Hands-off Model Integration in Spatial Index Structures (Regular Papers)

Ali Hadian
Imperial College London
hadian@imperial.ac.uk

Ankit Kumar
IIT Delhi
mt1170727@iitd.ac.in

Thomas Heinis
Imperial College London
t.heinis@imperial.ac.uk

ABSTRACT

Spatial indexes are crucial for the analysis of the increasing amounts of spatial data, for example generated through IoT applications. The plethora of indexes that has been developed in recent decades has primarily been optimised for disk. With increasing amounts of memory even on commodity machines, however, moving them to main memory is an option. Doing so opens up the opportunity to use additional optimizations that are only amenable to main memory.

In this paper we thus explore the opportunity to use lightweight machine learning models to accelerate queries on spatial indexes. We do so by exploring the potential of using interpolation and similar techniques on the R-tree, arguably the most broadly used spatial index. As we show in our experimental analysis, the query execution time can be reduced by up to 60% while simultaneously shrinking the index's memory footprint by over 90%.

AIDB Workshop Reference Format:
Ali Hadian, Ankit Kumar, Thomas Heinis. Hands-off Model Integration in Spatial Index Structures. *AIDB* 2020.

1. INTRODUCTION

Spatial data is generated and in need of analysis in many different applications. The proliferation of GPS devices, integrated in mobile devices, along with the growing use of high precision sensors, e.g., LIDAR, to map our surroundings has lead to a deluge of spatial datasets.

Efficiency and speed are key to analyse data in general and spatial data in particular as having timely, actionable insight is crucial. No wonder analytics has moved into main memory where feasible and while spatial datasets are big in size, many of them still fit into main memory, particularly in case of today's machines where main memory is abundantly available.

In addition to simply move the data into main memory, recent research also has developed a novel class of indexes that use machine learning methods at their core [18]. More precisely, simple machine learning models are used in traditional indexes to better approximate the specific data distribution of the data set at hand.

Most notably, machine learning models like interpolation have been used in B+-Trees and similar indexes with considerable success in that the models helped to accelerate queries by 50% [13] - a considerable improvement for data structures, such as the B+-Tree which have been carefully optimised and tuned for decades [10].

In this paper, we consequently study the use of machine learning techniques in spatial indexes. Straightforward reuse of the techniques used to optimise the B+-Tree [13] is unfortunately not possible as multidimensional data with multiple correlated dimensions is more complex. The higher complexity, however, also gives us more degrees of freedom to adapt the index structure. Our suggested solution accelerates the spatial data structure using a simple predictive model, but at the same time re-arranges the physical layout such that the predictive models are the most effective. As we show, use of interpolation techniques accelerates queries on spatial data in main memory by up to 1.8X (4.2X on multi-threaded execution) for a variety of datasets.

The remainder of the paper is organised as follows. We first discuss related work in Section2 and then motivate the idea of using simple machine learning models in spatial indexes in Sections 3 and 5. We then discuss how we implemented the machine learning models in the spatial indexes in Section 4. In Section 6 we discuss our experimental setup and, more importantly, the experimental results. We finally conclude the paper in Section 7.

2. RELATED WORK

A plethora of spatial indexes [6] has been developed in recent decades. In the following discussion, we focus on relevant work related to learned indexes for spatial data as well as to interpolation used for other, lower-dimensional indexes.

Learned index structures. In the past few years, tons of research have been done on using machine learning to optimize database systems [17], most notably learned index structures [18] in which a machine learning model replaces traditional index structures (such as B-tree and hash tables) for locating the physical position of records. Several learned models have been suggested so different indexing problems, such as range indexing [5, 7], bloom filters [25, 24], distributed indexing [20], and handling updates [3, 12].

Hybrid learned indexes. Some hybrid methods have been suggested to integrate machine learning into a well-

2nd International Workshop on Applied AI for Database Systems and Applications (AIDB'20), August 31, 2020, Tokyo, Japan.

known algorithmic model such as B+tree [13] or run an auxiliary algorithmic index alongside a learned model [22]. Such hybrid approaches ensure that while the learned model accelerates the search, the algorithmic index still guarantees the worst-case scenario when modelling is not effective for the given dataset, and also help in handling updates.

Linear models and interpolation. Among all learned index models, simple linear models have been arguably the most common [16, 7, 16, 22, 12, 3]. Even for a generic learned index such as the RMI model that supports a variety of linear and non-linear models [18], it is experimentally shown that the best configuration found for most real-world datasets is simply a linear spline model [15, 23]. Linear interpolation has been recently studied by some works [28, 9, 13]. Despite that an interpolation model is not as accurate as other linear models such as linear regression, it has the interesting feature that it does not need training, making interpolation an effective choice to be embedded inside hybrid indexes [13, 14].

