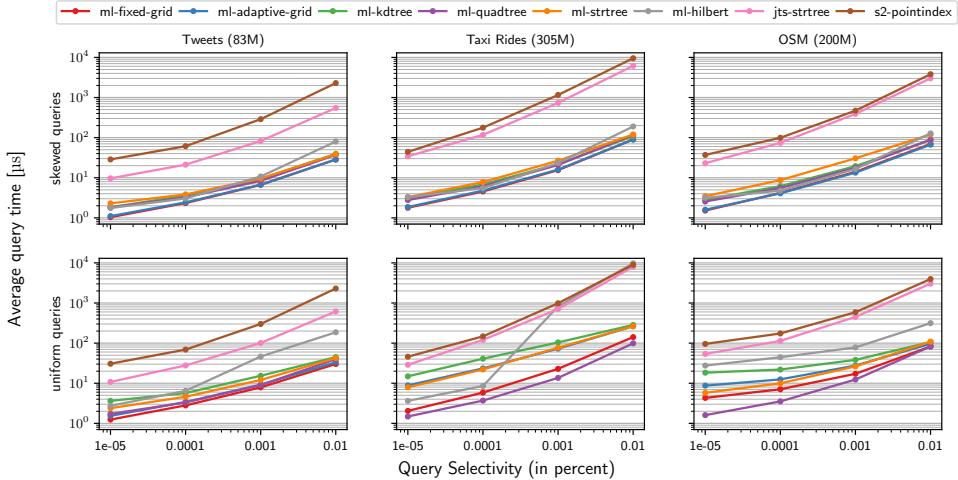


Fig. 7: Range Query Runtime on skewed and uniform queries for the three datasets.

optimal configuration for the workload, where query runtime is lowest. In contrast, for tree structures, as shown in Figure 6, most pruning happens during the index lookup, with the dominant cost being the number of points scanned per partition. To reduce this, tree structures require more partitions (fewer points per partition). They pay a higher cost during the index lookup phase due to random access and cache misses but scan fewer points once the target partition is found, as most partitions qualify for the query.

Key Takeaways. Tuning partition sizes is critical for grid-based indexing, where larger partitions enable faster searches with learned models. In contrast, tree-based indexes see less benefit from learned models due to their smaller optimal partition sizes.

3.4.2 Query Performance

Figure 7 shows the query runtime for all learned index structures. It can be seen that fixed-grid along with adaptive-grid (1D schemes) perform the best for all cases except for uniform queries on Taxi and OSM datasets.

Indexing	Taxi Rides		OSM	
	Skewed	Uniform	Skewed	Uniform
Fixed	1.97	7.98	1.72	23.73
Adaptive	1.74	31.57	1.51	24.80
k-d tree	1.70	21.62	1.56	30.95
Quadtree	1.79	2.12	1.37	7.96
STR	2.60	47.03	1.90	11.05

Tab. 2: Average number of partitions intersected for each indexing method for selectivity 0.00001% on Taxi Rides and OSM datasets.

For skewed queries, fixed-grid is 1.23 \times to 1.83 \times faster than the closest competitor, Quadtree (2D), across all datasets and selectivity. The slight difference in performance between fixed-grid and adaptive-grid comes from the index lookup. For adaptive-grid, we use binary search on the linear

scales to find the first partition the query intersects with. For fixed-grid, the index lookup is almost negligible as only an offset computation is needed to find the first intersecting partition. This is also in contrast to traditional knowledge that grid-based index structures can become skewed and thus perform worse than tree-based index structures. Since the index structures and data reside in memory and are tuned to optimal partition size, the grid-based structures perform better as (1) they avoid pointer chasing as in the case of tree-based index structures, thus leading to fewer random accesses, and (2) they can utilize the *fast lookups* using learned models within the large indexed partitions. This would not be possible for disk-based index structures. Note that partition sizes for optimal performance of grid-based index structures are very large for every datasets. For disk-based index structures to exhibit similar performance, it would require allocating very large pages on disk.

It can also be seen in the figure that the Quadtree is significantly better for uniform queries in the case of the Taxi Rides dataset ($1.37\times$) and OSM dataset ($2.68\times$) than the closest competitor, fixed-grid. There are two reasons for this. First, as Table 2 shows, the Quadtree intersects with fewer partitions than the other index structures. Second, for uniform queries, the Quadtree is more likely to traverse the sparse and low-depth region of the index. This is consistent with previously reported findings [KP07].

In Figure 7, we can also see the performance of the learned indexes compared to JTS STRtree and S2PointIndex. Fixed-grid is from $8.67\times$ to $43.27\times$ faster than the JTS STRtree and from $24.34\times$ to $53.34\times$ faster than S2PointIndex. Quadtree, on the other hand, is from $6.26\times$ to $33.99\times$ faster than JTS STRtree, and from $17.53\times$ to $41.91\times$ faster than S2PointIndex. Note that the index structures in the libraries are used with their default, out-of-the-box settings and are not tuned. S2PointIndex performs poorly because it operates on Hilbert curve values and is not optimized for range queries. S2PointIndex and the learned linearized Hilbert curve index counterpart are rather similar as they both apply linearization to one dimension for indexing. The learned counterpart is learned on the sorted values of the linearized values where the underlying implementation is a densely packed array, while S2PointIndex stores these linearized values in a main-memory optimized B-tree. S2PointIndex is a B-tree on the 64-bit integers called S2CellId. The cell ids are a result of the Hilbert curve enumeration of a *Quadtree*-like space decomposition. Hilbert curve (as previously mentioned in Section 2.1.1) suffers from skewed cases where the range query rectangle covers the whole curve. Our Hilbert curve implementation uses the MBR's extent to compute the min and max Hilbert values to filter points leading to scanning many unnecessary points. To minimize the effect of such cases, S2PointIndex divides the query rectangle into four parts. [Or89] first proposed a similar approach which computes the Z-curve decomposition of the query object, a computationally extensive operation, to minimize number of pages fetched from disk in the filter phase. The paper highlights that there is a trade-off between decomposition cost and filtering efficiency.

Key Takeaways. Fixed-grid excels in most queries by avoiding random pointer chasing and using fast learning-enhanced search within its large partitions. Quadtree outperforms fixed-grid for uniform queries on Taxi and OSM, as these queries intersect fewer partitions and only traverse the sparse, low-depth region of the index. The linearized Hilbert curve generally underperforms, as queries require scanning a large portion of the curve.

