Enhancing In-Memory Spatial Indexing with Learned Search

VARUN PANDEY[*], TU Berlin, Germany
ALEXANDER VAN RENEN[*†], Amazon Web Services, Germany
ELENI TZIRITA ZACHARATOU, IT University of Copenhagen, Denmark
ANDREAS KIPF[†], Amazon Web Services, Germany
IBRAHIM SABEK, University of Southern California, USA
JIALIN DING[†], Amazon Web Services, Germany
VOLKER MARKL, TU Berlin and DFKI GmbH, Germany
ALFONS KEMPER, TU Munich, Germany

Spatial data is ubiquitous. Massive amounts of data are generated every day from a plethora of sources such as billions of GPS-enabled devices (e.g., cell phones, cars, and sensors), consumer-based applications (e.g., Uber and Strava), and social media platforms (e.g., location-tagged posts on Facebook, Twitter, and Instagram). This exponential growth in spatial data has led the research community to build systems and applications for efficient spatial data processing. In this study, we apply a recently developed machine-learned search technique for single-dimensional sorted data to spatial indexing. 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. Adhering to the latest research trends, we tune the partitioning techniques to be instance-optimized. By tuning each partitioning technique for optimal performance, we demonstrate that: (i) grid-based index structures outperform tree-based index structures (from 1.23× to 2.47×), (ii) learning-enhanced variants of commonly used spatial index structures outperform their original counterparts (from 1.44× to 53.34× faster), (iii) machine-learned search within a partition is faster than binary search by 11.79% - 39.51% when filtering on one dimension, (iv) the benefit of machine-learned search diminishes 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 potentially be reduced by linearizing the indexed partitions.

Additional Key Words and Phrases: spatial data, indexing, machine-learning, spatial queries, geospatial

[*]These authors contributed equally to this work.
[†]Work done prior to joining Amazon.

Authors' addresses: Varun Pandey, TU Berlin, Berlin, Germany, varun.pandey@tu-berlin.de; Alexander van Renen, Amazon Web Services, Munich, Germany, vanralex@amazon.com; Eleni Tzirita Zacharatou, IT University of Copenhagen, Copenhagen, Denmark, elza@itu.dk; Andreas Kipf, Amazon Web Services, Munich, Germany, kipf@amazon.com; Ibrahim Sabek, University of Southern California, Los Angeles, USA, sabek@usc.edu; Jialin Ding, Amazon Web Services, Munich, Germany, jialind@amazon.com; Volker Markl, TU Berlin and, DFKI GmbH, Berlin, Germany, volker.markl@tu-berlin.de; Alfons Kemper, TU Munich, Munich, Germany, kemper@in.tum.de.

1 INTRODUCTION

With the increase in the amount of spatial data available today, the database community has devoted substantial attention to spatial data management. For example, the NYC Taxi Rides open dataset [53] consists of pick-up and drop-off locations of more than 2.7 billion rides taken in the city since 2009. This represents more than 650,000 taxi rides per day in one of the most densely populated cities in the world but is only a fraction of the location data that is captured by many applications today. Uber, a popular ride-hailing service, operates on a global scale and completed 10 billion rides in 2018 [89]. The unprecedented generation rate of location data has prompted a considerable amount of research efforts focused on scale-out systems [1, 3, 17–19, 29, 79–81, 97, 100, 101], databases [24, 48, 49, 54, 59], improving spatial query processing [22, 23, 36–38, 65, 72, 73, 75, 82–84, 87, 88, 92, 96, 103], or leveraging modern hardware and compiling techniques [13–16, 76–78, 86], to handle the increasing demands of applications today.

Recently, Kraska et al. [42] 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. [33] 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 top of these recent research results and thoroughly investigate the impact of applying ideas from learned index structures (e.g., Flood [51]) on classical multidimensional indexes.

Specifically, we focus on six core spatial indexing techniques, namely linearization using Hilbert space-filling curve, fixed-grid [5], adaptive-grid [52], Kd-tree [4], Quadtree [20] and STRtree [45]. 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 [39], 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-selectivity [1] range queries (similar to the observation from Kester et al. [33]). 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 (which selects 0.00001% of the data, i.e., 8 records), the index and boundary refinement times dominate the lookup. In contrast, for a high-selectivity query (which selects 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 (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 for 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 *simplest* to implement.

In this paper, we extend the work presented in our previous publication [60]. 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

[1] We adopt the definition of "selectivity" used by Pat Selinger et al. [74]. Therefore, low selectivity indicates that the result set of a query has few qualifying tuples, while high selectivity indicates the opposite.

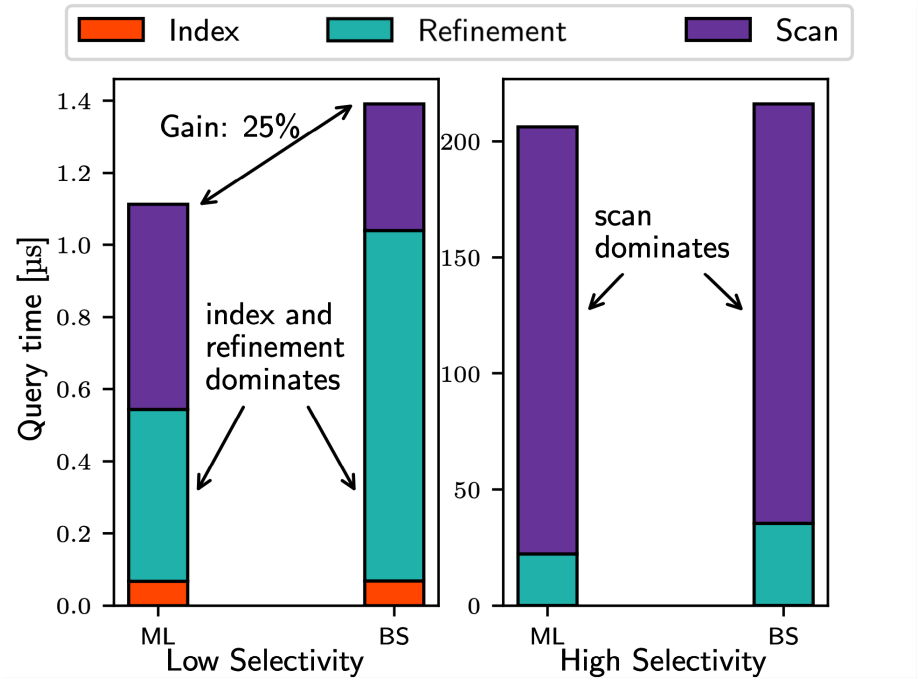


Fig. 1. Machine Learning vs. Binary Search (Spatial Range Query). For low selectivity (0.00001%), the index and boundary refinement phases dominate. For high selectivity (0.1%), the scan phase dominates. Parameters are tuned to favor Binary Search.

previous research by adding *three more query types*, *one more learned index*, and *two competitive methods* commonly used in various applications and systems. Specifically, we extend the previous work as follows:

- **Queries**: We implement *three additional query types* (i.e., point, distance, and spatial join) in this work, bringing the total number of evaluated queries to four.
- **Index Structures**: We also *add a learning-enhanced index based on linearization using the Hilbert curve* to the five previously evaluated learning-enhanced indexes (i.e., fixed-grid, adaptive-grid, K-d tree, Quadtree, and STRtree), bringing the total number of evaluated learning-enhanced indexes to six.
- **Competitors**: We further extend our preliminary study by including two index structures that are widely used in hundreds of applications and systems [61, 62]. Concretely, we implement *all four* queries using the JTS STRtree from the JTS library and the S2PointIndex from the Google S2 library.

We summarize the main findings of our experimental study as follows:

- **Grid-based vs Tree-based**: Grid-based index structures, when tuned for optimal performance, outperform tree-based index structures due to (1) fewer random accesses, and (2) allowing for fast search using learned models over large partitions. They are up to 2.47× *faster* compared to the closest tree-based competitor, exhibit *robust* performance across various queries, and are also the *simplest* to implement.
- **Effect of Learned Search**: We show that using learned models to search within partitions is 11.79% to 39.51% faster than binary search.

- **Compute-Intensive Operations**: The combined effect of grid-based indexes and learned models diminishes in the presence of compute-intensive operations, such as Haversine distance computations and point-in-polygon tests.
- **Performance Compared to Widely Used Indexes**: When compared to two commonly used index structures in various systems and applications, machine-learned indexes demonstrate superior performance, with speedups ranging from 1.44× to 53.34×.

Our study sheds light on the effectiveness of different spatial index structures when used in conjunction with learned models. By evaluating multiple query types and index structures, our aim is to guide researchers and practitioners in choosing the best approach for their specific 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 the related work in Section 4 before concluding in Section 5.

