# A Learned Bucket Index Supporting Spatial Queries

Martin Folmer
Department of Computer Science
Aalborg University
mfolme13@student.aau.dk

Raphael Neumann
Department of Computer Science
Aalborg University
rneuma14@student.aau.dk

Theevaahar Karunanithi
Department of Computer Science
Aalborg University
tkarun14@student.aau.dk

**Abstract— Learned indexes have proven its potential on one-dimensional and multidimensional relational data. In this paper, we investigate the applicability of learned indexes in a spatial context. We devise a projection function that orders spatial data into a learnable ordering. Using this function, we train models over the ordering of the data, and build an index consisting of a hierarchical structure and smaller utility components. We specifically build this index to accommodate range and nearest neighbor queries. To perform nearest neighbor queries, we convert them into range queries. We compare the speed of the index to that of the R-tree on the range and nearest neighbor queries by testing on datasets of different distributions and sizes. Through our evaluation, we show that the learned index is able to outperform the R-tree in most cases. With the implementation of our learned spatial index, we show that, despite the challenges that exist within a spatial context and the inherent uncertainty with models, learned indexes remain powerful tools with potential in many types of data.**

Keywords— *R-tree, Machine Learning, Deep Learning, Learned Spatial Index, Neural Network, Multidimensional Data, Spatial Queries*

## 1. Introduction

Machine learning has begun its advancement into various fields of computer science and databases are one of them. Index structures, such as B- and R-trees [1] [2], have for a long time been the standard for database indexing, but with the introduction of machine learning algorithms in this field, it has become possible to radically change the way indexes are created. This is demonstrated in [3], wherein multiple traditional data structures for one-dimensional data, such as B-trees, hash indexes and Bloom filters, are replaced with what they coin to be ”learned indexes” at an increased performance.

The authors state that traditional indexes, such as the aforementioned, are general purpose indexes, in that they treat all data equivalently without making any assumptions about patterns existing within the data. They therefore argue that machine learning models, which are able to learn patterns within the data, are theoretically able to perform the same procedures as some indexes at an increased performance. B-trees, for example, map a value to a location in an one-dimensional sorted container, see Figure 1. A B-tree can therefore be regarded as a model, that takes a value and determines its position in memory, which is a relationship a model can learn with the use of machine learning. This is especially simple in the case of B-trees, due to the sorted nature of the data, which regression models can take advantage of. It is for the same reason a trivial task to combat the inherent imprecision in model predictions and guarantee that the correct result is found, since a simple local search from the prediction can be utilized to locate the correct result.

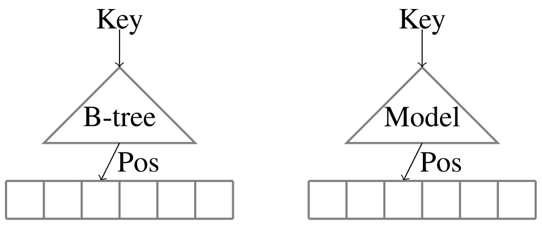

Fig. 1: How a B-tree can be regarded as a model, identical to Figure 1 in [3].

The authors in [3] further argue that the same principles for one-dimensional data can be applied to multidimensional data by also using machine learning. This is investigated in [4]. The authors' focus is on multidimensional relational data, namely the TPC-H benchmark database [5], consisting of items, orders, customers etc. By applying a projection function that creates groupings for each dimension, such that rows of the same value in a specific column are grouped together, they are able to create indexes with increased performance on range queries. While this strategy works well on multidimensional relational data, there is little to know about spatial data. This is largely due to the fact that while spatial data is also multidimen-