3.5 Point Query Performance

Section 2.4 defines how the point query has been implemented in this work. JTS STRtree does not provide a way to search for a point and thus we did not implement the point query using JTS STRtree. On the other hand, S2PointIndex in the Google S2 library allows querying for a point. Moreover, we run the point queries using only the skewed queries workload, which uniformly selects a random point from the dataset itself. This ensures that the point query actually produces a result and does not skew the results in favor of the learned indexes. It also aligns with real-world workloads, where searching for existing points in a dataset—such as retrieving metadata for a specific restaurant—is more common.

Figure 8 shows the point query runtime for all indexing techniques. Fixed-grid is again the best-performing index for point queries on skewed workloads. It is $1.94\times$, $2.27\times$, and $1.51\times$ faster than the closest tree-based competitor, kd-tree, across Tweets, Rides, and OSM datasets. However, the performance difference of fixed-grid with adaptive-grid are marginal. It is $1.37\times$, $1.1\times$, $1.04\times$ faster than the adaptive-grid across the Tweets, Rides, and OSM datasets. Lastly, fixed-grid is also $2.44\times$, $2.56\times$, $2.79\times$ faster than S2PointIndex. Another very important observation is that the learned index on the linearized values is very competitive in the point queries. This is counter-intuitive from the observation in range queries in Section 3.4.2. We noted that range searches on sorted Hilbert curve values perform poorly in skewed cases where the query rectangle covers a large portion of the curve. However, for point queries, only one point on the curve needs to be searched, rather than scanning multiple points. This makes the learned index on the linearized values very competitive for point queries. In fact, in the case of the OSM dataset, it is the best-performing index with an average query time of 385ns compared to 390ns for the fixed-grid (the best-performing index in the other two datasets).

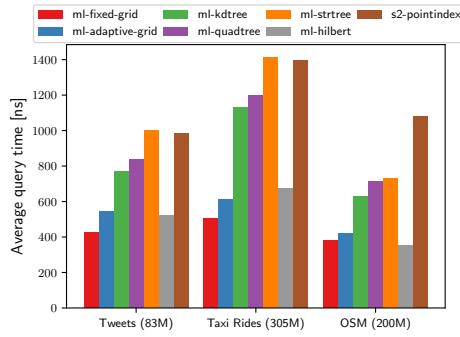


Fig. 8: **Point Query Performance** for skewed queries on the three datasets.

Key Takeaways. Fixed-grid performs best across all queries. In contrast to range queries, the linearized Hilbert curve is highly competitive for point queries that require searching for a single point on the curve.

3.6 Distance Query Performance

We implemented the distance query using the filter and refine [Or89] approach (see Section 2.5), which is the norm in spatial databases such as Oracle Spatial [KRA02] and PostGIS [Pos23]. We index GPS coordinates and use the *Harvesine* distance in the refinement phase.

Figure 9 shows the distance query runtime for all indexing techniques as well as the two spatial indexes, S2PointIndex and JTS STRtree. We can make two important observations.

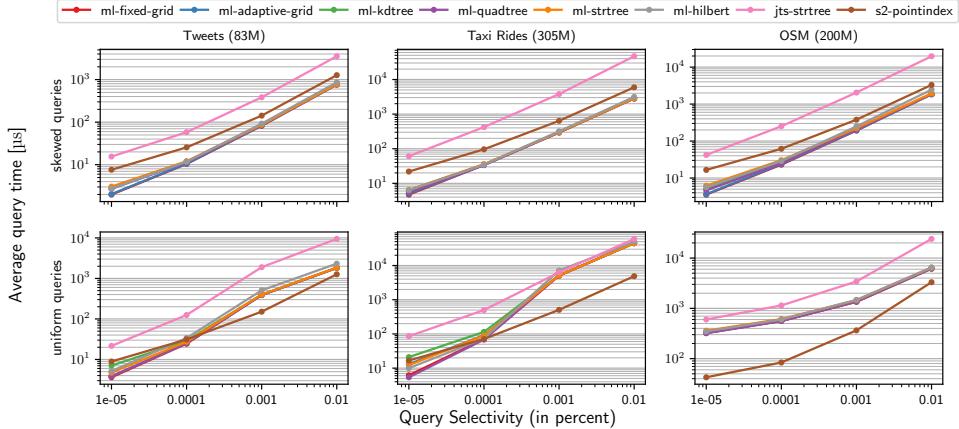


Fig. 9: **Distance Query Runtime** on skewed and uniform queries for the three datasets.

First, the difference in performance between the learned indexes diminishes quickly as we increase the selectivity of the query. Grid-based indexes perform the best for lower selectivities (0.00001% and 0.0001%), except for uniform queries on Taxi Rides and OSM datasets where Quadtree is better, similar to range query. However, as more points qualify in the filter phase, Haversine distance computation becomes the dominant cost, causing the performance of all indexing methods to converge. Haversine distance is computationally expensive and requires multiple additions, multiplications, and divisions as well as three trigonometric function calls. Although we only use Harvesine distance on a subset of points in the filter phase, it is still expensive to compute. The second observation is that S2PointIndex outperforms most indexes for uniform queries on the OSM dataset. The reason for this is that after the filter phase, many points need refinement for uniform queries for the OSM dataset. For example, for the OSM dataset, the average number of points that need refinement after the filter phase for skewed queries is 25, 257, and 2561 for selectivities 0.00001%, 0.0001%, and 0.001%, respectively. For uniform queries, the average number of points that need refinement after the filter phase is 4257, 7263, 17612 (6 \times to 170 \times more than skewed queries) for the OSM dataset. The dominant cost for most index structures is the Haversine distance computation, and thus we also do not observe much difference in performance between the learned indexes. S2PointIndex implements a variant¹² of the branch and bound [RKV95] algorithm for distance queries. The distance query is represented as an S2Cap, a region with a point and a radius. Its covering is computed, and intersected with that of the S2PointIndex. The resulting cells in the intersection are then added to a priority queue. The traversal algorithm for S2PointIndex then continues in a manner similar to the [RKV95] traversal method, except that it is constrained by distance rather than the k nearest objects. Therefore, the distance query using the S2PointIndex for uniform queries on the OSM dataset is from 1.91 \times to 7.75 \times faster than the learned indexes. This effect is not reflected in the other datasets, since after the filter phase the number of points that qualify for Haversine distance computation is similar to that for the skewed queries in the OSM dataset. The comparison of learned indexes with JTS STRtree is more fair since