2 APPROACH

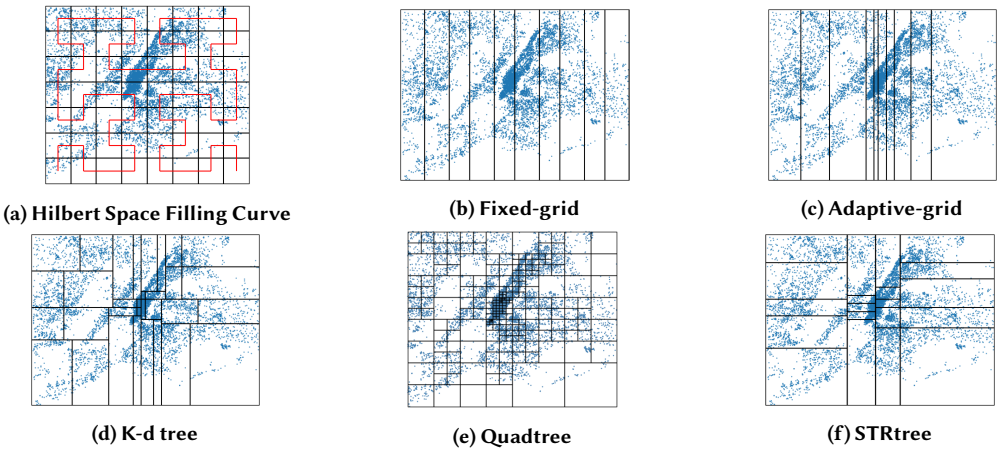


Fig. 2. An illustration of the different partitioning techniques.

In this section, we first describe the spatial indexing techniques that we implemented in our work (Section 2.1). We then proceed to explain how we built the learned indexes (Section 2.2). Finally, Sections 2.3- 2.6 explain how we implemented each of the queries covered in this work (i.e., range, point, distance, and spatial join).

2.1 Indexing Techniques

Multidimensional access methods are broadly classified into two categories: Point Access Methods (PAMs) and Spatial Access Methods (SAMs) [21]. PAMs are designed to handle point data without spatial extent. SAMs, however, manage extended objects such as linestrings and polygons. In this work, we focus mainly on PAMs.

Spatial partitioning is the process of splitting a spatial dataset into partitions, or cells, where objects within the same partition are close to each other in space. Spatial partitioning techniques can be divided into two categories: space partitioning ones, which partition the embedded space, and data partitioning ones, which partition the data space. In this work, we

employ three space partitioning techniques: linearization using the Hilbert curve, fixed-grid [5], and Quadtree [20]. Additionally, we employ three data partitioning techniques: adaptive-grid [52], K-d tree [4], and Sort-Tile-Recursive (STR) [45]. Figure 2 illustrates these techniques on a sample of the Tweets dataset that we use in our experiments (further details can be found in Section 3.2). The figure shows the sample points and partition boundaries as dots and grid axes, respectively.

2.1.1 Linearization using Hilbert Curve. In a one-dimensional data space, all data points can be sorted along a single dimension, making it easy to perform range queries by retrieving all data between given lower and upper bounds. However, applying learned models to a spatial (or multidimensional) index is challenging because multidimensional data does not have an inherent sort order. One way to tackle this challenge is to linearize the multidimensional space using space-filling curves. In this work, we utilize the Hilbert space-filling curve (SFC) [30] to map a multidimensional vector space to a one-dimensional space. As shown in Figure 2(a), the process of linearization using the Hilbert SFC involves dividing the two-dimensional space with a uniform grid and then using the Hilbert curve to enumerate the cells of this grid. Once all cells have been enumerated, we can sort their identifiers, and thus learn an index on this sorted order. This approach is similar to the recently proposed Z-order Model (ZM) index [94], which uses the Z-curve to enumerate the cells. We chose the Hilbert curve instead as it has been shown to perform better for multi-dimensional indexing [43, 44, 50]. The Hilbert curve is also the choice of multidimensional clustering in the recent Databricks runtime engine [7], replacing the Z-curve. However, we show that linearization-based techniques can suffer from skewed cases, where the queries cover a large portion of the curve, as we will demonstrate in Section 3.4.2. For example, if a query rectangle covers all partitions at the bottom of Figure 2(a), the whole curve would lie within the query rectangle. This would result in scanning a large number of points that do not satisfy the query, leading to poor performance.

2.1.2 Fixed and Adaptive Grid. Grid-based indexing methods are primarily designed to optimize the retrieval of records from disk. They work by dividing the d-dimensional attribute space into cells, where each cell corresponds to one data page (or bucket) and stores a pointer to the data page it indexes. Data points that fall within the boundaries of a cell are stored on the corresponding data page. This allows for quick navigation to the specific data page containing the desired records, rather than having to search through the entire dataset. The fixed-grid [5] enforces equidistant grid lines, while the adaptive-grid (or grid file [52]) 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 the 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 [4] is a binary search tree that recursively subdivides the space into equal subspaces using rectilinear (or iso-oriented) hyperplanes. The splitting hyperplanes, known as discriminators, alternate between the k dimensions at each level of the tree. For example, in a 2-dimensional space, the splitting hyperplanes are alternately perpendicular to the x- and y-axes. The original K-d tree partitions the *space* into equal partitions. For example, if the input space consists of a GPS coordinate system (-90.0, -180 to 90, 180), it would be divided into equal halves (-45, -90 to 45, 90). This results in an unbalanced K-d tree if the data is skewed, i.e., most of the data lies in one partition. However, we can make the K-d tree data aware by dividing the data at each level into two halves based on the median point in the data. This ensures that both partitions in the binary tree are balanced. In our work, we implemented this data-aware version of the K-d tree.

2.1.4 Quadtree. The Quadtree [20], along with its many variants, is a tree data structure that partitions the space similarly to the K-d tree. The term quadtree typically refers to the two-dimensional variant, but the concept can easily be generalized to multiple dimensions. Like the K-d tree, the Quadtree decomposes the space using rectilinear hyperplanes. However, it differs from the K-d tree in that it is not a binary tree. For d dimensions, interior nodes have 2^d children. In the case of 2 dimensions, each interior node has four children, each representing a rectangle. The search space is recursively divided into four quadrants until the number of objects in each quadrant is below 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 [27] is a hierarchical data structure that is primarily designed for the efficient execution of range queries. The R-tree approximates arbitrary geometric objects with their minimum bounding rectangle (MBR) and stores the resulting collection of rectangles. Each node in the R-tree stores a maximum of N entries, each containing a rectangle R and a pointer P. At the leaf level, P points to the actual object and R is the MBR of the object. In internal nodes, R represents the MBR of the subtree pointed to by P.

The Sort-Tile-Recursive (STR) packing algorithm [45] is a method for filling R-trees that aims to maximize space utilization. The main idea behind STR packing is to tile the data space into an $S \times S$ grid. Assuming that the number of points in a data set is P and the capacity of a node is N, $S = \sqrt{P/N}$. STR first sorts the data on the x-dimension (in the case of rectangles, the x-dimension of the centroid), and then divides it into S vertical *slices*. Within each vertical slice, it sorts the data on the y-dimension and packs it into nodes by grouping them into runs of length N, forming S horizontal slices. This process continues recursively, resulting in completely filled nodes, except for the last node 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 that index a given location dataset D, which contains points in latitude/longitude format (referred to as the x-dimension and y-dimension respectively for ease of understanding). First, we partition D using one of the techniques described in Section 2.1. Each partition has a size of l points, also known as the leaf size or partition size. Once D has been partitioned, we iterate through all partitions and sort all the points within each partition on the y-dimension. Then, we build a learned index on the y-dimension for each partition. Algorithm 1 outlines the index building process.

Algorithm 1: A generic way of building learning-enhanced indexes

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

```
D' ← {}
P ← Partition(some approach from the techniques described in Section 2.1, l)
for p ∈ P do
    Sort(p, y)
    BuildLearnedIndex(p, y)
    D' ← D' ∪ {p}
end
return D'
```

2.3 Range Query Processing

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

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

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 involves identifying the partitions that intersect with the given range query using the index directory, i.e., the grid directories or trees. These partitions are represented by IP, which stands for intersected partitions and is reflected in line 2 of Algorithm 2. Note that the specific method used for this step depends on the partitioning technique.