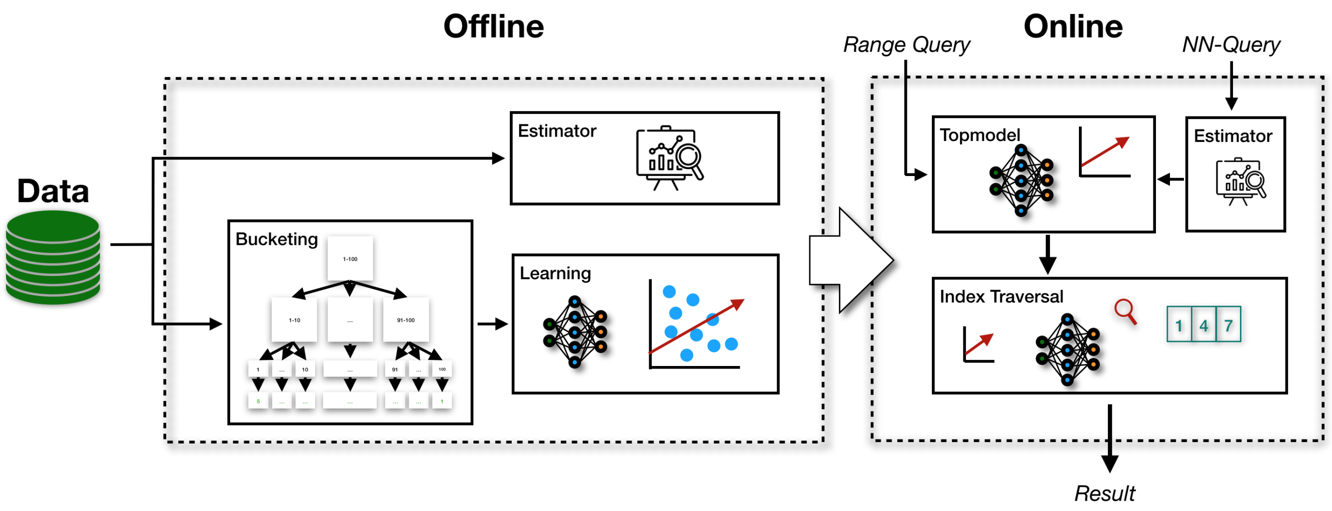

Fig. 2: The architecture of the Learned Spatial Bucket Index.

sional data, the relationship between its dimensions are vastly different. It is therefore inevitably harder to reach the same effect when applying the projection function, proposed in [4], on spatial data compared to multidimensional data.

The main contribution of this paper is to create a learned spatial index by devising a projection function suitable for spatial data. We show that the inherent problem of imprecision when using machine learning models is not as severe, as the learned spatial index is able to provide the same hard semantic guarantee, that the correct result is returned, as the R-tree in a multidimensional setting. We argue that the support of range queries and nearest neighbor queries shows the novelty of the Learned Spatial Bucket Index, and furthermore the potential of answering computationally more complex queries using a learned index.

The architecture of the index we propose, which we name the Learned Spatial Bucket Index, is shown in Figure 2. The offline phase, which occurs before any queries are executed, involves applying the projection function, which we name bucketing, on the data, followed by training models. Furthermore, an estimator is built based on the data to be able to estimate bounds that are required for executing nearest neighbor (NN) queries. This estimation is necessary because the NN-query is translated into a range query. The range query and the NN-query have the same flow in the online phase, with the exception that the NN-query is converted into a range query, through the utilization of the aforementioned estimator. Subsequently, the index is traversed by invoking the required models and retrieving the correct result for the queries. Each component will be explained in details throughout the paper.

The rest of this paper is structured as follows: Section 2 introduces the concept of bucketing by explaining how an order can be achieved on multidimensional data. Section 3 describes the challenges that arise when working with spatial data. Section 4 covers how learning is deeply integrated into the Learned Spatial Bucket Index. Section 5 describes the spatial queries that the Learned Spatial Bucket Index supports, namely range queries and nearest neighbor queries. Section 6 covers how the models are trained and the results of the tests. Section 7 mentions work that is related to our paper. Section 8 concludes on the paper and Section 9 proposes future directions.