¹² The corresponding code can be found at: https://github.com/google/s2geometry/blob/master/src/s2/s2closest_point_query_base.h#L456

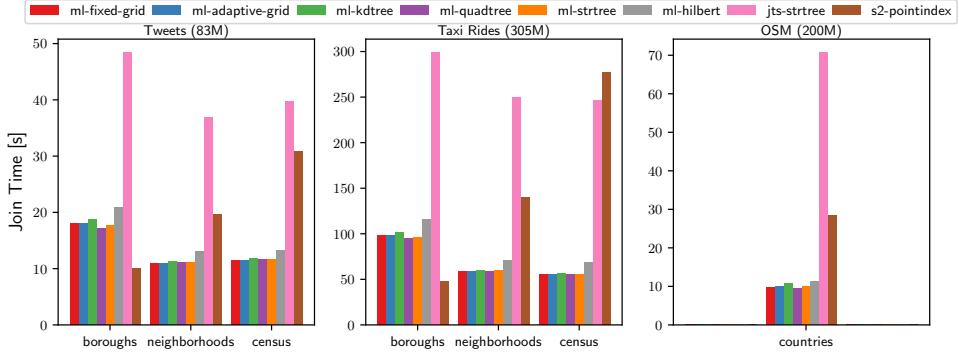


Fig. 10: **Join Query Performance** - Total join query runtime for the three datasets.

both the learned indexes and JTS STRtree employ the *filter and refine* approach to evaluate distance queries. Fixed-grid is from 1.33× to 11.92× faster than JTS STRtree.

Key Takeaways. Learned search boosts performance for lower selectivity and skewed queries but loses its advantage in higher selectivity or uniform queries due to the increased number of points for refinement (i.e., Haversine distance computations).

3.7 Join Query Performance

We evaluate join queries using the *filter and refine* approach for the learned indexes and JTS STRtree. We use the bounding box of the polygon objects and issue a range query on the indexed points, while in S2, we utilize the S2ShapeIndex which is specifically built to test for the containment of points in polygons. As mentioned in Section 2.6, we index the polygon objects using an interval tree in case of the learned indexes. For JTS STRtree, we use the PreparedGeometry¹³ abstraction to index line segments of all individual polygons, which helps accelerating the refinement check.

We joined three different polygonal datasets with the location datasets that are in the NYC area (i.e., Tweets and Taxi Rides datasets). Specifically, we used the Boroughs, Neighborhoods, and Census block boundaries consisting of five, 290, and 39192 polygons, respectively. These datasets have an average of 662.2, 29.55, and 12.58 edges per polygon, respectively, and their raw file sizes are 116KB, 321KB, and 20MB. For the OSM dataset, we perform the join using the Countries dataset which consists of 255 country boundaries, has an average of 24.72 edges per polygon, and the original size of the file is 17MB. Similarly to range and distance queries, we first find the optimal partition size for each learned index and dataset.

Figure 10 shows the join query performance, where we observe that most of the learned indexes have similar performance. The reason behind this is that the *filter* phase is not expensive for the join query, while the *refinement* phase is the dominant cost. This result

¹³ <https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/prep/PreparedGeometry.html>

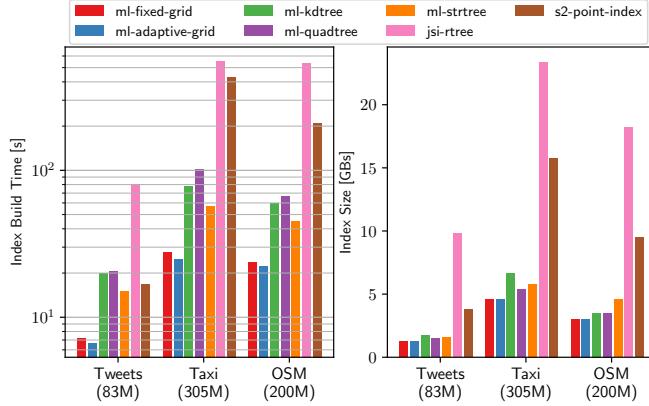


Fig. 11: **Indexing Costs** - Index build times and sizes for the three datasets.

is in conformance to earlier studies [Pa20b, Pa21], which compared state-of-the-art spatial libraries used by hundreds of systems and other libraries. Although we use an interval tree to index the edges of the polygons to quickly determine the edges intersecting the ray casted from the candidate point, this phase is still expensive. For future work, we plan to investigate the performance using the main-memory index for polygon objects proposed in [Ki20a].

Figure 10 also shows that the learned indexes significantly outperform both JTS STRtree and S2ShapeIndex in the join query. Fixed-grid, for example, is $1.81\times$ to $2.69\times$ faster than S2ShapeIndex and $2.7\times$ to $3.44\times$ faster than JTS STRtree for the Tweets dataset across all three polygonal datasets. Similarly, for the Taxi Rides dataset, fixed-grid is $2.39\times$ to $4.96\times$ faster than S2ShapeIndex and $3.017\times$ to $4.49\times$ faster than JTS STRtree. Lastly, for the OSM dataset, fixed-grid outperforms S2ShapeIndex by $2.89\times$ and JTS STRtree by $7.311\times$.

Key Takeaways. The *refinement* phase of the join query involves many costly point-in-polygon tests, negating the benefit of the fast *filter* phase.

3.8 Indexing Costs

Figure 11 shows that fixed-grid and adaptive-grid are faster to build than tree-based learned indexes. Fixed-grid is $2.11\times$, $2.05\times$, and $1.90\times$ faster to build than the closest competitor, STRtree. Quadtree is the slowest to build because it generates a large number of cells for optimal configuration. Not all partitions in Quadtree contain an equal number of points as it divides the space rather than the data, thus leading to an imbalanced number of points per partition. Fixed-grid and adaptive-grid do not generate a big number of partitions, as the partitions are quite large for optimal configuration. They are smaller for similar reasons. The index size in Figure 11 also includes the size of the indexed data.