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 ← {}
   /* find intersected partitions (IP) */
2  IP ← IndexLookup(D′, q)
3  for ip ∈ IP do
       /* if completely inside x-dimension range */
4      if q_xl <= ip_xl and ip_xh <= q_xh then
           /* if also completely inside y-dimension range, copy entire partition */
5          if q_yl <= ip_yl and ip_yh <= q_yh then
               /* copy all points in partition */
6              RQ ← RQ ∪ ip
7          else
               /* lower bound */
8              lb ← EstimateFrom(ip, q_yl)
               /* get exact lower bound */
9              lb ← LocalSearch(ip, lb, q_yl)
               /* upper bound */
10             ub ← EstimateTo(ip, q_yh)
               /* get exact upper bound */
11             ub ← LocalSearch(ip, ub, q_yh)
               /* copy all points between lower and upper bound */
12             RQ ← RQ ∪ ip.range(lb, ub)
13         end
14     else
           /* lower bound */
15         lb ← EstimateFrom(ip, q_yl)
16         lb ← SearchPoint(ip, lb, q_yl)
           /* upper bound */
17         ub ← EstimateTo(ip, q_yh)
18         ub ← LocalSearch(ip, ub, q_yh)
           /* scan */
19         for i ∈ [lb,ub] do
               /* ith point in partition ip */
20             p ← ip_i
21             if p within q then
22                 RQ ← RQ ∪ {p}
23             end
24         end
25     end
26 end
27 return RQ
```

Phase II: Boundary Refinement. After identifying the intersected partitions in the index lookup phase, the next step is to locate the bounds of the query on the sorted dimension within each partition. However, when a partition is fully contained within the query range, there is no need for boundary refinement, and all points within that partition can be

immediately returned. This is reflected in line 6 of Algorithm 2. Otherwise, when the query only partially intersects with a partition, there are two cases: (1) the partition is fully inside the x-dimension range. In this case, we employ a search technique to compute both the lower and upper bounds, and then copy all points within these bounds (reflected in lines 8-12 of the algorithm), (2) the partition is not fully contained within the x-dimension range. In this case, we compute the lower and upper bounds on the sorted y-dimension and then switch to the scan phase.

Typically, binary search is used as the search technique. In this paper, we propose replacing binary search with a learned model. Specifically, we use the RadixSpline index [39, 40] to efficiently search the sorted dimension of our data. RadixSpline consists of two components: a set of spline points and a radix table. The radix table is used to quickly identify the spline points for a given lookup key (in our case, the dimension over which the data is sorted). At lookup time, the radix table is consulted first to determine the range of spline points to examine. Next, these spline points are searched to locate the spline points surrounding the lookup key. Finally, linear interpolation is applied to predict the position of the lookup key within the sorted array.

Given the inherent error introduced by the RadixSpline (and generally, learned indexes), a local search (referred to as *LocalSearch*() in Algorithm 2) is necessary to find the exact lookup point, which, in our context, corresponds to the 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 possible scenarios. In the first scenario, the estimated value from the spline is lower than the true lower bound within the sorted dimension. Thus, we scan the partition upward until we reach the lower bound. Conversely, when the estimated value is higher than the actual lower bound, we scan the partition downward until we reach the lower bound while also materializing all encountered points. Consequently, in this scenario, the local search does not incur additional materialization costs (unless the estimated value exceeds the query's upper bound), as the points within the query bounds are materialized in any case.

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 onwards.

2.4 Point Query Processing

In this study, we also implement point queries, in keeping with the trend in recent research [64, 94]. A point query takes as input a query point q_p, and a set of geometric objects D. The query returns true if q_p is found within D, and false if it is not. Formally:

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

We use the partitioned and indexed dataset, D', from Algorithm 1, to speed up point query processing. Point query processing is outlined in Algorithm 3. First, we issue an *IndexLookup*() using a degenerate rectangle from the query point q_p. Since the point can only exist in one partition, we first check if the *IndexLookup*() phase produces any intersected partition. If no intersected partition is found, we immediately return false. Next, we search for the point within the intersected partition using a two-step process: (i) we estimate the location of the point in the y-dimension of the partition using the learned search technique, and (ii) we refine the result using the *SearchPoint*() procedure, which aims to mitigate the error introduced by the learned search technique, similarly to the *LocalSearch*() procedure used in range query processing.

Algorithm 3: Point Query Algorithm

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 ≠ ∅ 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
```

For the RadixSpline, there can now be three cases for the *SearchPoint*() procedure. First, if the estimated value from the spline is lower than the actual lower bound on the sorted dimension, we simply scan upward comparing the elements on the sorted dimension until we reach the actual lower bound. We then continue scanning on both dimensions until we find the query point or reach the upper bound of the partition. Note that for two points to be considered equal, their values should match on both dimensions. Second, if the estimated value is higher than the upper bound on the sorted dimension, we scan downward comparing the values on the sorted dimension until we reach the upper bound. We then continue scanning downward and comparing on both dimensions until we find the query point or reach the lower bound of the partition. Third, if the estimated value matches the query point's value on the search dimension, we perform an upward scan, comparing on both dimensions, to locate the query point. If the partition's upper bound is reached without finding the point, we then continue with a downward scan, again comparing on both dimensions, until we locate the query point or reach the partition's lower bound.

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

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 of query point q_p. Formally:

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

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* [55] approach, which is commonly used in popular database systems such as Oracle Spatial [32]. Note that using the *filter and refine* approach is also prevalent in many *recent* research works where various spatial queries (e.g., kNN queries, distance queries, and spatial join queries) are first decomposed to range queries as a preliminary filter, followed by query-specific refinement [25, 46, 47, 64].

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 [6] and then handle the edge cases of the poles and the 180th meridian.

Algorithm 4: Distance Query Algorithm

```
Input  : D′: partitioned and indexed input dataset; q_p: a query point; d: distance
Output : DQ: a set of all points in D′ within distance d of q_p
DQ ← {}
/* Get minimum bounding rectangle (mbr) of the circle */
mbr ← GetMBR(q_p, d)
/* Filter using mbr */
RQ ← RangeQuery(D′, mbr)
/* Refine */
for p ∈ RQ do
    if WithinDistance(p, q_p, d) then
        DQ ← DQ ∪ {p}
    end
end
return DQ
```

However, currently, we only use 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. One way to improve the efficiency is to break the bounding box into two parts, one on either side of the 180th meridian. We leave this optimization for future work. After materializing all the points within the MBR of q and d, we refine this candidate set of points. To that end, we compute the Haversine distance between the query point q and each of the candidate points and add to the final result all the points that are found to be within the specified distance d.

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:

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

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* [55] 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 casted 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.

3 EVALUATION

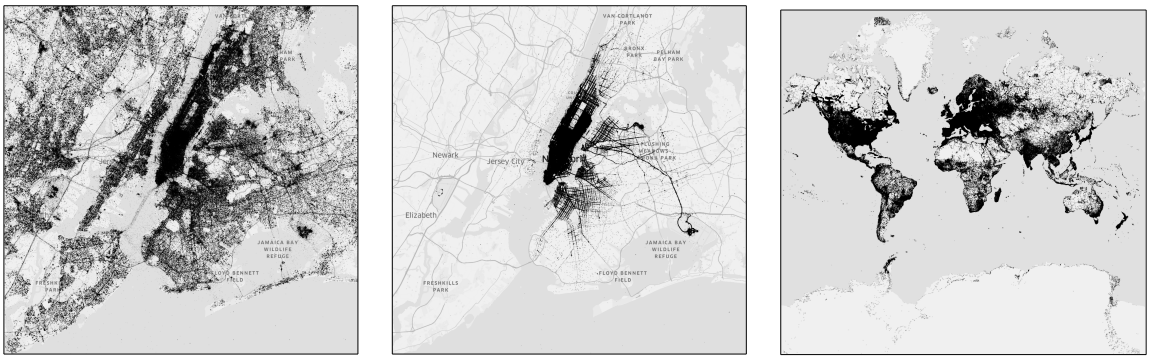
3.1 Experimental Setup

Hardware Configuration. All experiments were performed single-threaded on an Ubuntu 18.04 machine equipped with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 10 cores, 3.00 GHz turbo)[2] and 256 GB DDR3 RAM. We use the *numactl*

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

Algorithm 5: Join Query Algorithm

```
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
JQ ← {}
for polygon ∈ polygons do
    /* Get minimum bounding rectangle (mbr) of the polygon */
    mbr ← GetMBR(polygon)
    /* Filter using MBR */
    RQ ← RangeQuery(D, mbr)
    contained ← {}
    /* Refine */
    for p ∈ RQ do
        if Contains(polygon, p) then
            contained ← contained ∪ {p}
        end
    end
    JQ ← JQ ∪ {contained}
end
return JQ
```



(a) Twitter **(b) Taxi Trips** **(c) OSM**

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.

command to bind the thread and memory to one node to avoid NUMA effects. CPU scaling was also disabled during benchmarking using the *cpupower* command.

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.

3.2 Datasets and Queries

For evaluation, we used three datasets, the New York City Taxi Rides dataset [53] (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 [85] and contains 83 million tweets. The OSM dataset is taken from [58] 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

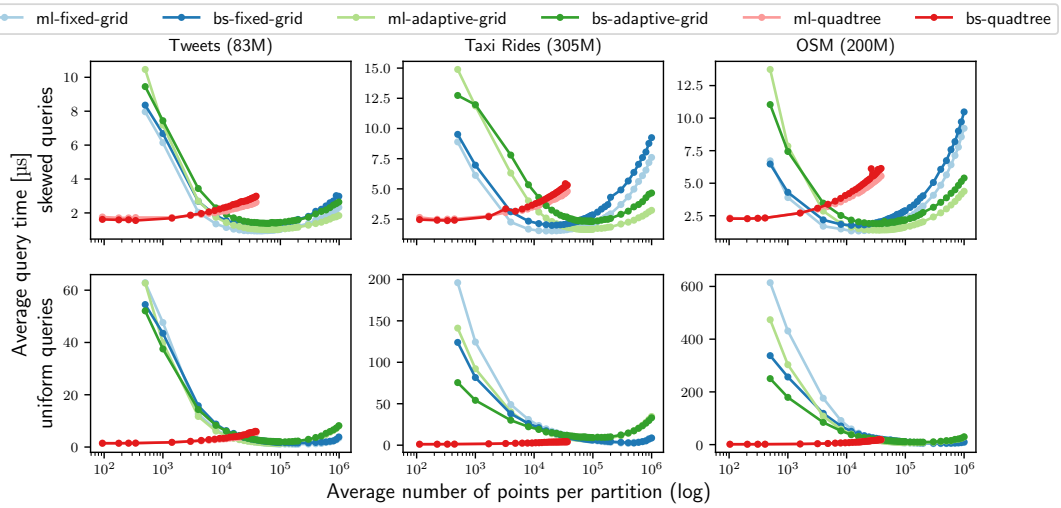


Fig. 4. Range Query Configuration - ML vs. BS for low selectivity (0.00001%).

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, in the case of 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).