Learned spatial indexing. Recently, some efforts have been done on accelerating spatial and multidimensional indexes with matching learning techniques. Nathan et al. suggested a specific learned multidimensional index that learns from the query distribution how to optimize a grid index structure [26]. Also, LISA [21] is a learned disk-resident R-tree. The difference between LISA and our work is that LISA focuses on minimizing the IO on disk by transforming the data into a single dimension using a lattice regression model. However, such data transformation is not worthwhile in a main-memory index due to the closer ratios of CPU and memory access overheads. Following a different line of work, some effort has been made to learning and soft functional dependencies between the dimensions in spatial indexes and exploiting them for performance optimisation on spatial indexes [8, 4].

3. MOTIVATION

Different approaches have been developed for indexing multidimensional datasets. One common approach is to use tree structures, based on space-oriented partitioning such as KD-Tree, Octree, Quadtree, or based on data-oriented partitioning like the R-tree. Hierarchical trees are easy to manage and update, and are efficient choices for databases. Arguably the most prominent spatial index is the R-Tree [11], the de-facto spatial index of a modern DBMS and used in IBM Informix, MySQL, PostgreSQL, Oracle, and PostGIS. However, in main memory, hierarchical trees require excessive pointer-chasing to execute queries. While on disk, for which the R-Tree historically has been developed, the time to follow pointers is insignificant (compared to the overhead of retrieving data from disk) but in main memory the time for chasing pointers contributes significantly to the overall query execution time.

New hardware thus calls for new approaches for designing indexing algorithms. First, the new indexes we design need to reduce latency at the cost of bandwidth, e.g., having bigger nodes in search trees addresses the issue of excessive pointer chasing. Second, given that the compute power of modern CPUs has improved considerably (as opposed to the memory bandwidth), the use of the CPU can and should be increased to reduce the need to access data. Third, new indexes should give preference to computations that are more friendly for new hardware, e.g., preference should be given to arithmetic operations over branch-heavy operators which can deteriorate the pipeline. In this regard, machine learning is an effective tool that facilitates designing hardware-efficient indexes that exploit patterns in data to reduce data retrieval.

While there is great potential for applying advanced machine learning techniques powered by SIMD operations, this paper focuses entirely on linear interpolation — the simplest possible model — and surprisingly we show significant performance improvement even in this case.

4. INTERPOLATION FRIENDLY SPATIAL INDEXES

4.1 Spatial Indexing Principles

A vast number of spatial indexing methods have been developed in the last decades [6]. Arguably the most broadly used spatial indexes are the R-Tree, the KD-Tree and the Octree (and its two dimensional equivalent, the Quadtree). We primarily focus on these indexes in our work, but the techniques described can be used to optimise further spatial indexes as well.

All these indexes work based on the same principles: they recursively partitions space and create a hierarchical structure so as to guide query execution. At the bottom of the trees, the data (or pointers to the data) is stored in the leaf nodes. The other nodes, the non-leaf nodes, are used to guide query execution and store the partitioning of the space.

The major difference between the R-Tree and the KD-tree as well as the Octree is the approach to space partitioning used. Historically developed for use on disk, the R-Tree uses a data-oriented partitioning, i.e., it partitions space such that each leaf node neatly fills a disk page (and therefore optimises the use of the disk bandwidth). Doing so leads to a partitioning entirely driven by the distribution of the data. The partitioning is stored in the non-leaf nodes, i.e., each non-leaf node N stores a number of pairs $< M, P >$. M is the minimum bounding rectangle (MBR) enclosing all MBRs (or spatial objects in case of the leaf node) of one child node C while P is the pointer to C.

The other approaches use a space-oriented partitioning, i.e., they partition the space recursively. The KD-tree cycles through the dimensions and at every level takes the median in one of the dimensions and splits space accordingly into two half spaces. It is essentially a binary tree, split on the median of the data indexed, i.e., using a hyperplane to split the space. Instead of only two children, the Octree and the Quadtree use eight and four children respectively and split the space/subtree in the middle rather than on the median.

In case of all indexes, a spatial range or point query (essentially a range query with extent zero), is executed by starting at the root node and traversing down the hierarchy using the MBRs (in case of the R-tree) or hyperplanes (in all other cases) in non-leaf nodes to ultimately find the answer to the query stored in the leaf nodes.

Primarily due to their broad use in practice, we focus on the R-tree, KD-Tree and the Octree (and its two dimensional equivalent, the Quadtree). While the R-Tree has historically been developed to speed up execution of queries on data stored on hard disk, it is still competitive when used in main memory as we will also show experimentally.

4.2 Approach & Contribution

In line with previous work on the B+-tree we therefore aim to use a model to predict the location of the records, primarily on the leaf nodes. Doing so will accelerate scanning the leaf nodes. At the same time, as scanning the leaf nodes becomes substantially faster, the optimal setting for the leaf node size is likely to be bigger as well. Bigger leaf node sizes in turn mean fewer leaf nodes which means fewer pointers and thus less pointer chasing, thus speeding up access further.