In the figure, we can also see that the learned indexes are faster to build and consume less memory than the S2PointIndex and JTS STRtree. Fixed-grid, for example, is from $2.34\times$

to $15.36\times$ faster to build than S2PointIndex, and from $11.09\times$ to $19.74\times$ faster to build than JTS STRtree. It also consumes less memory than S2PointIndex (from $3.04\times$ to $3.4\times$) and JTS STRtree (from $4.96\times$ to $8.024\times$). However, we note that the comparison of the index size with JTS STRtree is not completely fair. JTS STRtree is a SAM (spatial access method), where it stores four coordinates for each point (since the points have been stored as degenerate rectangles). The learned indexes implemented in this work are PAMs (point access method), where we only store two coordinates for each data point.

Key Takeaways. Grid-based indexes are faster to build and use less space than tree-based ones. Optimally-tuned grid-based indexes have fewer, larger partitions, while tree-based indexes create many smaller ones. Embedding learned models in each partition increases tree-based indexes' build time and size, as they maintain more learned models.

4 Related Work

Work by Kraska et al. [Kr18] proposed replacing traditional database indexes with learned models. Since then, machine learning has been applied to various aspects of data management [Ki22, SUK22, Ya23, PMW22a, Sa22, SK23]. Furthermore, there has been a corpus of work on extending the concept of learned indexes to spatial and multidimensional data.

Learned Multidimensional Indexing and Partitioning. Flood [Na20] is an in-memory read-optimized multidimensional index that partitions d -dimensional data into grids over the d -dimensional space based on query workload and data distribution. Similarly to Flood, our grid index implementation partitions data across $d - 1$ dimensions and uses the last dimension for sorting. Tsunami [Di20a, Di20b] improves Flood for handling skewed and correlated data. Machine learning has also been applied to reduce I/O costs of disk-based multidimensional indexes. Qd-tree [Ya20] uses reinforcement learning to optimize partitioning, while RW-Tree [Do22], PLATON [YC23], ACR-Tree [HWL23] present learning-based approaches to effectively pack R-Trees. The ZM-index [Wa19] combines the standard Z-order space-filling curve with Recursive-Model Indexes (RMI) [Kr18]. BMT [Li23b], LMSFC [Ga23], and WAZI [PMW24] use space-filling curves to optimize the page layout based on data and workload. The ML-index [DMM20] combines the ideas of iDistance [Ja05] and RMI [Kr18] for range and kNN queries. There are also efforts to augment existing indexes with lightweight models to accelerate range and point queries [HKH20]. For a broader perspective, we refer the reader to a comprehensive survey [Ma24] and an experimental analysis study [Li24] on multi-dimensional indexing.

Learned Spatial Indexes and Algorithms. LISA [Li20] is a disk-based learned spatial index that achieves low storage consumption and I/O cost while supporting range and nearest neighbor queries, insertions, and deletions. In [PMW22b], the authors propose an instance-optimized Z-curve index with alternate ordering and partitioning, and two greedy heuristics to optimize ordering for specific workloads. When querying a traditional R-tree, all child nodes that overlap with a given range query must be visited. In [AI22], the authors propose identifying high-overlap queries and building an AI tree that uses uniquely

assigned IDs to the R-tree leaf nodes as class labels for multi-label classification. High-overlap queries use the AI tree to minimize the number of visited nodes, while low-overlap queries fall back on the regular R-tree. In [WY22], the authors focus on SAMs, compute the Z-curve extent of the MBR enclosing a spatial object, sort the objects based on the Z-address interval, and build a hierarchical tree structure. The internal nodes store linear regression models and pointers to the child nodes, while the leaf nodes store linear regression models and an array of actual objects with their MBRs. This differs from the ZM-index, which indexes points rather than spatial objects with extents, and from the Hilbert-curve index, which we also use for storing points. SPRIG [Zh21] is a grid index that uses spatial bilinear interpolation to predict the position of the query points and then refines the result using a local binary search. Spatial Join Machine Learning (SJML) [Vu21a, Vu22] is a machine learning-based query optimizer for distributed spatial joins. It consists of three levels: (1) a model that estimates the spatial join result size, (2) a model that predicts the number of geometric comparisons based on the predicted result size and other dataset characteristics, and (3) a classification model to predict the best join algorithm. In [Gu23], the authors identify that *ChooseSubtree* and *Split* operations for the R-tree construction can be considered sequential decision-making problems. They model them as two Markov Decision Processes (MDP) and use reinforcement learning to train a model for each. Machine learning techniques have been applied to spatial data in various scenarios, including spatio-textual queries [Di22], social media data [Gu22], passage retrieval [WMW22], streaming [Yu22], and other areas [Ki22, SUK22, Ya23, PMW22a, Gu24]. Several surveys provide an in-depth review of the various applications of machine learning to spatial data [SM19, SM20, SM21].

5 Conclusions

In this work, we implement learning-enhanced variants of six classical *read-only* spatial indexes. We found that, in most cases, the fixed grid outperforms other learning-enhanced indexes while being the simplest index to implement. **For range queries**, tuning indexes to the dataset and query workloads is crucial for performance, with learned search improving performance by 11-39% over binary search. We also found that learned models offer minimal gains for tree-based index structures, with fixed-grid outperforming them by $1.23\times$ to $1.83\times$. **For point queries**, fixed-grid was $1.51\times$ to $2.27\times$ faster than the closest tree-based competitor, k-d tree. The linearized Hilbert curve index also performed well, as point queries require searching for one point on the curve, in contrast to range queries where the index may scan many redundant points. **In distance queries**, fixed-grid has again the best performance, though gains diminish with higher selectivity, as the Haversine distance computation becomes the bottleneck. **For join queries**, the filter phase performs similarly to range queries, but the refinement phase is the dominant cost, with limited gains from learned indexes.

Literaturverzeichnis

- [Aj13] Aji, Ablimit; Wang, Fusheng; Vo, Hoang; Lee, Rubao; Liu, Qiaoling; Zhang, Xiaodong; Saltz, Joel H.: Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB*, 6(11):1009–1020, 2013.