3.3 Baselines

Firstly, we compare the performance of learned indexes and binary search as search techniques within a partition. Furthermore, we compare our implementation of the learned indexes with the two best-performing indexes from earlier studies [61, 62], 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. The source code used in this work is available on GitHub[3]. Additionally, we will open source the implementation of the learned indexes and the query workloads with the camera-ready version of this work. Given the popularity of machine-learned indexes in current research, we believe that our implementations will be useful in evaluating many influential works to come in the near future.

[3] https://github.com/varpande/learnedspatial

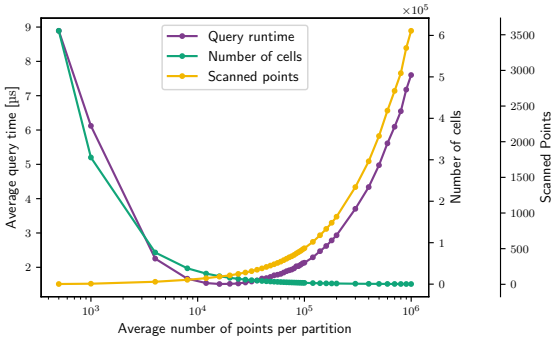


Fig. 5. Effect of the number of cells and scanned points for fixed-grid on Taxi Trip dataset for skewed queries (0.00001% selectivity).

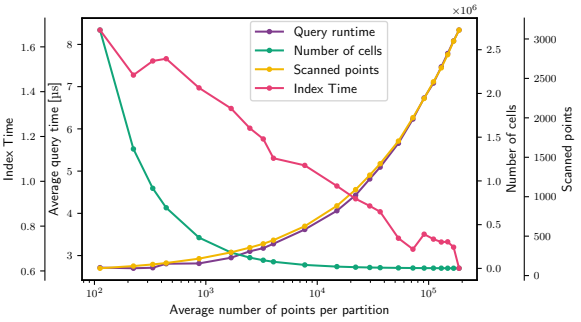


Fig. 6. Effect of the number of cells and scanned points for Quadtree on Taxi Trip dataset for skewed queries (0.00001% selectivity).

	Taxi Trips (Skewed Queries)						Taxi Trips (Uniform Queries)					
	Fixed		Adaptive		Quadtree		Fixed		Adaptive		Quadtree	
Selectivity (%)	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

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

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 has focused on learning from the data and the query workload. The essential idea behind learning from both the data and the query workload is that a particular use case can be instance-optimized [9, 41]. 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 (to cover a broad spectrum, we tune the indexes using queries with low and high selectivity) for both skewed and uniform queries. We omit the results for tuning the indexing techniques for the rest of the queries (point, distance, and join queries) as they are similar to the ones for range queries.

Figure 4 shows the effect of tuning when the indexes are tuned for the lowest selectivity workload for the two query types. It can be seen in the figure that it is essential to tune the grid indexing techniques for a particular workload. Firstly, they are *highly* susceptible to the size of the partition. As the size of the partition increases, we notice an improvement in the performance until a particular partition size is reached that corresponds to the optimal performance. After this point, increasing the size of the partitions only degrades the performance. This shows that determining the optimal partition is crucial for optimal performance. For example, fixing the partition size of fixed-grid to 100 points per partition (a fairly common default value of the leaf size for an index in many open-source spatial libraries) results in up to *300× worse* performance compared to

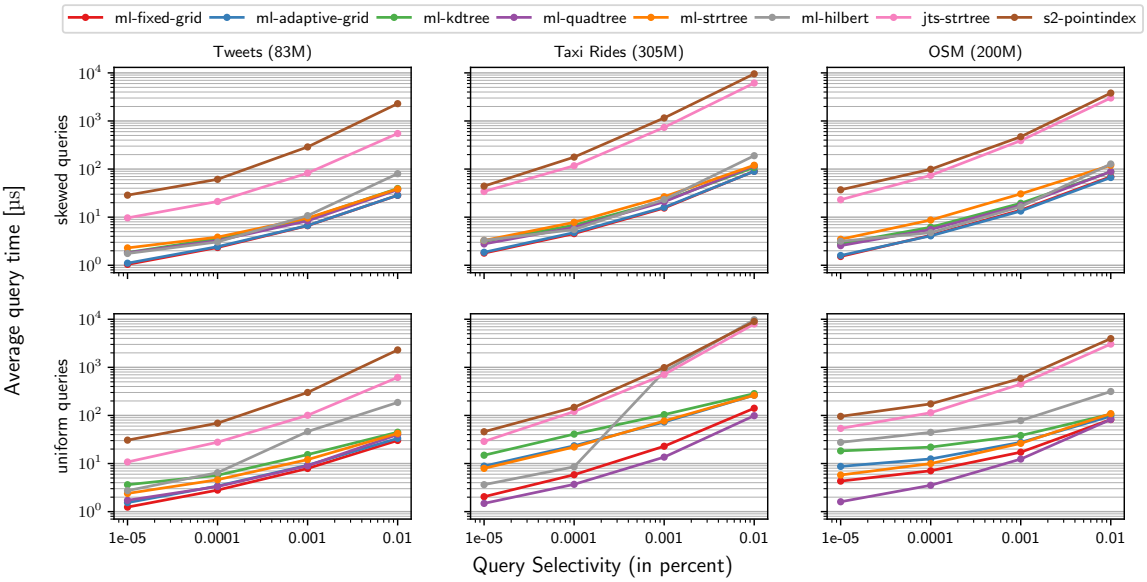


Fig. 7. Range Query Runtime - Total range query runtime on skewed and uniform queries for the three datasets.

the optimal partition size. It can be seen that, usually, for grid (single-dimension) indexing techniques, the optimal partition sizes are much larger compared to indexing techniques that filter on both dimensions (only Quadtree is shown in the figure but the same holds for the other indexing techniques we have covered in this work; we do not show the other trees because the curve is similar for them). Due to the large partition sizes in grid indexing techniques, we notice a large increase in performance while using a learned index compared to binary search. This is especially evident for skewed queries (which follow the underlying data distribution). We encountered a speedup from 11.79% to 39.51% compared to 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. Learned indexes do not help much for indexing techniques that filter on both dimensions. In contrast, as shown in Table 1, the performance of Quadtree (and STRtree) dropped in many cases. The reason is that the optimal partition size for these techniques is very low (less than 1,000 points per partition for most configurations). The refinement cost for the learned indexes is an overhead in such cases. The k-d tree, on the other hand, contains more points per partition (from 1200 to 7400) for the optimal configuration for Taxi Trips and OSM datasets, and thus the learned indexes perform faster by 2.43% to 9.17% than binary search. For the Twitter dataset, the optimal configuration contains less than 1200 points per partition, and we observed a similar drop in performance using learned indexes.

Figure 5 shows the effect of the number of cells and the number of points that are scanned in each partition on the query runtime for fixed-grid on Taxi Trips dataset for the lowest selectivity. As the number of points per partition increases (i.e., there are fewer partitions), the number of cells decreases. At the same time, the number of points that need to be scanned for the query increases. The point where these curves meet is the optimal configuration for the workload, which corresponds to the lowest query runtime. For tree structures, the effect is different. Figure 6 shows that the structures that filter on both dimensions do most of the pruning in the index lookup. The dominating cost in these structures is the number of points scanned within the partition, and the query runtime is directly proportional to this number. To minimize the number of points scanned, they do most of the pruning during index lookup, which requires more partitions

(i.e., fewer points per partition). Tree-based structures pay more during the index lookup phase, which requires chasing pointers (random access) during the index lookup and leads to more cache misses. However, once the desired partition is reached, these index structures scan very few points, as most of the partitions qualify for the query.

Key Takeaways. Tuning partition sizes is crucial for grid-based indexing techniques. The optimal size of grid partitions is typically large, which enables fast searches within the partitions using learned models. In contrast, tree-based indexes do not gain as much from learned models because optimal partition sizes are typically small.

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. For skewed queries, fixed-grid is 1.23× to 1.83× 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×) and OSM dataset (2.68×) 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 [34], where the authors compare the Quadtree with the R*-tree and the Pyramid-Technique.

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× to 43.27× faster than the JTS STRtree. Fixed-grid is also from 24.34× to 53.34× faster than S2PointIndex. Quadtree, on the other hand, is from 6.26× to 33.99× faster than JTS STRtree, and from 17.53× to 41.91× faster than S2PointIndex. Note that the index structures in the libraries are not tuned and are taken as is out of the box with default values. The poor performance of S2PointIndex is because it works on top of the Hilbert curve values and is not optimized for range queries. S2PointIndex and the learned linearized Hilbert curve index counterpart are rather similar in nature. Both use 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. T he 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. To avoid such a case, S2PointIndex decomposes the query rectangle into four parts to reduce the overlap with the curve. However, it still scans many superfluous points.

Key Takeaways. Fixed-grid performs the best across all queries, except for uniform queries on Taxi and OSM datasets. This is because it avoids random pointer chasing and utilizes fast learning-enhanced search within its large partitions. Quadtree outperforms fixed-grid for uniform queries on Taxi and OSM because these queries intersect with fewer partitions and only traverse the sparse, low-depth region of the index. Finally, the linearized Hilbert curve does not perform well in most cases because the queries need to scan a large portion of the curve.

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

	Taxi Rides		OSM	
Indexing	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

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. S2PointIndex in the Google S2 library, on the other hand, 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 is also consistent with real-world workloads, where it is more common to search for an existing point in the dataset, such as retrieving metadata for a specific restaurant location.