## 2. Ordering Multidimensional Data

The inherent inaccuracies in machine learning (ML) models requires the learned index to have the ability to handle these. Without this ability, it is impossible to provide the hard guarantee that the index will return the correct result. Therefore, a projection function must create an ordering of multidimensional data such that this guarantee can be provided.

Bucketing is such a projection function which maps each record of a dataset to a specific sort key, which in turn determines its position in memory. With the use of this projection function, it becomes possible to use ML models, e.g. neural networks, linear regression, etc., to learn the relationship between specific sort keys and the locations of their content in memory, effectively

| A | B | C | D |
|---|---|---|---|
| $x_1$ | $y_1$ | $z_1$ | $v_1$ |
| $x_2$ | $y_1$ | $z_2$ | $v_1$ |
| $x_2$ | $y_2$ | $z_1$ | $v_2$ |
| $x_1$ | $y_3$ | $z_4$ | $v_1$ |
| $x_4$ | $y_4$ | $z_3$ | $v_2$ |
| $x_3$ | $y_2$ | $z_3$ | $v_3$ |
| $x_1$ | $y_3$ | $z_4$ | $v_2$ |
| $x_4$ | $y_4$ | $z_4$ | $v_1$ |

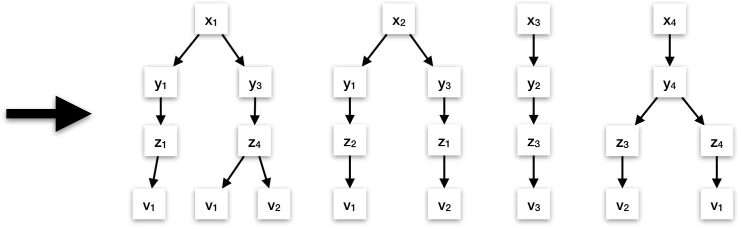

Fig. 3: Applying bucketing on a dataset with four dimensions, where each box represents a bucket containing the value specified on it.

creating an index.

When bucketing a multidimensional dataset, all the records are ordered on a sequence of dimensions. Bucketing is performed by grouping records that share the same value on a chosen dimension. These groupings are called buckets. The buckets and the values within the buckets on that dimension are then sorted, and a model is then learned on the bucket keys and their content. Within each bucket, this process is recursively applied for the remaining dimensions. This effectively creates new buckets within the original buckets, that are ordered on a different dimension than the previous one.

The process of bucketing is seen in Figure 3, where a table of data is transformed into buckets. Firstly, dimension A is bucketed and sorted: Since there in dimension A are eight rows, but only four unique values, there will only be four buckets; one for each unique value. This transformation of data into buckets enables the learning of a model that can be trained on the buckets formed by grouping dimension A, meaning a single model would learn the association between the four x-values to the four different buckets. Values in dimension B that are associated to values from dimension A, based on the rows in the dataset, are bucketed in the next level, e.g. since both $y_1$ and $y_3$ in dimension B have the value $x_1$ in dimension A, these are sub-bucketed in $x_1$'s bucket. Models on dimension B are then trained on the association between the y-values' position within a given x-bucket. This process continues for each bucket in each dimension.

In a scenario with knowledge of the query workload, it would be the most advantageous to bucket by the dimensions which are queried the most. Taking the TPC-H dataset as an example, if the orders table is queried mostly on the discount dimension, then this dimension should be the one to bucket first. The reason for this is due to the nested nature of the buckets. Any queries on only latter dimensions will have their ordering shifted by the previous dimension. The answers to queries not containing the previous dimension are therefore slower to locate, since the model that can quickly determine the relevant buckets to search is not used. In a spatial context however, since queries will almost always cover each dimension, the question of which dimension to order by first is less relevant.

## 3. The Challenge of Spatial Data

Due to the nature of spatial data, the relationship between the dimensions is vastly different than between dimensions of relational data. This difference is made evident by the nature of spatial queries which includes queries expressed as *"which object is nearest to this object"* and *"which objects are within 10 kilometers of this object"*. These types of queries do not fit the context of relational data such as the TPC-H benchmark.