- [Al22] Al-Mamun, Abdullah; Haider, Ch. Md. Rakin; Wang, Jianguo; Aref, Walid G.: The $\ddot{\text{A}}$ + R tree: An Instance-optimized R - tree. In: 23rd IEEE International Conference on Mobile Data Management, MDM 2022, Paphos, Cyprus, June 6-9, 2022. S. 9–18, 2022.
- [AN20] Amemiya, Koichiro; Nakao, Akihiro: Layer-Integrated Edge Distributed Data Store for Real-time and Stateful Services. In: NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020. S. 1–9, 2020.
- [Be75] Bentley, Jon Louis: Multidimensional Binary Search Trees Used for Associative Searching. Commun. ACM, 18(9):509–517, 1975.
- [BF79] Bentley, Jon Louis; Friedman, Jerome H.: Data Structures for Range Searching. ACM Comput. Surv., 11(4):397–409, 1979.
- [BS13] Bronshtein, Ilja N; Semendyayev, Konstantin A: Handbook of mathematics. Springer Science & Business Media, Germany, 2013.
- [Cu22] Cuza, Carlos Enrique Muniz; Ho, Nguyen; Tzirita Zacharatou, Eleni; Pedersen, Torben Bach; Yang, Bin: Spatio-temporal graph convolutional network for stochastic traffic speed imputation. In: Proceedings of the 30th International Conference on Advances in Geographic Information Systems. S. 1–12, 2022.
- [Dat21] Databricks Runtime 7.6, 2021. <https://docs.databricks.com/release-notes/runtime/7.6.html>.
- [DF20] Doraiswamy, Harish; Freire, Juliana: A GPU-friendly Geometric Data Model and Algebra for Spatial Queries. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020. S. 1875–1885, 2020.
- [DF22a] Doraiswamy, Harish; Freire, Juliana: GPU-Powered Spatial Database Engine for Commodity Hardware: Extended Version. CoRR, abs/2203.14362, 2022.
- [DF22b] Doraiswamy, Harish; Freire, Juliana: SPADE: GPU-Powered Spatial Database Engine for Commodity Hardware. In: 38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022. S. 2669–2681, 2022.
- [Di20a] Ding, Jialin; Nathan, Vikram; Alizadeh, Mohammad; Kraska, Tim: Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. Proc. VLDB Endow., 14(2):74–86, 2020.
- [Di20b] Ding, Jialin; Nathan, Vikram; Alizadeh, Mohammad; Kraska, Tim: Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. Proc. VLDB Endow., 14(2):74–86, 2020.
- [Di21] Ding, Jialin; Minhas, Umar Farooq; Chandramouli, Badrishi; Wang, Chi; Li, Yinan; Li, Ying; Kossmann, Donald; Gehrke, Johannes; Kraska, Tim: Instance-Optimized Data Layouts for Cloud Analytics Workloads. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. S. 418–431, 2021.
- [Di22] Ding, Xiaofeng; Zheng, Yinting; Wang, Zuan; Choo, Kim-Kwang Raymond; Jin, Hai: A learned spatial textual index for efficient keyword queries. Journal of Intelligent Information Systems, S. 1–25, 2022.
- [DMM20] Davitkova, Angjela; Milchevski, Evica; Michel, Sebastian: The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In: 2020 Conference on Extending Database Technology (EDBT). 2020.

- [Do22] Dong, Haowen; Chai, Chengliang; Luo, Yuyu; Liu, Jiabin; Feng, Jianhua; Zhan, Chaoqun: RW-Tree: A Learned Workload-aware Framework for R-tree Construction. In: 38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022. IEEE, S. 2073–2085, 2022.
- [El17] Eldawy, Ahmed; Sabek, Ibrahim; Elganainy, Mostafa; Bakeer, Ammar; Abdelmotaleb, Ahmed; Mokbel, Mohamed F.: Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. In: Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings. S. 65–83, 2017.
- [El21] Eldawy, Ahmed; Hristidis, Vagelis; Ghosh, Saheli; Saeedan, Majid; Sevim, Akil; Siddique, A. B.; Singla, Samriddhi; Sivaram, Ganesh; Vu, Tin; Zhang, Yaming: Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In (Demartini, Gianluca; Zuccon, Guido; Culpepper, J. Shane; Huang, Zi; Tong, Hanghang, Hrsg.): CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021. S. 3796–3807, 2021.
- [EM15] Eldawy, Ahmed; Mokbel, Mohamed F.: SpatialHadoop: A MapReduce framework for spatial data. In: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015. S. 1352–1363, 2015.
- [FB74] Finkel, Raphael A.; Bentley, Jon Louis: Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.
- [Ga20] García-García, Francisco; Corral, Antonio; Iribarne, Luis; Vassilakopoulos, Michael: Improving Distance-Join Query processing with Voronoi-Diagram based partitioning in SpatialHadoop. *Future Gener. Comput. Syst.*, 111:723–740, 2020.
- [Ga23] Gao, Jian; Cao, Xin; Yao, Xin; Zhang, Gong; Wang, Wei: LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves. *Proc. VLDB Endow.*, 16(10):2605–2617, 2023.
- [GG98] Gaede, Volker; Günther, Oliver: Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.
- [Go19] Gomes, David: . MemSQL Live: Nikita Shamgunov on the Data Engineering Podcast, 2019. <https://www.memsql.com/blog/memsql-live-nikita-shamgunov-on-the-data-engineering-podcast/>.
- [GTM23] Georgiadis, Thanasis; Tzirita Zacharatou, Eleni; Mamoulis, Nikos: APRIL: Approximating Polygons as Raster Interval Lists. *CoRR*, abs/2307.01716, 2023.
- [Gu84] Guttman, Antonin: R-Trees: A Dynamic Index Structure for Spatial Searching. In: SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984. S. 47–57, 1984.
- [Gu22] Guo, Na; Wang, Yaqi; Jiang, Haonan; Xia, Xiufeng; Gu, Yu: TALI: An Update-Distribution-Aware Learned Index for Social Media Data. *Mathematics*, 10(23):4507, 2022.
- [Gu23] Gu, Tu; Feng, Kaiyu; Cong, Gao; Long, Cheng; Wang, Zheng; Wang, Sheng: The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proc. ACM Manag. Data*, 1(1):63:1–63:26, 2023.
- [Gu24] Gu, Tu; Feng, Kaiyu; Yang, Jingyi; Cong, Gao; Long, Cheng; Zhang, Rui: BT-Tree: A Reinforcement Learning Based Index for Big Trajectory Data. *Proc. ACM Manag. Data*, 2(4):194:1–194:27, 2024.