One particular issue on making a learning-augmented spatial index is to understand which part of the spatial index can be augmented with prediction models. Unlike one-dimensional range indexes such as B+-trees that have a nearly identical layout on all levels, the R-Tree has entirely different nodes as the leaf and non-leaf (internal) nodes. Internal nodes of the R-Tree store the MBRs of their children along with pointers to the children while the leaf nodes only store data, i.e., points. The MBRs stored in internal nodes consist of four values in 2D (and six in 3D) and the optimisation potential for internal nodes is therefore limited as we, for example, only can interpolate on one dimension/value yet we store four or six dimensions (plus a pointer per child node). We consequently focus on optimising access to leaf nodes where we can, for example, interpolate on one dimension but only have to store one additional dimension (or two in 3D).

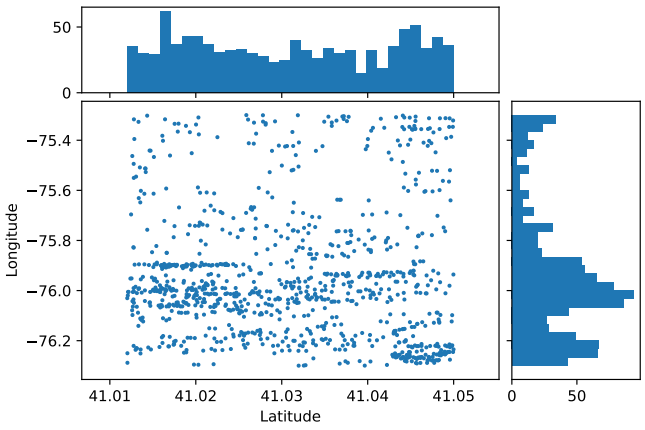
4.2.1 *Defining a Storage Order*

The IF-X indexes enrich their unmodified equivalent indexes with insights from learned indexes. A major challenge, however, is how to define an order for the records so that we can best predict. In a range index like B-tree, records are sorted by the key, hence the learned model simply learns the cumulative density function which maps the value of a key to the position of the key in the sorted array. In multi-dimensional data, however, there is no such total order defined for the records as there are multiple different dimensions over which the data can be sorted. One approach is to learn a projection $\mathcal{L} : \mathcal{R}^d \rightarrow \mathcal{R}$ that maps each d-dimensional record into a single dimension, hence recording the data points [17]. Nonetheless, such a conversion is computationally expensive, and this approach has only been effective for disk-based R-trees where the CPU time is negligible compared to IO cost [21].

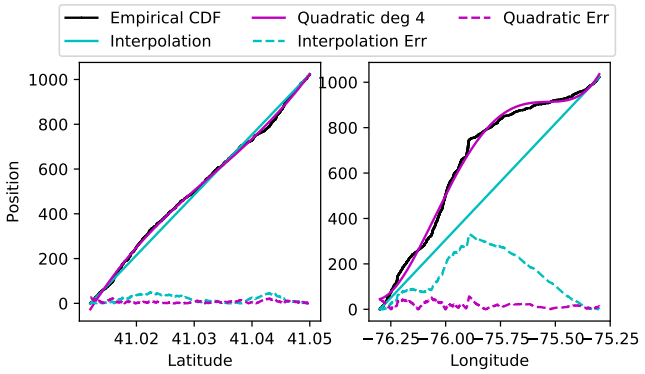
We take a different approach that does not need data transformation: sorting based on one of the existing dimensions. We choose for each leaf node individually the dimension in which it is the most predictable, i.e., the dimension for which the error of the model is the smallest. Figure 1a shows the data distribution of a leaf node containing 1024 2D records (spatial points), and the 1D projection of the points in each of the dimensions.

Figure 1b illustrates this by showing the CDF of the values in each dimension, along with two prediction models: linear interpolation (based on minimum and maximum values), and a quadratic polynomial with degree of 4 ($pos = ax^4 + bx^3 + cx^2 + dx + e$) that is fit to the data. The dashed lines show the error of the models. In the example leaf, the latitude values are more uniform and more predictable for both models.

Note that more complex models such as polynomial or RBF models have a higher capacity to fit the CDF and are



(a) Data distribution in a leaf node and the 1D mappings



(b) Evaluating the prediction error on different dimensions

Figure 1: Choosing the most predictable axis in IF-X

able to predict the positions more accurately. However, such complex models need to store more parameters for prediction and are also slower to compute, hence we only use linear interpolation. An additional benefit of interpolation is that there is no need to run an expensive training process.

4.2.2 *Maximum Versus Average Error*

. The choice of the best dimension for physical storage order also depends on the local search algorithm. Once the location for a key is predicted by the model, a local search is performed around the predicted position to find the correct result of the query. Local search in a learned index can be done using binary search, which requires specifying a range, or by other algorithms that do not need a specified range, such as linear search and exponential search. The common practice in learned indexes is to keep track of the *maximum prediction error* per node, say Δ, so that a search can be done on $[p\hat{o}s(x) \pm \Delta]$ where the result is guaranteed to be. Therefore, the complexity of the search is in the leaf node is $O(\log \Delta)$. Binary search is very effective and can be implemented branch-free [2]. However, if Δ is large, binary search can incur multiple TLB- and cache-misses, which can drastically reduce the performance. One effective solution to counter this issue is to check if the *average prediction error*, say $\bar{\Delta}$, is far less than the maximum prediction error (Δ). In this case, it is more efficient to use linear search or