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×, 2.27×, and 1.51× faster that the closest tree-based competitor, kdtree, across Tweets, Rides, and OSM datasets. However, the performance difference of fixed-grid with adaptive-grid are marginal. It is 1.37×, 1.1×, 1.04× faster than the adaptive-grid across the Tweets, Rides, and OSM datasets. Lastly, fixed-grid is also 2.44×, 2.56×, 2.79× 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

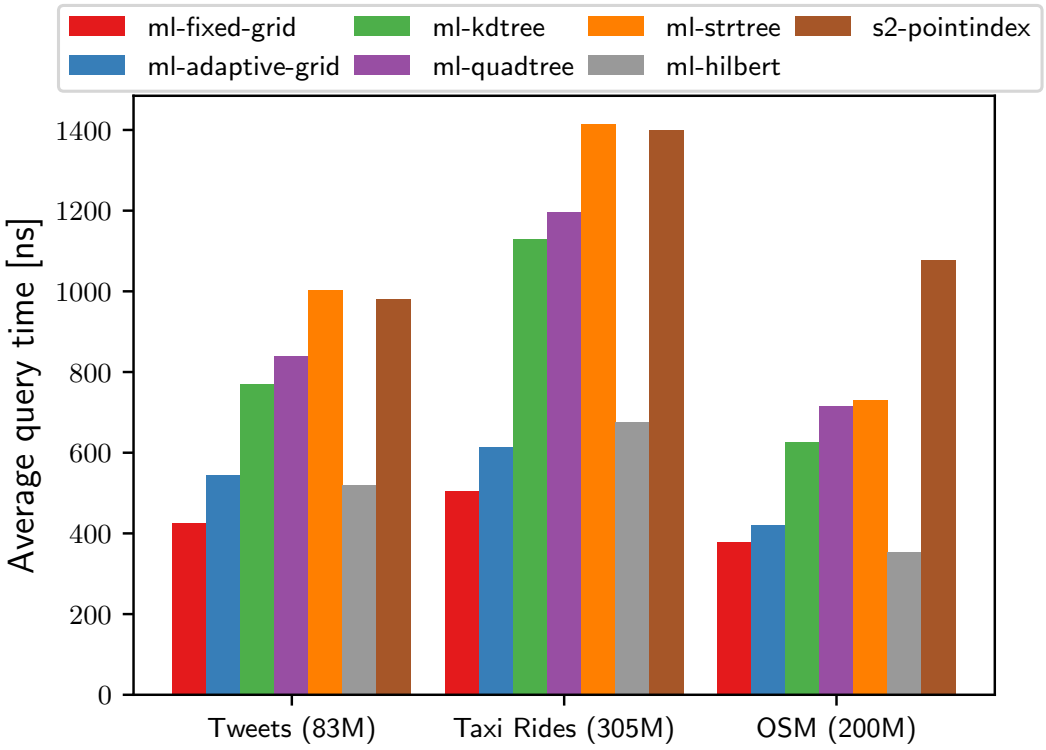


Fig. 8. Point Query Performance - Total point query runtime for skewed queries on the three datasets.

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).

Key Takeaways. Fixed-grid performs the best across all queries. The linearized Hilbert curve is highly competitive for point queries since it only requires searching for a single point on the curve. This is in contrast to range queries where a large portion of the curve needs to be scanned.

3.6 Distance Query Performance

We implemented the distance query using the filter and refine [55] approach (see Section 2.5), which is the norm in spatial databases such as Oracle Spatial [32] and PostGIS [63]. 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. 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.

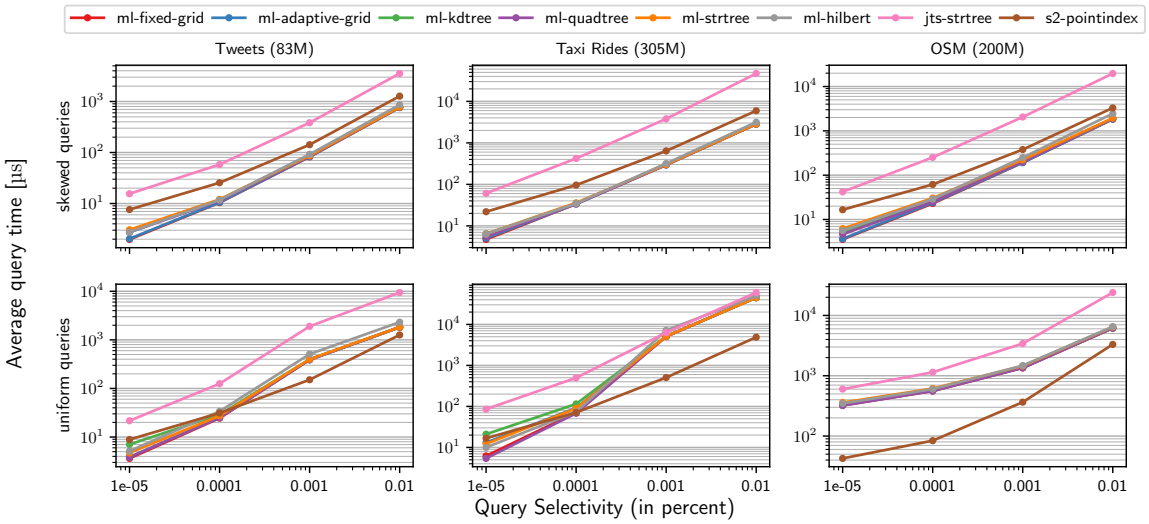


Fig. 9. Distance Query Performance - Total distance query runtime on skewed and uniform queries for the three datasets.

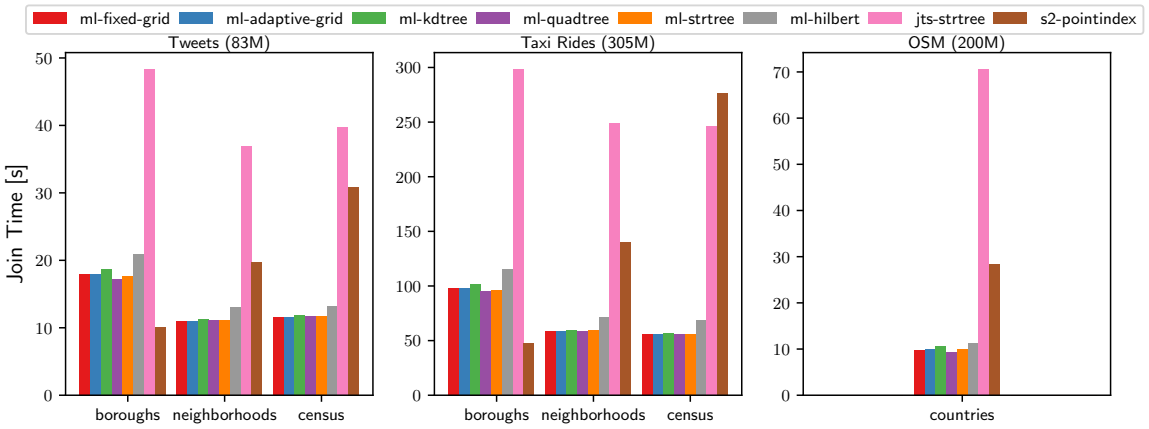


Fig. 10. Join Query Performance - Total join query runtime for the three datasets.

The second observation is that S2PointIndex outperforms most of the 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× to 170× 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 on the other hand has optimizations for distance queries, where it carefully increases the radius of the internal data structure called S2Cap (a circular disc with a center and a radius) to visit the Hilbert curve. It does not explicitly rely on the Haversine distance computation but works similarly to a range query with a radius. The S2PointIndex first searches for the point in the Hilbert curve

using the query points and then increases the radius along the Hilbert curve until the query radius is satisfied. Therefore, the distance query using the S2PointIndex for uniform queries on the OSM dataset is from 1.91× to 7.75× faster than the learned indexes. This effect does not reflect 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 fairer since both the learned indexes and JTS STRtree deploy the *filter and refine* approach to evaluate distance queries. Fixed-grid is from 1.33× to 11.92× faster than JTS STRtree.

Key Takeaways. The computation of Haversine distance dominates the cost of distance queries. While learned search offers significant performance gains for lower selectivity queries and for queries with skewed distributions, its advantages diminish for queries with higher selectivity and uniform distributions. These queries produce a large number of points for refinement (i.e., Haversine distance computations). As a result, in many cases, S2PointIndex outperforms learned indexes.

3.7 Join Query Performance

For join queries, we utilized 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 polygonal objects. 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 utilize the PreparedGeometry[4] abstraction, to index line segments of all individual polygons, which helps in accelerating the refinement check.

We used three different polygonal datasets for the join query 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 approximately 40 thousand polygons, respectively. For the OSM dataset, we perform the join using the Countries dataset which consists of 255 country boundaries. 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. It can be observed in the figure that most of the learned indexes are similar in join query 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 is in conformance to earlier studies [61, 62], 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 [36].

It can also be seen in the figure that the learned indexes are considerably faster than JTS STRtree and S2ShapeIndex for the join query. Fixed-grid, for example, is 1.81× to 2.69× faster than S2ShapeIndex and 2.7× to 3.44× faster than JTS STRtree for the Tweets dataset across all three polygonal datasets. Similarly, for the Taxi Rides dataset, fixed-grid is 2.39× to 4.96× faster than S2ShapeIndex and 3.017× to 4.49× faster than JTS STRtree. Finally, for the OSM dataset, it is 2.89× faster than S2ShapeIndex and 7.311× faster than JTS STRtree.