- [HGS17] Hagedorn, Stefan; Götze, Philipp; Sattler, Kai-Uwe: The STARK Framework for Spatio-Temporal Data Analytics on Spark. In: Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informati-onssysteme“(DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings. S. 123–142, 2017.
- [Hi35] Hilbert, David: Über die stetige Abbildung einer Linie auf ein Flächenstück. In: Dritter Band: Analysis. Grundlagen der Mathematik. Physik Verschiedenes, S. 1–2. 1935.
- [HKH20] Hadian, Ali; Kumar, Ankit; Heinis, Thomas: Hands-off Model Integration in Spatial Index Structures. In (He, Bingsheng; Reinwald, Berthold; Wu, Yingjun, Hrsg.): AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan. 2020.
- [HWL23] Huang, Shuai; Wang, Yong; Li, Guoliang: ACR-Tree: Constructing R-Trees Using Deep Reinforcement Learning. In: Database Systems for Advanced Applications - 28th International Conference, DASFAA 2023, Tianjin, China, April 17-20, 2023, Proceedings, Part I. Jgg. 13943 in Lecture Notes in Computer Science. Springer, S. 80–96, 2023.
- [Ja05] Jagadish, H. V.; Ooi, Beng Chin; Tan, Kian-Lee; Yu, Cui; Zhang, Rui: IDistance: An Adaptive B+-Tree Based Indexing Method for Nearest Neighbor Search. ACM Trans. Database Syst., 30(2):364–397, Juni 2005.
- [KAI17] Kester, Michael S.; Athanassoulis, Manos; Idreos, Stratos: Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. S. 715–730, 2017.
- [Ki18] Kipf, Andreas; Lang, Harald; Pandey, Varun; Persa, Raul Alexandru; Boncz, Peter A.; Neumann, Thomas; Kemper, Alfons: Approximate Geospatial Joins with Precision Guarantees. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018. S. 1360–1363, 2018.
- [Ki20a] Kipf, Andreas; Lang, Harald; Pandey, Varun; Persa, Raul Alexandru; Anneser, Christoph; Tzirita Zacharou, Eleni; Doraiswamy, Harish; Boncz, Peter A.; Neumann, Thomas; Kemper, Alfons: Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins. In: Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020. S. 347–358, 2020.
- [Ki20b] Kipf, Andreas; Marcus, Ryan; van Renen, Alexander; Stoian, Mihail; Kemper, Alfons; Kraska, Tim; Neumann, Thomas: RadixSpline: a single-pass learned index. In (Bordawekar, Rajesh; Shmueli, Oded; Tatbul, Nesime; Ho, Tin Kam, Hrsg.): Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020. ACM, S. 5:1–5:5, 2020.
- [Ki20c] Kipf, Andreas; Marcus, Ryan; van Renen, Alexander; Stoian, Mihail; Kemper, Alfons; Kraska, Tim; Neumann, Thomas: RadixSpline: a single-pass learned index. In: Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020. S. 5:1–5:5, 2020.
- [Ki22] Kipf, Andreas; Horn, Dominik; Pfeil, Pascal; Marcus, Ryan; Kraska, Tim; LSI: a learned secondary index structure. In (Bordawekar, Rajesh; Shmueli, Oded; Amsterdamer, Yael;

Firmani, Donatella; Marcus, Ryan, Hrsg.): aiDM '22: Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Philadelphia, Pennsylvania, USA, 17 June 2022. ACM, S. 4:1–4:5, 2022.

- [KP07] Kim, You Jung; Patel, Jignesh M.: Rethinking Choices for Multi-dimensional Point Indexing: Making the Case for the Often Ignored Quadtree. In: CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings. S. 281–291, 2007.
- [Kr18] Kraska, Tim; Beutel, Alex; Chi, Ed H.; Dean, Jeffrey; Polyzotis, Neoklis: The Case for Learned Index Structures. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. S. 489–504, 2018.
- [Kr21] Kraska, Tim: Towards instance-optimized data systems. Proc. VLDB Endow., 14(12):3222–3232, 2021.
- [KRA02] Kanth, Kothuri Venkata Ravi; Ravada, Siva; Abugov, Daniel: Quad-tree and R-tree indexes in oracle spatial: a comparison using GIS data. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002. S. 546–557, 2002.
- [LEL97] Leutenegger, Scott T.; Edgington, J. M.; López, Mario Alberto: STR: A Simple and Efficient Algorithm for R-Tree Packing. In: Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK. S. 497–506, 1997.
- [Li20] Li, Pengfei; Lu, Hua; Zheng, Qian; Yang, Long; Pan, Gang: LISA: A Learned Index Structure for Spatial Data. In: Proceedings of the 2020 International Conference on Management of Data. SIGMOD '20, 2020.
- [Li23a] Li, Jiangneng; Wang, Zheng; Cong, Gao; Long, Cheng; Kiah, Han Mao; Cui, Bin: Towards Designing and Learning Piecewise Space-Filling Curves. Proc. VLDB Endow., 16(9):2158–2171, 2023.
- [Li23b] Li, Jiangneng; Wang, Zheng; Cong, Gao; Long, Cheng; Kiah, Han Mao; Cui, Bin: Towards Designing and Learning Piecewise Space-Filling Curves. Proc. VLDB Endow., 16(9):2158–2171, 2023.
- [Li24] Liu, Qiyu; Li, Maocheng; Zeng, Yuxiang; Shen, Yanyan; Chen, Lei: How Good Are Multi-dimensional Learned Indices? An Experimental Survey. CoRR, abs/2405.05536, 2024.
- [LK00] Lawder, Jonathan K.; King, Peter J. H.: Using Space-Filling Curves for Multi-dimensional Indexing. In (Lings, Brian; Jeffery, Keith G., Hrsg.): Advances in Databases, 17th British National Conference on Databases, BNCOD 17, Exeter, UK, July 3-5, 2000, Proceedings. Jgg. 1832 in Lecture Notes in Computer Science, S. 20–35, 2000.
- [LK01] Lawder, Jonathan K.; King, Peter J. H.: Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. SIGMOD Rec., 30(1):19–24, 2001.
- [Ma19] Makris, Antonios; Tserpes, Konstantinos; Spiliopoulos, Giannis; Anagnostopoulos, Dimitris; Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data. In: Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019. Jgg. 2322 in CEUR Workshop Proceedings, 2019.
- [Ma24] Mamun, Abdullah Al; Wu, Hao; He, Qiyang; Wang, Jianguo; Aref, Walid G.: A Survey of Learned Indexes for the Multi-dimensional Space. CoRR, abs/2403.06456, 2024.