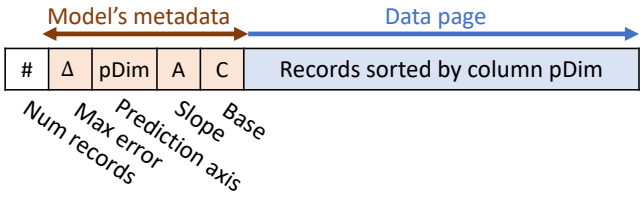


Figure 2: Layout of leaf nodes of IF-X indexes

exponential search, which rely on the average error instead of the maximum error. Linear search, for example, starts from the predicted location and scans towards left or right until it finds the first result belonging to the query, where it *stops* searching.

In this case, it is more efficient to use linear search, which is $O(\bar{\Delta})$ or exponential search ($O(\log \bar{\Delta})$). This also affects the choice of the dimension. If the search algorithm is unbounded (linear or exponential search), then the dimension which has the smallest $\bar{\Delta}$ is chosen as the storage order. Therefore, the most predictable dimension (storage order) in each leaf node is chosen with respect to the specific model being used as well as the in-leaf search algorithm.

5. IMPLEMENTATION

5.1 Data Structures

5.1.1 Leaf Node Layout

As discussed earlier, IF-X indexes use the same index structure as of their basic/plain indexes, except on the leaf level. None of the indexes considers any specific order for the records stored in each leaf node. Therefore, if the boundaries of a leaf node overlaps with the query, the indexes examine (compare) all records in the leaf node and select the ones that match with the query. As scanning large nodes is generally costly, in-memory the indexes are best configured have smaller leaf nodes which increases the depth of the tree.

IF-X indexes, on the other hand, sort the records in each leaf node based on the best order using which the interpolation error is minimized, as explained earlier. Our goal is to store all necessary information in the header of the leaf node, such that no extra memory lookup or excessive computation is required other than loading the data pages. Figure 2 shows the layout of the leaf node. The leaf contains the number of records K, the most predictable dimension used as the storage order (`pDim`), and model parameters (the slope A and base C of the line for linear interpolation). Also, we store the maximum error in case that binary search is used as the local search algorithm. Nothing else needs to be stored.

Note that in case of linear interpolation, it is not essential to store the model parameters in the leaf node. Linear interpolation does not need training and can be done using the minimum and maximum values of the chosen dimension, which can be fetched from the records page. However, accessing both ends of the records' page incurs further memory lookups and TLB/cache misses. Instead, at build time, we consider the minimum and maximum values in each node, pre-compute the model parameters (slope A and the base C), and store those parameters in the header of the leaf node. The model parameters for linear interpolation are:

$$A = \frac{maxPos - minPos}{maxVal - minVal}$$
$$B = minPos - A \times minVal \quad (1)$$

The position of a query point in a leaf node can be predicted as $p\hat{o}s = A \times q[pDim] + B$. In fact, slope precomputation and re-use is shown to be an effective optimization for iterative interpolation search [28] and we observed the same benefit when interpolation is used as a model.

5.1.2 Storage Order Implementation

Evaluating the predictability of each dimension requires sorting data on each dimension ($O(K \log(K))$ where K is the number of records in page), and then training and evaluating the model on that order. In case that linear interpolation is used, the model requires no training and takes a single scan to evaluate, hence the total complexity is $O(dK \log(K))$.

An additional optimization we use is to define a second sort dimension for records that have the same value for the primary storage dimension. In case that many records have the same value in the primary sort dimension, the second dimension allows for a faster linear search among those records with the same value on the primary sort dimension. We go one step further and sort the records lexicographically based on all columns, i.e., sorting the records by $[pDim, 0, 1, \cdots, d-1]$. This enables further optimization on the scan operator, e.g., reducing branch misprediction in unrolled loops.

5.2 Building & Updating

We use the STR [19] bulkloading strategy to partition the data and then use our leaf node creation algorithm to create the leaf nodes. All other nodes are subsequently created using the STR approach. Any other approach to build an R-Tree or the other trees can be used whilst modifying it to use our leaf node creation approach.

Our suggested modification to spatial indexes does not affect the ability of IF-X indexes to handle updates. Indeed, any update strategy can be used in connection with recently proposed methods to outfit one-dimensional learned indexes for update-heavy workloads [3, 12], extended to multidimensional learned index structures.

5.3 Querying

5.3.1 Point Queries

Processing point queries is straightforward, the only difference in implementation with IF-X indexes is on the leaf node search. Given a query $q = x_0, x_1, \cdots, x_{d-1}$, we first consider the predictable storage dimension of the leaf ($pDim$) and try to find the first record where $points[i][pDim] = q[pDim]$. To do so, we use the learned model of the leaf node to predict the location of the record. For a linear model (including linear interpolation), the predicted position is $p\hat{o}s = A \times points[pDim] + B$. Then, a local search is performed around the predicted location. Local search can be done using either binary, linear, or exponential search. If the local search algorithm is binary search, the boundary for search would be $pos[p\hat{o}s \pm \Delta][pDim]$. Once the first (leftmost) record with $points[i][pDim] = q[pDim]$ is found, all records with equal value on $pDim$ are compared with query on all dimension to examine if they match.