Key Takeaways. The *filter* phase of the join query produces a large number of points for the *refinement* phase, which involves computationally-expensive point-in-polygon tests. The computational overhead of the point-in-polygon tests diminishes the benefits gained from the fast *filter* phase. As a result, the query performance is similar to that of the distance query, where the dominant cost is the Haversine distance computation.

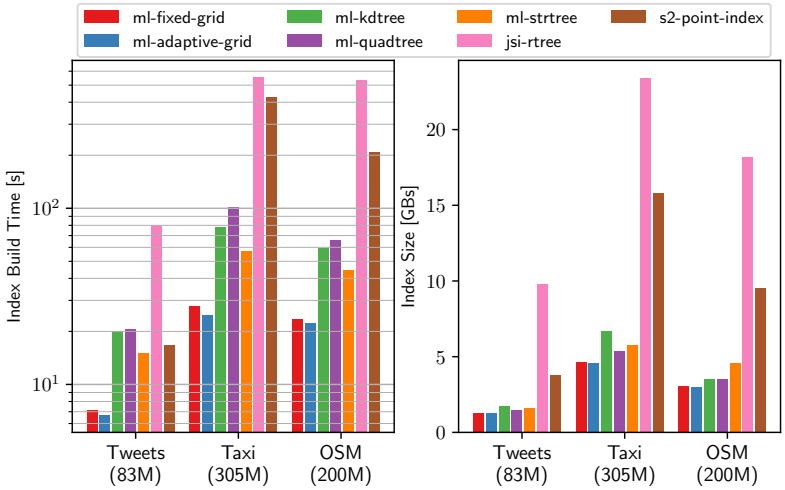


Fig. 11. Indexing Costs - Index build times and sizes for the three datasets.

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×, 2.05×, and 1.90× 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× to 15.36× faster to build than S2PointIndex, and from 11.09× to 19.74× faster to build than JTS STRtree. It also consumes less memory than S2PointIndex (from 3.04× to 3.4×) and JTS STRtree (from 4.96× to 8.024×). 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 consume less space compared to tree-based indexes. Optimally-tuned grid-based indexes use larger partitions, which results in a small total number of partitions. In contrast, tree-based indexes produce a large number of smaller partitions. Since we embed learned models within each partition, tree-based indexes have a higher build time and are larger in size, as they maintain more learned models.

4 RELATED WORK

Recent work by Kraska et al. [42] proposed the idea of replacing traditional database indexes with learned models. Since then, there has been a corpus of work on extending the ideas of the learned index to spatial and multidimensional data.

[4] https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/prep/PreparedGeometry.html

Learned Multidimensional Indexing and Partitioning. Flood [51] is an in-memory read-optimized multidimensional index that organizes the physical layout of d-dimensional data by dividing each dimension into some number of partitions, which forms a grid over the d-dimensional space and adapts to the data and query workload. Similarly to Flood, our implementation of the grid indexes partitions the data using a grid across $d-1$ dimensions and uses the last dimension as the sort dimension. Tsunami[10, 11] is an improvement over Flood, designed to efficiently handle correlated and skewed data. Machine learning techniques have also been applied to reduce the I/O cost for disk-based multidimensional indexes. Qd-tree [99] uses reinforcement learning to construct a partitioning strategy that minimizes the number of disk-based blocks accessed by a query. The ZM-index [94] combines the standard Z-order space-filling curve with the Recursive-Model Indexes (RMI) proposed by Kraska et al. [42] by mapping multidimensional values into a single-dimensional space that is learnable using models. The ML-index [8] combines the ideas of iDistance [31] and RMI [42] to support range and KNN queries. There are also efforts to augment existing indexes with lightweight models to accelerate range and point queries [28]. Machine learning has also been applied to various other aspects of data management [35, 56, 66, 70, 71, 98].

Learned Spatial Indexes and Algorithms. LISA [47] 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 [57], the authors propose an instance-optimized Z-curve index. They present alternate ordering and partitioning of the Z-curve and propose two greedy heuristics that learn the most effective ordering for a particular workload and dataset.

In traditional R-trees, the branches visited for a particular range query depend on the overlap of the query with multiple children nodes. In [2], the authors propose using the overlap ratio (the required number of leaf nodes / actually visited leaf nodes) to identify high-overlap queries. They also build an AI tree that uses uniquely assigned IDs to the R-tree leaf nodes as class labels for multi-label classification. At runtime, their technique identifies queries with high overlap and uses the AI tree for them; otherwise, it falls back to the regular R-tree for low-overlap queries. The AI tree helps minimize the number of visited leaf nodes. In [93], the authors work on SAMs (i.e., spatial objects with extent such as linestrings and polygons). The authors compute the Z-curve extent (minimum and maximum of the Z-interval) of the MBR enclosing the spatial object, sort the geometries based on the Z-address interval and build a hierarchical tree-like structure where internal nodes contain the linear regression model and an array of pointers to child nodes, while the leaf node contains the linear regression model and an array of actual data along with the MBR of the data. This is in contrast to the ZM-index which uses Z-curve values of the points instead of spatial object with extents, as well as the Hibert-curve index that we use to store the points. SPRIG [104] is a spatial interpolation function-based grid index. The authors use spatial bilinear interpolation function to predict the position of the query points and then use a local binary search to refine the result. Spatial Join Machine Learning (SJML) [90, 91] is a machine learning-based query optimizer for distributed spatial joins. It consists of three levels: (1) the first level builds the cardinality estimates model that learns the result size of the spatial join, (2) the second level combines the predicted result size with other dataset characteristics to build a separate model which predicts the number of geometric comparison operations, and (3) builds a classification model which is able to predict the best join algorithm. In [25], the authors exploit the fact that the R tree can be constructed using multiple splitting strategies, e.g., linear split, quadratic split, R*-tree split etc. They run a microbenchmark to find that the query performance varies for various query workloads and datasets. They identify that ChooseSubtree and Split operations for the tree construction can be considered as sequential decision-making problems. They model them as two Markov Decision Processes (MDP) and use reinforcement learning to learn the model for the two operations.

Machine Learning techniques on spatial data have also been applied to various scenarios such as spatio-textual queries [12], social media data [26], passage retrieval [95], and streaming [102], and other areas [35, 56, 70, 98]. There also exists various surveys that cover in-depth various works that apply machine learning to spatial data [67–69].

5 CONCLUSIONS AND FUTURE WORK

In this work, we implemented learning-enhanced variants of six classical spatial indexes. We found that, in most cases, the fixed-grid is the best-performing learning-enhanced index and also the simplest one to implement. Next, we summarize the results based on the four queries covered in this work and discuss avenues for future work.

Range Queries. Recent advancements in applying machine learning to databases have focused on creating instance-optimized [9, 41] versions of various components, including index structures. Motivated by this, in Section 3.4.1 we showed that the performance of various index structures fluctuates depending on the query workload and dataset. We demonstrated that tuning the index structures based on both the dataset and the query workloads is essential to obtain optimal performance. Additionally, we showed that using learned models instead of binary search improves performance by 11-39%.

We also found that learned models do not help much in the case of tree-based index structures. The performance gains were minimal, ranging from 2% to 9%, and in some instances, learned models even resulted in worse performance. We showed that fixed-grid was the best-performing index for range queries, outperforming the closest tree-based competitor by a factor of 1.23× up to 1.83×. This is in *contrast* to traditional knowledge that grid-based structures suffer from skewed partitions. We argue that when all data can be indexed and stored in main memory, grid-based index structures, when tuned for optimal partition sizes, outperform tree-based index structures. We also showed that fixed-grid is from 8.67× to 43.27× faster than JTS STRtree and from 4.34× to 53.34× faster than S2PointIndex. Additionally, we addressed the reasons behind the poor performance of Hilbert curve-based approaches, i.e., S2PointIndex, and learned linearized Hilbert curve-based index.

Point Queries. In Section 3.5, we again observe that fixed-grid is the best-performing index. It is 1.51× to 2.27× faster compared to the closest tree-based competitor, kdtree. We also showed that the learned index based on linearized Hilbert curve values is highly competitive, as point queries require searching for a specific point on the Hilbert curve. This is in contrast to range queries where it may end up scanning a lot of redundant points.

Distance Queries. The fixed-grid index has again the best performance for distance queries. However, we note that, as in the case of high selectivity range queries, the performance gains diminish with increasing selectivity, and the Haversine distance computation becomes the *dominant* cost. We also observe that S2PointIndex performs the best in the case of uniform queries in the OSM dataset. There are two reasons for this: (1) the filter phase produces a lot more candidate points in the case of uniform queries compared to other datasets and query workloads, and (2) S2PointIndex traverses the Hilbert curve in an efficient manner avoiding the overheads of the *filter and refine* approach.

Join Queries. The *filter* phase of the join, which internally issues a range query to the index structures using the bounding box of the polygons, exhibits the same performance as range queries. However, the *refinement* phase of the join is the most *dominant* cost. This leads to similar performance of the learned index structures and performance gains are limited. Currently, we utilize interval trees to index the polygons used in the join operation. In the future, we plan to use learned index structures, such as those proposed in [87] and [93], to index the polygons as well.