- [Mo01] Moon, Bongki; Jagadish, H. V.; Faloutsos, Christos; Saltz, Joel H.: Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Trans. Knowl. Data Eng.*, 13(1):124–141, 2001.
- [Mon13] MongoDB Releases - New Geo Features in MongoDB 2.4, 2013. <https://www.mongodb.com/blog/post/new-geo-features-in-mongodb-24/>.
- [Na20] Nathan, Vikram; Ding, Jialin; Alizadeh, Mohammad; Kraska, Tim: Learning Multi-dimensional Indexes. In: Proceedings of the 2020 International Conference on Management of Data. SIGMOD '20, 2020.
- [NHS84] Nievergelt, Jürg; Hinterberger, Hans; Sevcik, Kenneth C.: The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.
- [NYC19] NYC Taxi and Limousine Commission (TLC) - TLC Trip Record Data, 2019. <https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page>.
- [Or89] Orenstein, Jack A.: Redundancy in Spatial Databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989.
- [Ora19] Oracle Spatial and Graph Spatial Features, 2019. <https://www.oracle.com/technetwork/database/options/spatialandgraph/overview/spatialfeatures-1902020.html/>.
- [Pa16] Pandey, Varun; Kipf, Andreas; Vorona, Dimitri; Mühlbauer, Tobias; Neumann, Thomas; Kemper, Alfons: High-Performance Geospatial Analytics in HyPerSpace. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. S. 2145–2148, 2016.
- [Pa18] Pandey, Varun; Kipf, Andreas; Neumann, Thomas; Kemper, Alfons: How Good Are Modern Spatial Analytics Systems? Proc. VLDB Endow., 11(11):1661–1673, 2018.
- [Pa20a] Pandey, Varun; van Renen, Alexander; Kipf, Andreas; Ding, Jialin; Sabek, Ibrahim; Kemper, Alfons: The Case for Learned Spatial Indexes. In: AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan. 2020.
- [Pa20b] Pandey, Varun; van Renen, Alexander; Kipf, Andreas; Kemper, Alfons: An Evaluation Of Modern Spatial Libraries. In: Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 21-24, 2020, Proceedings, Part II. Jgg. 12113 in Lecture Notes in Computer Science, S. 157–174, 2020.
- [Pa21] Pandey, Varun; van Renen, Alexander; Kipf, Andreas; Kemper, Alfons: How Good Are Modern Spatial Libraries? Data Sci. Eng., 6(2):192–208, 2021.
- [PMW22a] Pai, Sachith Gopalakrishna; Mathioudakis, Michael; Wang, Yanhao: Towards an Instance-Optimal Z-Index. 2022.
- [PMW22b] Pai, Sachith Gopalakrishna; Mathioudakis, Michael; Wang, Yanhao: Towards an Instance-Optimal Z-Index. 2022.
- [PMW24] Pai, Sachith; Mathioudakis, Michael; Wang, Yanhao: WaZI: A Learned and Workload-aware Z-Index. In: Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28. OpenProceedings.org, S. 559–571, 2024.
- [Pos23] PostGIS, 2023. <http://postgis.net/>.

- [Qi20] Qi, Jianzhong; Liu, Guanli; Jensen, Christian S.; Kulik, Lars: Effectively Learning Spatial Indices. Proc. VLDB Endow., 13(11):2341–2354, 2020.
- [RKV95] Roussopoulos, Nick; Kelley, Stephen; Vincent, Frédéric: Nearest Neighbor Queries. In (Carey, Michael J.; Schneider, Donovan A., Hrsg.): Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995. ACM Press, S. 71–79, 1995.
- [Sa22] Sabek, Ibrahim; Vaidya, Kapil; Horn, Dominik; Kipf, Andreas; Mitzenmacher, Michael; Kraska, Tim: Can Learned Models Replace Hash Functions? Proc. VLDB Endow., 16(3):532–545, 2022.
- [Se79] Selinger, Patricia G.; Astrahan, Morton M.; Chamberlin, Donald D.; Lorie, Raymond A.; Price, Thomas G.: Access Path Selection in a Relational Database Management System. In (Bernstein, Philip A., Hrsg.): Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1. S. 23–34, 1979.
- [SE22] Saeedan, Majid; Eldawy, Ahmed: Spatial parquet: a column file format for geospatial data lakes. In (Renz, Matthias; Sarwat, Mohamed, Hrsg.): Proceedings of the 30th International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2022, Seattle, Washington, November 1-4, 2022. ACM, S. 102:1–102:4, 2022.
- [Si18] Sidlauskas, Darius; Chester, Sean; Tzirita Zacharou, Eleni; Ailamaki, Anastasia: Improving Spatial Data Processing by Clipping Minimum Bounding Boxes. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018. S. 425–436, 2018.
- [SK23] Sabek, Ibrahim; Kraska, Tim: The Case for Learned In-Memory Joins. Proc. VLDB Endow., 16(7):1749–1762, 2023.
- [SM19] Sabek, Ibrahim; Mokbel, Mohamed F.: Machine Learning Meets Big Spatial Data. Proc. VLDB Endow., 12(12):1982–1985, 2019.
- [SM20] Sabek, Ibrahim; Mokbel, Mohamed F.: Machine Learning Meets Big Spatial Data. In: 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020. S. 1782–1785, 2020.
- [SM21] Sabek, Ibrahim; Mokbel, Mohamed F.: Machine Learning Meets Big Spatial Data (Revised). In: 22nd IEEE International Conference on Mobile Data Management, MDM 2021, Toronto, ON, Canada, June 15-18, 2021. S. 5–8, 2021.
- [SUK22] Sabek, Ibrahim; Ukyab, Tenzin Samten; Kraska, Tim: LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. 2022.
- [Ta16] Tang, MingJie; Yu, Yongyang; Malluhi, Qutaibah M.; Ouzzani, Mourad; Aref, Walid G.: LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. PVLDB, 9(13):1565–1568, 2016.
- [Th19] Theocharidis, Konstantinos; Liagouris, John; Mamoulis, Nikos; Bouros, Panagiotis; Terrovitis, Manolis: SRX: efficient management of spatial RDF data. VLDB J., 28(5):703–733, 2019.
- [To20] Toliopoulos, Theodoros; Nikolaidis, Nikodimos; Michailidou, Anna-Valentini; Seitaridis, Andreas; Gounaris, Anastasios; Bassiliades, Nick; Georgiadis, Apostolos; Liopoulos, Fotis: Developing a Real-Time Traffic Reporting and Forecasting Back-End System. In:

Research Challenges in Information Science - 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23-25, 2020, Proceedings. Jgg. 385 in Lecture Notes in Business Information Processing, S. 58–75, 2020.