Algorithm 1 describes the search algorithm for point queries.

Algorithm 1 Local search in leaf nodes for point query

```
1: procedure LeafSearch(q, points, model, pDim, sDim)
2:     pôs = model.predictLoc(q[pDim])
3:     if q[pDim] > points[pôs][pDim] then
4:         pos = search(from=pôs − Δ, to = pôs, axis=d)
5:     else if q[pDim] < points[pôs][pDim] then
6:         pos = search(from=pôs, to = pôs + Δ, axis=d)
7:     end if
8:     if q[pDim] = points[pos][pDim] then
9:         // Use the second dimension for guiding search
10:        if q[sDim] > points[pos][sDim] then
11:            pos = LinearSearch(from=pos, direction=left, axis=sDim)
12:        else
13:            pos = LinearSearch(from=pos, direction=right, axis=sDim)
14:        end if
15:        while q[pDim,sDim] = pos[pos][pDim,sDim] do
16:            if query = points[pos] then
17:                return data[pos]
18:                pos += 1
19:            end if
20:        end while
21:    end if
22:    return null
23: end procedure
```

5.3.2 *Range Queries*

In range queries, the query is defined by a rectangle, i.e., $q = [(l_0, u_0), \cdots, (l_{d-1}, u_{d-1})]$. Since the records are sorted by $pDim$, we find the first record where $q[pDim].l \geq points[i][pDim]$. To find such a record, we do the same range search as in range indexes [18], which is similar to the point query search (Algorithm 1), but replaces the equality check with inequality ($\leq$) in the matching constraint. Once the first record is found, we perform a scan from the found position and compare all records with the query constraints, until we reach the first record that hits the upper limit of the query on $pDim$'s dimension, i.e., $q[pDim].u < points[i][pDim]$, after which no further records will match the query.

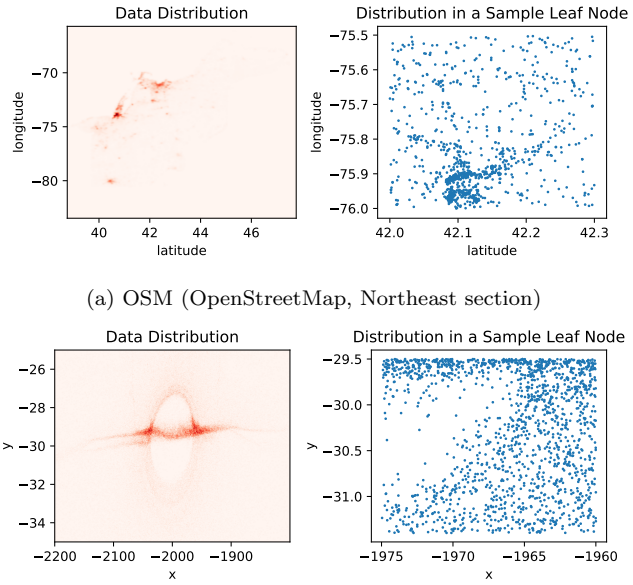
6. EXPERIMENTAL ANALYSIS

6.1 Experimental Setup

Hardware. Single-core experiments are performed on a system with 16 GB of memory and Intel Core i7-6700 (Skylake), which has four cores and is running at 3.4 GHz with 32 KB L1, 256 KB L2, and 8 MB L3 caches. Multithreaded experiments (Figure 5b) are conducted on an AMD EPYC 7401 server running at 2GHz with Virtualization, having 24 available cores with 32 KB L1, 512 KB L2, and 64 MB L3 caches. Both machines have 16GB of memory.

Software. All indexes are implemented in C++ and compiled with GCC 9.2. Experiments are run on Ubuntu 18.04 with kernel version 4.15.0-65.

Datasets. We used two spatial datasets: *osm*: latitude and longitude of landmarks from the US northeast section of OpenStreetMap data [1], with the timestamp being used as the third dimension in the 3D experiments; and *3DScans*:



(a) OSM (OpenStreetMap, Northeast section)

(b) 3DScans

Figure 3: Distribution of the datasets

A point clouds collected for robotic experiments, built by a 3D scan (containing X,Y,Z coordinates) taken in front of the cathedral in Zagreb, Croatia [27]. We used $1M$, $10M$ and $100M$ sample records from each dataset. Figure 3 shows the heatmap of data distribution in both OSM and 3DScans data, along with an illustration of data distribution in sample leaf nodes within each dataset. We used 2D data with 10M records as the default configuration for the experiments. Values from all dimensions are stored as single-precision floating point values.