Future Work. Thus far, we have only studied the case where both the indexes and data fit into RAM. For disk-based use cases, the performance will likely be dominated by I/O, and the search within partitions will be less important. We expect the partition sizes to be performance-optimal when aligned with the physical page size. To reduce I/O, it will be crucial to eliminate any unnecessary points from the partitions. Therefore, we anticipate that using two-dimensional indexing will be the best approach for disk-based storage. For further discussion on this topic, we refer to LISA [47]. Our findings demonstrate that grid-based index structures outperform tree-based indexes by reducing random accesses and using efficient fast search over large partitions. However, the gains diminish when computationally expensive operations, such as Haversine distance computation and point-in-polygon tests, are required. To minimize the overall query runtime,

it is necessary to develop novel effective indexing methods for polygons (to minimize or eliminate point-in-polygon tests) and circles (to avoid Haversine distance computation). Moreover, in our work, we tuned the index structures using a manual process. In the future, our aim is to investigate the use of deep learning-based methods to tune the index structures for various types of queries, datasets, and workloads.

REFERENCES

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

[2] Abdullah Al-Mamun, Ch. Md. Rakin Haider, Jianguo Wang, and Walid G. Aref. 2022. The "AI + R" - tree: An Instance-optimized R - tree. In *23rd IEEE International Conference on Mobile Data Management, MDM 2022, Paphos, Cyprus, June 6-9, 2022*. 9–18. https://doi.org/10.1109/MDM55031.2022.00023

[3] Koichiro Amemiya and Akihiro Nakao. 2020. 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*. 1–9. https://doi.org/10.1109/NOMS47738.2020.9110436

[4] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517. https://doi.org/10.1145/361002.361007

[5] Jon Louis Bentley and Jerome H. Friedman. 1979. Data Structures for Range Searching. *ACM Comput. Surv.* 11, 4 (1979), 397–409. https://doi.org/10.1145/356789.356797

[6] Ilja N Bronshtein and Konstantin A Semendyayev. 2013. *Handbook of mathematics*. Springer Science & Business Media, Germany.

[7] Databricks Runtime [n.d.]. *Databricks Runtime 7.6*. https://docs.databricks.com/release-notes/runtime/7.6.html.

[8] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *2020 Conference on Extending Database Technology (EDBT)*.

[9] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 418–431. https://doi.org/10.1145/3448016.3457270

[10] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86. https://doi.org/10.14778/3425879.3425880

[11] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86. https://doi.org/10.14778/3425879.3425880

[12] Xiaofeng Ding, Yinting Zheng, Zuan Wang, Kim-Kwang Raymond Choo, and Hai Jin. 2022. A learned spatial textual index for efficient keyword queries. *Journal of Intelligent Information Systems* (2022), 1–25.

[13] Harish Doraiswamy and Juliana Freire. 2020. 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*. 1875–1885. https://doi.org/10.1145/3318464.3389774

[14] Harish Doraiswamy and Juliana Freire. 2020. A GPU-friendly Geometric Data Model and Algebra for Spatial Queries: Extended Version. *CoRR* abs/2004.03630 (2020). arXiv:2004.03630 https://arxiv.org/abs/2004.03630

[15] Harish Doraiswamy and Juliana Freire. 2022. GPU-Powered Spatial Database Engine for Commodity Hardware: Extended Version. *CoRR* abs/2203.14362 (2022). https://doi.org/10.48550/arXiv.2203.14362 arXiv:2203.14362

[16] Harish Doraiswamy and Juliana Freire. 2022. 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*. 2669–2681. https://doi.org/10.1109/ICDE53745.2022.00245

[17] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, A. B. Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. 2021. Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). 3796–3807. https://doi.org/10.1145/3459637.3481897

[18] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 1352–1363.

[19] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. 2017. 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*. 65–83. https://doi.org/10.1007/978-3-319-64367-0_4

[20] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (1974), 1–9. https://doi.org/10.1007/BF00288933

[21] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231. https://doi.org/10.1145/280277.280279

[22] Francisco García-García, Antonio Corral, Luis Iribarne, and Michael Vassilakopoulos. 2020. Improving Distance-Join Query processing with Voronoi-Diagram based partitioning in SpatialHadoop. *Future Gener. Comput. Syst.* 111 (2020), 723–740. https://doi.org/10.1016/j.future.2019.10.037

[23] Thanasis Georgiadis, Eleni Tzirita Zacharatou, and Nikos Mamoulis. 2023. APRIL: Approximating Polygons as Raster Interval Lists. *CoRR* abs/2307.01716 (2023). https://doi.org/10.48550/arXiv.2307.01716 arXiv:2307.01716

[24] David Gomes. 2019. *MemSQL Live: Nikita Shamgunov on the Data Engineering Podcast.* https://www.memsql.com/blog/memsql-live-nikita-shamgunov-on-the-data-engineering-podcast/.

[25] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proc. ACM Manag. Data* 1, 1 (2023), 63:1–63:26. https://doi.org/10.1145/3588917

[26] Na Guo, Yaqi Wang, Haonan Jiang, Xiufeng Xia, and Yu Gu. 2022. TALI: An Update-Distribution-Aware Learned Index for Social Media Data. *Mathematics* 10, 23 (2022), 4507.

[27] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984.* 47–57. https://doi.org/10.1145/602259.602266

[28] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. 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*, Bingsheng He, Berthold Reinwald, and Yingjun Wu (Eds.). https://drive.google.com/file/d/1c3uPyv9apCHWz2wcr1bYNB3JVgCKteKM/view?usp=sharing

[29] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. 2017. 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 Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings.* 123–142.

[30] David Hilbert. 1935. Über die stetige Abbildung einer Linie auf ein Flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes.* 1–2.

[31] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. IDistance: An Adaptive B+-Tree Based Indexing Method for Nearest Neighbor Search. *ACM Trans. Database Syst.* 30, 2 (June 2005), 364–397. https://doi.org/10.1145/1071610.1071612

[32] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. 2002. 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.* 546–557. https://doi.org/10.1145/564691.564755

[33] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. 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.* 715–730. https://doi.org/10.1145/3035918.3064049

[34] You Jung Kim and Jignesh M. Patel. 2007. 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.* 281–291. http://cidrdb.org/cidr2007/papers/cidr07p32.pdf

[35] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. LSI: a learned secondary index structure. In *aiDM '22: Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Philadelphia, Pennsylvania, USA, 17 June 2022*, Rajesh Bordawekar, Oded Shmueli, Yael Amsterdamer, Donatella Firmani, and Ryan Marcus (Eds.). ACM, 4:1–4:5. https://doi.org/10.1145/3533702.3534912

[36] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2020. Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins. In *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020.* 347–358. https://doi.org/10.5441/002/edbt.2020.31

[37] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2018. Adaptive Geospatial Joins for Modern Hardware. *CoRR* abs/1802.09488 (2018). arXiv:1802.09488 http://arxiv.org/abs/1802.09488

[38] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2018. Approximate Geospatial Joins with Precision Guarantees. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* 1360–1363. https://doi.org/10.1109/ICDE.2018.00150

[39] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. 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*, Rajesh Bordawekar, Oded Shmueli, Nesime Tatbul, and Tin Kam Ho (Eds.). ACM, 5:1–5:5. https://doi.org/10.1145/3401071.3401659

[40] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. 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.* 5:1–5:5. https://doi.org/10.1145/3401071.3401659

[41] Tim Kraska. 2021. Towards instance-optimized data systems. *Proc. VLDB Endow.* 14, 12 (2021), 3222–3232. https://doi.org/10.14778/3476311.3476392

[42] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. 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.* 489–504. https://doi.org/10.1145/3183713.3196909

[43] Jonathan K. Lawder and Peter J. H. King. 2000. Using Space-Filling Curves for Multi-dimensional Indexing. In *Advances in Databases, 17th British National Conferenc on Databases, BNCOD 17, Exeter, UK, July 3-5, 2000, Proceedings (Lecture Notes in Computer Science)*, Brian Lings and Keith G. Jeffery (Eds.), Vol. 1832. 20–35. https://doi.org/10.1007/3-540-45033-5_3

[44] Jonathan K. Lawder and Peter J. H. King. 2001. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *SIGMOD Rec.* 30, 1 (2001), 19–24. https://doi.org/10.1145/373626.373678

[45] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. 1997. 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*. 497–506. https://doi.org/10.1109/ICDE.1997.582015

[46] Jiangneng Li, Zheng Wang, Gao Cong, Cheng Long, Han Mao Kiah, and Bin Cui. 2023. Towards Designing and Learning Piecewise Space-Filling Curves. *Proc. VLDB Endow.* 16, 9 (2023), 2158–2171. https://www.vldb.org/pvldb/vol16/p2158-li.pdf

[47] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. https://doi.org/10.1145/3318464.3389703

[48] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. 2019. 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 (CEUR Workshop Proceedings)*, Vol. 2322. http://ceur-ws.org/Vol-2322/BMDA_3.pdf

[49] MongoDB 2013. *MongoDB Releases - New Geo Features in MongoDB 2.4.* https://www.mongodb.com/blog/post/new-geo-features-in-mongodb-24/.