Trying to apply the bucketing approach presented in Section 2, would in a spatial context result in a large number of individual buckets as the frequency of duplicate data points along the first dimension is low as compared to e.g. order dates in TPC-H. The large number of buckets will mean that the model, which is used to locate the appropriate bucket for a given query, will be inaccurate as there will only exist a single, or very few, values to train the model on for each bucket. Consider the scenario seen in Figure 4; here it is evident that trying to bucket and closely approximate a function, i.e. a model, capable of, for each bucket on a given dimension, outputting the memory location of that bucket, will be impractical as the accuracy of the function will be poor.

The inaccuracy of the function will in turn mean that this will have to be corrected. This was proven trivial in [3] with one-dimensional data, as the data simply can be sorted. However, with multidimensional data

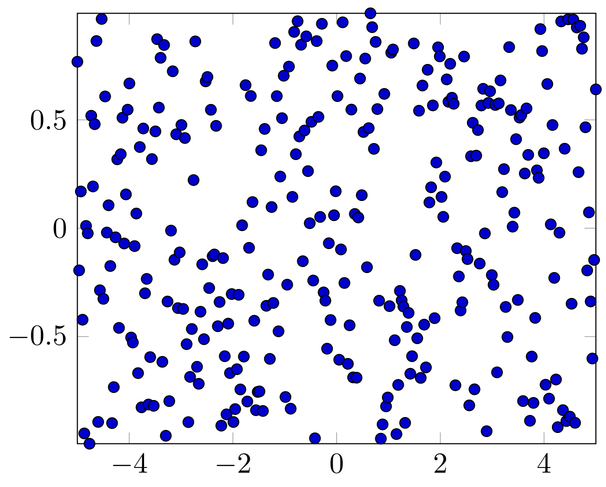

Fig. 4: Randomly distributed two-dimensional data points.

this issue is harder as no learnable sort order enabling correction exists, e.g. the Z-order, which creates an ordering of multidimensional data, is impossible to correct and hard to learn [4]. This issue is again evident in Figure 4, as sorting these points, such that model imprecision can be corrected, is no trivial task. Furthermore, as the function trained per definition will be an approximation, imprecision will occur and therefore has to be accounted for, in order to provide the hard guarantee that the correct result is always found.

Overall these issues all arise from the nature of spatial data and spatial queries and their vast difference from their relational counterparts. This means that simply applying the multidimensional projection strategy, presented in Section 2, in a spatial data context is not sufficient to obtain promising results. Therefore, it is crucial that a projection function suitable for spatial data is established in order to support spatial queries. Such a projection must enable the created buckets to be learnable and ensure an ordering, such that any imprecision can be corrected, in order to provide the critical hard guarantees.

### 3.1 Spatial Data Bucketing

Applying bucketing on spatial data is structurally similar to the bucketing of relational data described in Section 2. Each bucket, however, now contains a range of unique values. This change is needed because there are almost exclusively unique values given the nature of spatial data, which inherently has values of high decimal precision. The amount of values within the range need not to be fixed allowing for dynamic bucket sizes. Covering ranges of values instead of single values provides buckets with enough data to train on, thereby producing buckets that are easier to learn. This strategy also maintains the ordering of the buckets, thereby ensuring that the correction of predictions is still possible. The spatial bucketing approach is part of the offline phase in the architecture, shown in Figure 2. Figure 5 illustrates the structure of the spatial bucketing.

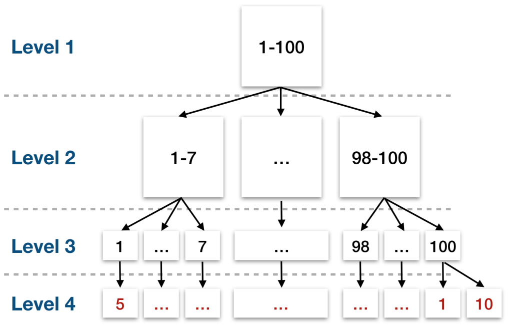

Fig. 5: Spatial Bucketing, where Level 1-3 represents bucketing performed on the first dimension and Level 4 represents bucketing on the second dimension.