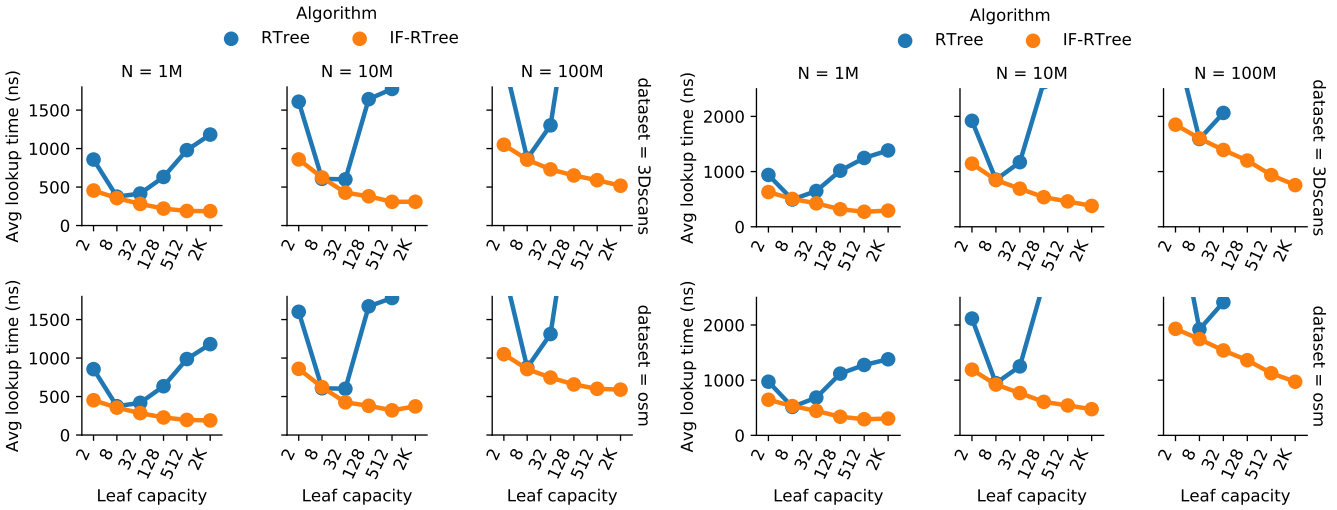
6.2 Tuning

Each spatial index is tuned with its internal parameters, most notably with node sizes between 2 and 32K records per node ($2^i, 1 \leq i \leq 15$). Figure 4, shows how the optimization has been done on R-tree index, optimized for point queries. For each data size and query type, we took the optimum leaf size for each index and compare the indexes with best configuration. However, the optimal configuration for each indexing algorithm does not vary much for different dimensions.

In our default configuration (2D index, 10M records, tuning for point queries), the best leaf sizes found were: R-tree: 16, KD-tree: 128, Octree: 256, IF-RTree: 512, {IF-KDTree, IF-QuadTree, IF-Octree}: 2048. The optimal configurations found for 2D and 3D data are similar.

6.3 Point Queries

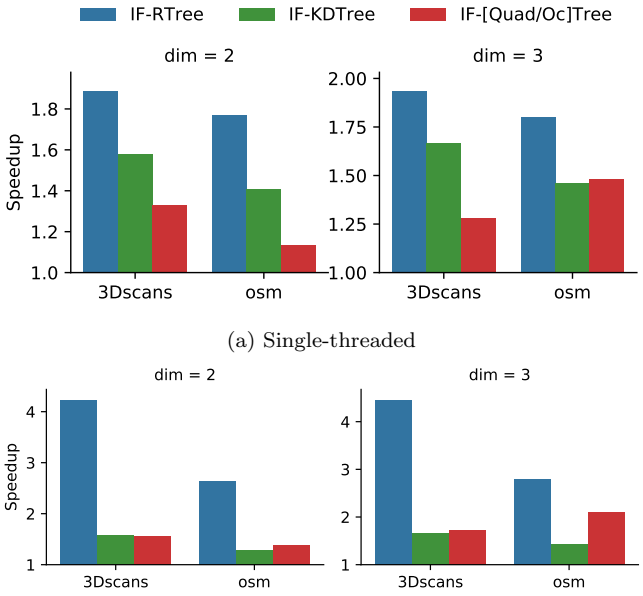
Overall performance We first benchmark how well our IF-X indexes can optimize their base versions. The point queries are randomly sampled from the datasets. Figure 5a shows the speedup obtained by each IF-X index over its own baseline for each dataset. IF-RTree has the highest speedup (1.8X speedup over R-tree in osm, 1.9X in 3DScans; for both 2D and 3D data). Other algorithms, IF-KDTree and IF-[Quad/Oc]-Tree have speedup between 1.15X and 1.6X



(a) 2D data

(b) 3D data

Figure 4: Index parameter tuning: effect of leaf capacity on R-tree and IF-RTree indexes



(a) Single-threaded

(b) Multi-threaded (24 Cores)

Figure 5: Speedup of the IF-X indexes to their original version for point queries

for different datasets and dimensions.