[50] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Trans. Knowl. Data Eng.* 13, 1 (2001), 124–141. https://doi.org/10.1109/69.908985

[51] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. https://doi.org/10.1145/3318464.3380579

[52] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (1984), 38–71. https://doi.org/10.1145/348.318586

[53] NYCTaxiData 2019. *NYC Taxi and Limousine Commission (TLC) - TLC Trip Record Data.* https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[54] Oracle 2019. *Oracle Spatial and Graph Spatial Features.* https://www.oracle.com/technetwork/database/options/spatialandgraph/overview/spatialfeatures-1902020.html/.

[55] Jack A. Orenstein. [n.d.]. Redundancy in Spatial Databases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*. https://doi.org/10.1145/67544.66954

[56] Sachith Gopalakrishna Pai, Michael Mathioudakis, and Yanhao Wang. 2022. Towards an Instance-Optimal Z-Index. (2022).

[57] Sachith Gopalakrishna Pai, Michael Mathioudakis, and Yanhao Wang. 2022. Towards an Instance-Optimal Z-Index. (2022).

[58] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *Proc. VLDB Endow.* 11, 11 (2018), 1661–1673. https://doi.org/10.14778/3236187.3236213

[59] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. 2016. 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*. 2145–2148.

[60] Varun Pandey, Alexander van Renen, Andreas Kipf, Jialin Ding, Ibrahim Sabek, and Alfons Kemper. 2020. 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*. https://drive.google.com/file/d/1Q_kmSPhM86FeeZb8Kz196eNIn7uWcu8L/view?usp=sharing

[61] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. 2020. 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 (Lecture Notes in Computer Science)*, Vol. 12113. 157–174. https://doi.org/10.1007/978-3-030-59416-9_46

[62] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. 2021. How Good Are Modern Spatial Libraries? *Data Sci. Eng.* 6, 2 (2021), 192–208. https://doi.org/10.1007/s41019-020-00147-9

[63] PostGIS [n.d.]. *PostGIS.* http://postgis.net/.

[64] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 11 (2020), 2341–2354. http://www.vldb.org/pvldb/vol13/p2341-qi.pdf

[65] Keven Richly. 2019. Optimized Spatio-Temporal Data Structures for Hybrid Transactional and Analytical Workloads on Columnar In-Memory Databases. In *Proceedings of the VLDB 2019 PhD Workshop, co-located with the 45th International Conference on Very Large Databases (VLDB 2019), Los Angeles, California, USA, August 26-30, 2019 (CEUR Workshop Proceedings)*, Vol. 2399. http://ceur-ws.org/Vol-2399/paper10.pdf

[66] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *Proc. VLDB Endow.* 16, 7 (2023), 1749–1762. https://www.vldb.org/pvldb/vol16/p1749-sabek.pdf

[67] Ibrahim Sabek and Mohamed F. Mokbel. 2019. Machine Learning Meets Big Spatial Data. *Proc. VLDB Endow.* 12, 12 (2019), 1982–1985. https://doi.org/10.14778/3352063.3352115

[68] Ibrahim Sabek and Mohamed F. Mokbel. 2020. Machine Learning Meets Big Spatial Data. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. 1782–1785. https://doi.org/10.1109/ICDE48307.2020.00169

[69] Ibrahim Sabek and Mohamed F. Mokbel. 2021. 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*. 5–8. https://doi.org/10.1109/MDM52706.2021.00014

[70] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. (2022).

[71] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions? *Proc. VLDB Endow.* 16, 3 (2022), 532–545. https://www.vldb.org/pvldb/vol16/p532-sabek.pdf

[72] Majid Saeedan and Ahmed Eldawy. 2022. Spatial parquet: a column file format for geospatial data lakes. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2022, Seattle, Washington, November 1-4, 2022*, Matthias Renz and Mohamed Sarwat (Eds.). ACM, 102:1–102:4. https://doi.org/10.1145/3557915.3561038

[73] Majid Saeedan and Ahmed Eldawy. 2022. Spatial parquet: a column file format for geospatial data lakes. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2022, Seattle, Washington, November 1-4, 2022*. 102:1–102:4. https://doi.org/10.1145/3557915.3561038

[74] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). 23–34. https://doi.org/10.1145/582095.582099

[75] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. 2018. 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*. 425–436. https://doi.org/10.1109/ICDE.2018.00046

[76] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 307–322. https://doi.org/10.1145/3183713.3196893

[77] Ruby Y. Tahboub and Tiark Rompf. 2016. On supporting compilation in spatial query engines: (vision paper). In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*. 9:1–9:4. https://doi.org/10.1145/2996913.2996945

[78] Ruby Y. Tahboub and Tiark Rompf. 2020. 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*. 2103–2118. https://doi.org/10.1145/3318464.3389701

[79] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB* 9, 13 (2016), 1565–1568.

[80] Konstantinos Theocharidis, John Liagouris, Nikos Mamoulis, Panagiotis Bouros, and Manolis Terrovitis. 2019. SRX: efficient management of spatial RDF data. *VLDB J.* 28, 5 (2019), 703–733. https://doi.org/10.1007/s00778-019-00554-z

[81] Theodoros Toliopoulos, Nikodimos Nikolaidis, Anna-Valentini Michailidou, Andreas Seitaridis, Anastasios Gounaris, Nick Bassiliades, Apostolos Georgiadis, and Fotis Liotopoulos. 2020. 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 (Lecture Notes in Business Information Processing)*, Vol. 385. 58–75. https://doi.org/10.1007/978-3-030-50316-1_4

[82] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. 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*, Farnoush Banaei Kashani, Goce Trajcevski, Ralf Hartmut Güting, Lars Kulik, and Shawn D. Newsam (Eds.). ACM, 516–519. https://doi.org/10.1145/3347146.3359343

[83] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. 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*. 516–519. https://doi.org/10.1145/3347146.3359343

[84] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2020. A Two-level Spatial In-Memory Index. *CoRR* abs/2005.08600 (2020). arXiv:2005.08600 https://arxiv.org/abs/2005.08600

[85] TweetsDataset 2020. *Tutorials: Filtering Tweets by location.* https://developer.twitter.com/en/docs/tutorials/filtering-tweets-by-location.

[86] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. 2017. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. *PVLDB* 11, 3 (2017), 352–365. https://doi.org/10.14778/3157794.3157803

[87] Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. 2021. 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*. http://cidrdb.org/cidr2021/papers/cidr2021_paper19.pdf

[88] Eleni Tzirita Zacharatou, Darius Sidlauskas, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2019. 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*. 139–148. https://doi.org/10.1145/3347146.3359077

[89] Uber. 2018. Uber Newsroom: 10 Billion. https://www.uber.com/newsroom/10-billion/.

[90] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. 2021. A Learned Query Optimizer for Spatial Join. In *SIGSPATIAL '21: 29th International Conference on Advances in Geographic Information Systems, Virtual Event / Beijing, China, November 2-5, 2021*, Xiaofeng Meng, Fusheng Wang, Chang-Tien Lu, Yan Huang, Shashi Shekhar, and Xing Xie (Eds.). 458–467. https://doi.org/10.1145/3474717.3484217

[91] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. 2022. Towards a Learned Cost Model for Distributed Spatial Join: Data, Code & Models. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*, Mohammad Al Hasan and Li Xiong (Eds.). 4550–4554. https://doi.org/10.1145/3511808.3557712

[92] Tin Vu, Ahmed Eldawy, Vagelis Hristidis, and Vassilis J. Tsotras. 2021. Incremental Partitioning for Efficient Spatial Data Analytics. *Proc. VLDB Endow.* 15, 3 (2021), 713–726. https://doi.org/10.14778/3494124.3494150

[93] Congying Wang and Jia Yu. 2022. GLIN: A Lightweight Learned Indexing Mechanism for Complex Geometries. *CoRR* abs/2207.07745 (2022). https://doi.org/10.48550/arXiv.2207.07745 arXiv:2207.07745

[94] H. Wang, X. Fu, J. Xu, and H. Lu. 2019. Learned Index for Spatial Queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. 569–574.

[95] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. 2022. LIDER: An Efficient High-dimensional Learned Index for Large-scale Dense Passage Retrieval. *Proc. VLDB Endow.* 16, 2 (2022), 154–166. https://www.vldb.org/pvldb/vol16/p154-wang.pdf

[96] Christian Winter, Andreas Kipf, Christoph Anneser, Eleni Tzirita Zacharatou, Thomas Neumann, and Alfons Kemper. 2021. 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*. 169–180. https://doi.org/10.5441/002/edbt.2021.16

[97] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. 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*. 1071–1085.

[98] Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. 2023. 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*. 234–246. https://doi.org/10.48786/edbt.2023.19

[99] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar F. Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. https://doi.org/10.1145/3318464.3389770

[100] Simin You, Jianting Zhang, and Le Gruenwald. 2015. 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*. 34–41.

[101] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. 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*. 70:1–70:4.

[102] Tong Yu, Guanfeng Liu, An Liu, Zhixu Li, and Lei Zhao. 2022. LIFOSS: a learned index scheme for streaming scenarios. *World Wide Web* (2022), 1–18.

[103] Haitao Yuan, Guoliang Li, and Zhifeng Bao. [n.d.]. Route Travel Time Estimation on A Road Network Revisited: Heterogeneity, Proximity, Periodicity and Dynamicity. ([n. d.]).

[104] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. 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*, Erik Hoel, Dev Oliver, Raymond Chi-Wing Wong, and Ahmed Eldawy (Eds.). 96–105. https://doi.org/10.1145/3469830.3470892