- [TR20] Tahboub, Ruby Y.; Rompf, Tiark: Architecting a Query Compiler for Spatial Workloads. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020. S. 2103–2118, 2020.
- [Ts19] Tsitsigkos, Dimitrios; Bouros, Panagiotis; Mamoulis, Nikos; Terrovitis, Manolis: Parallel In-Memory Evaluation of Spatial Joins. In: Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019. S. 516–519, 2019.
- [Twe20] Tutorials: Filtering Tweets by location, 2020. <https://developer.twitter.com/en/docs/tutorials/filtering-tweets-by-location>.
- [Tz17] Tzirita Zacharatou, Eleni; Doraiswamy, Harish; Ailamaki, Anastasia; Silva, Cláudio T.; Freire, Juliana: GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. PVLDB, 11(3):352–365, 2017.
- [Tz19] Tzirita Zacharatou, Eleni; Sidlauskas, Darius; Tauheed, Farhan; Heinis, Thomas; Ailamaki, Anastasia: Efficient Bundled Spatial Range Queries. In: Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019. S. 139–148, 2019.
- [Tz21] Tzirita Zacharatou, Eleni; Kipf, Andreas; Sabek, Ibrahim; Pandey, Varun; Doraiswamy, Harish; Markl, Volker: The Case for Distance-Bounded Spatial Approximations. In: 11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings. 2021.
- [Ub18] Uber: Uber Newsroom: 10 Billion, 2018. <https://www.uber.com/newsroom/10-billion/>.
- [Vu21a] Vu, Tin; Belussi, Alberto; Migliorini, Sara; Eldawy, Ahmed: A Learned Query Optimizer for Spatial Join. In (Meng, Xiaofeng; Wang, Fusheng; Lu, Chang-Tien; Huang, Yan; Shekhar, Shashi; Xie, Xing, Hrsg.): SIGSPATIAL ’21: 29th International Conference on Advances in Geographic Information Systems, Virtual Event / Beijing, China, November 2-5, 2021. S. 458–467, 2021.
- [Vu21b] Vu, Tin; Eldawy, Ahmed; Hristidis, Vagelis; Tsotras, Vassilis J.: Incremental Partitioning for Efficient Spatial Data Analytics. Proc. VLDB Endow., 15(3):713–726, 2021.
- [Vu22] Vu, Tin; Belussi, Alberto; Migliorini, Sara; Eldawy, Ahmed: Towards a Learned Cost Model for Distributed Spatial Join: Data, Code & Models. In (Hasan, Mohammad Al; Xiong, Li, Hrsg.): Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022. S. 4550–4554, 2022.
- [Wa19] Wang, H.; Fu, X.; Xu, J.; Lu, H.: Learned Index for Spatial Queries. In: 2019 20th IEEE International Conference on Mobile Data Management (MDM). S. 569–574, 2019.
- [Wi21] Winter, Christian; Kipf, Andreas; Anneser, Christoph; Tzirita Zacharatou, Eleni; Neumann, Thomas; Kemper, Alfons: GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons. In: Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021. S. 169–180, 2021.

- [WMW22] Wang, Yifan; Ma, Haodi; Wang, Daisy Zhe: LIDER: An Efficient High-dimensional Learned Index for Large-scale Dense Passage Retrieval. Proc. VLDB Endow., 16(2):154–166, 2022.
- [WY22] Wang, Congying; Yu, Jia: GLIN: A Lightweight Learned Indexing Mechanism for Complex Geometries. CoRR, abs/2207.07745, 2022.
- [Xi16] Xie, Dong; Li, Feifei; Yao, Bin; Li, Gefei; Zhou, Liang; Guo, Minyi: Simba: Efficient In-Memory Spatial Analytics. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. S. 1071–1085, 2016.
- [Ya20] Yang, Zongheng; Chandramouli, Badrish; Wang, Chi; Gehrke, Johannes; Li, Yinan; Minhas, Umar F.; Larson, Per-Åke; Kossmann, Donald; Acharya, Rajeev: Qd-tree: Learning Data Layouts for Big Data Analytics. In: Proceedings of the 2020 International Conference on Management of Data. SIGMOD '20, 2020.
- [Ya23] Yang, Guang; Liang, Liang; Hadian, Ali; Heinis, Thomas: FLIRT: A Fast Learned Index for Rolling Time frames. In: Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023. S. 234–246, 2023.
- [YC23] Yang, Jingyi; Cong, Gao: PLATON: Top-down R-tree Packing with Learned Partition Policy. Proc. ACM Manag. Data, 1(4):253:1–253:26, 2023.
- [Yu22] Yu, Tong; Liu, Guanfeng; Liu, An; Li, Zhixu; Zhao, Lei: LIFOSS: a learned index scheme for streaming scenarios. World Wide Web, S. 1–18, 2022.
- [YWS15] Yu, Jia; Wu, Jinxuan; Sarwat, Mohamed: GeoSpark: a cluster computing framework for processing large-scale spatial data. In: Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015. S. 70:1–70:4, 2015.
- [YZG15] You, Simin; Zhang, Jianting; Gruenwald, Le: Large-scale spatial join query processing in Cloud. In: 31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015. S. 34–41, 2015.
- [Zh21] Zhang, Songnian; Ray, Suprio; Lu, Rongxing; Zheng, Yandong: SPRIG: A Learned Spatial Index for Range and kNN Queries. In: Proceedings of the 17th International Symposium on Spatial and Temporal Databases, SSTD 2021, Virtual Event, USA, August 23-25, 2021. S. 96–105, 2021.