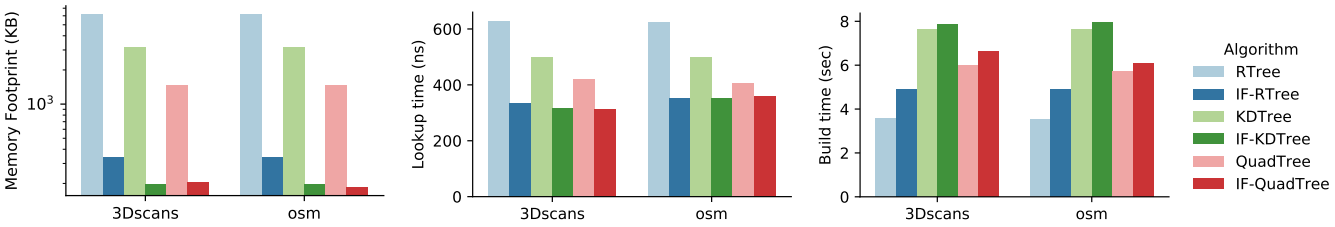
Dimensionality An interesting observation is that IF-X indexes almost retain their speedup for the 3D data, with even a rise in the speedup of IF-Octree (1.5X speedup on 3D osm, compared to 1.15 in 2D). This is due to the fact that even though the interpolation is done on one dimension, with higher dimensions the IF-X indexes have more freedom to choose the best storage order (separately for each leaf), therefore some of the leaf nodes that are skew in 2D become more predictable when sorted on the third dimensions. We can think of it as, for example, how in Figure 3b(right) the x axis makes the data more predictable compared to a one-dimensional index over y only.

Scalability. Along with the single-threaded experiments, we compared the speedup of the IF-X indexes in a multi-threaded setting using a 24-core machine (explained in section 6.1). In this setting, we use the total execution of the query batch, hence the measurements and the reported speedup shows the improvement in throughput, not the average query times of individual queries. All the three spatial indexes have benefited from model-assisted acceleration in all datasets, with over 4X speedup in IF-RTree.

Performance breakdown. We are interested to see where does the speedup of IF-X indexes over the baselines come from. To further analyze the speedups gained for the default configuration (Figure 5a-left), we measure the index size and average lookup times.

Memory footprint. The optimal parameters found in the tuning phase suggest that IF-X indexes prefer much larger leaf nodes than the original X indexes. Larger leaf sizes manifest in smaller footprints in the IF-X index. By footprint, we mean the internal nodes of the indexes and put aside the size of the leaf nodes (which equals the data size). Figure 6a shows, in logarithmic scale, the memory usage of each index in its optimal setting for 2D data with 10M records. The difference in memory footprint is significant. For example, IF-RTree takes up only 5.5% of the size of R-tree, and the other IF-X indexes are no larger than 10% of their base version. Such a saving in memory footprint is typically enough to push the index up in the memory hierarchy and fit the internal nodes into processor's caches.

Average lookup times. Comparing the lookup times of the algorithms provides insights into why some IF-X indexes have a comparably smaller speedup compared to the others. Lookup times are shown in Figure 6b, which show that all IF-X algorithms derived from different base indexes result in almost identical lookup time. This suggests that the IF-X family from different base methods converge to a similar: A cache-resident index structure along with large, flattened leaf nodes that are sorted to optimize the predictability of



(a) Memory footprint (b) Average lookup time (c) Build time

Figure 6: Comparison of indexes on their best configuration

the record locations.

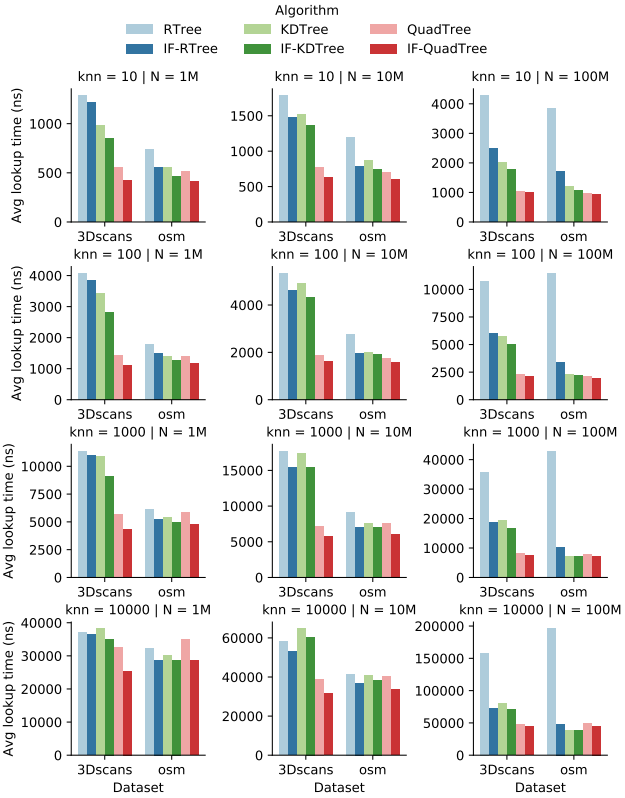


Figure 7: Performance comparison on range queries

Build times We also report the build times of the indexes. Building IF-RTree takes 37% longer time than RTree, and the other IF-X indexes take less than 10% longer to build. Note that each algorithm in the chart has a different leaf size. Unlike the base indexes in which a significant share of the build time is taken on building the internal nodes, the build time of IF-X indexes are mostly taken by sorting the large leaf nodes on multiple axes in order to choose the most predictable storage order. This shows a considerable potential in scaling IF-X methods with parallel execution, because sorting large leaf nodes for storage selection in IF-X can be done independently for each node using multiple threads; while paralleizing the baseline algorithms requires concurrent access control mechanisms on tree indexes and is harder to scale. However, scaling up the build algorithm is not considered in this work.

6.4 Range Queries

Query generation. For range queries, we considered four sets of queries with different selectivities, $\sigma \in S = \{10, 100, 1K, 10K\}$. We build a KD-Tree for the chosen datasets (for appropriate sizes). For each $\sigma \in S$, we then choose random points in the KD-Tree, and find the set of nearest σ points (with euclidean metric), P. The MBR of P then gives us the desired query.

Performance. Figure 7 shows the performance of the indexes for range queries. For lower selectivities (small K), query time is mainly affected by the initial lookup of the corner point of the query rectangle and hence the speedups are more similar to that of point queries. For larger selectitivies, however, query times are be mostly taken by the scan operations. Yet, since all IF-X indexes favour larger leaf nodes, the scan operations enjoys a considerable speedup.

7. CONCLUSION AND FUTURE WORK

In this paper we have integrated learned models into the most broadly used spatial indexes. We dub the resulting indexes IF-X indexes for all of them as they all are based on space-partitioning trees for the organisation of data. Integrating learned models is not straightforward in spatial indexes. Multiple dimensions and the absence of a total order make it challenging to find computationally efficient models that accelerate query execution. As our experiments show, however, our IF-X indexes are consistently considerably faster than the their plain, unmodified equivalents and they also consistently have a smaller footprint.

In our current work we did not make any assumptions about the query workload. In case that the query workload is known a priori, the leaf creation algorithm can be modified to use the dimension that is most predictable and most helpful for executing the workload. For example, if most queries do not put a constraint on a specific dimension, then that column is not selected for ordering the records. We leave this as future work.

8. REFERENCES

[1] Openstreetmap on aws. https://aws.amazon.com/public-datasets/osm. Accessed 2020-05-29.

[2] Performance comparison: linear vs binary search. https://dirtyhandscoding.github.io/posts/performance-comparison-linear-search-vs-binary-search.html.

[3] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska. Alex: An updatable adaptive learned index. page 969–984, 2020.

[4] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads, 2020.

[5] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8), 2020.

[6] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Survey*, 30(2):170–231, 1998.

[7] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the International Conference on Management of Data*, 2019.

[8] B. Ghaffari, A. Hadian, and T. Heinis. Leveraging soft functional dependencies for indexing multi-dimensional data. *arXiv preprint arXiv:2006.16393*, 2020.

[9] G. Graefe. B-tree Indexes, Interpolation Search, and Skew. In *DaMoN*, 2006.

[10] G. Graefe et al. Modern B-tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[11] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the International Conference on Management of Data*, 1984.

[12] A. Hadian and T. Heinis. Considerations for Handling Updates in Learned Index Structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2019.

[13] A. Hadian and T. Heinis. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *EDBT*, 2019.

[14] A. Hadian and T. Heinis. Madex: Learning-augmented algorithmic index structures. In *Proceedings of the 2nd International Workshop on Applied AI for Database Systems and Applications*, 2020.

[15] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[16] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2020.

[17] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A Learned Database System. 2019.

[18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, 2018.

[19] S. Leutenegger, M. Lopez, and J. Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. 1997.

[20] P. Li, Y. Hua, P. Zuo, and J. Jia. A scalable learned index scheme in storage systems. *arXiv preprint arXiv:1905.06256*, 2019.

[21] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the International Conference on Management of Data*, 2020.

[22] A. Llavesh, U. Sirin, R. West, and A. Ailamaki. Accelerating b+tree search by using simple machine learning techniques. In *Proceedings of the 1st International Workshop on Applied AI for Database Systems and Applications*, 2019.

[23] R. Marcus, E. Zhang, and T. Kraska. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the International Conference on Management of Data*, 2020.

[24] M. Mitzenmacher. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884*, 2018.

[25] M. Mitzenmacher. Optimizing learned bloom filters by sandwiching. In *NIPS*, 2018.

[26] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *Proceedings of the International Conference on Management of Data*, 2020.

[27] J. Schauer. Robotic 3D scan repository. http://kos.informatik.uni-osnabrueck.de/3Dscans/, 2017.

[28] P. Van Sandt, Y. Chronis, and J. M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the International Conference on Management of Data*, 2019.