Here, given a set of 100 two-dimensional continuous value pairs $(x, y) = \{(x, y)|x, y \in \mathbb{R}, 1 \leq x \wedge y \leq 100\}$ and a bucket size of 10 values, there will at Level 1 be a single bucket containing all values on the $x$-dimension. At Level 2, there will be 10 buckets, since each bucket can contain 10 values. In turn there will at Level 3 be 10 buckets for each Level 2 bucket. These buckets contain only a single $x$-value. Each of these buckets have one or more buckets at Level 4, where each bucket contains a single $y$-value. These buckets and their associated Level 3 buckets correspond to the data points in the dataset. If the data point $(1, 5)$ exists in the dataset, there will, as seen in Figure 5, be a bucket at Level 3 containing $1$ and a bucket at Level 4 containing $5$, which is pointed to from $1$'s buckets.

The hierarchical structure of the buckets makes it possible to perform aggressive pruning of buckets that would otherwise be considered in answering a query. Section 5.1 will cover this in-depth.

## 4. Learning Spatial Buckets

While bucketing produces an ordering that is learnable, it is equally important to identify the models that are the most advantageous, in order to have an as optimal index as possible. E.g. if an index is created on a column with an autoincrement value, it is adequate to use a linear regression model, since the relationship between the values and their positions will exhibit an exact linear relationship. A neural network may also be able to learn this relationship, but would be less preferable, as a linear regression model has a lower cost of inference. However, if the index is created on data which does not exhibit an easily learnable linear relationship, a neural network might be more advantageous despite the higher inference cost, since the neural network may be able to approximate the desired function better and provide higher accuracy. Figure 6 shows an example of an ideal function for a given ordering, and how an approximate function could estimate this ideal function.

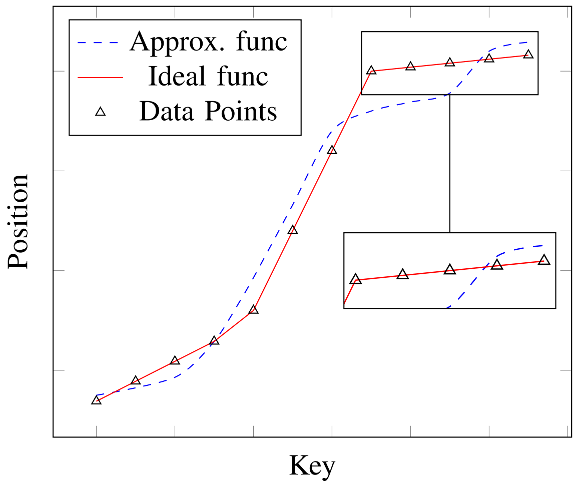

Fig. 6: The ideal function covering a set of points and how an approximate function might estimate that ideal function.

In the figure, a neural network could be learned on the entirety of the data points, creating a function (dashed line) that approximates the relationship between the data points' positions and the sort keys. While this approach may be able to learn the overall distribution of the data, in a scenario with millions of data points, the approximation function may have difficulty at being precise at locating individual data points. In contrast, a linear regression model may be better at accurately predicting at the level of individual data points, if the relationship between the data points and the sort keys is sufficiently linear. This is preferable in the setting of Figure 6, as evident by the ideal function, as the zoom-in on the last points shows, that these points are ordered linearly between their position and key. This relationship would therefore be easily learnable by a linear regression model. Having multiple linear regression models, which have been trained on different parts of the data, instead of a single large model, can therefore be advantageous, since their low cost of inference provides the index with more speed. It is, however, an option to have a combination of multiple types of models. A single model could learn the overall distribution of the data, where smaller models are trained on different subsets of the data, in cases where the last mile accuracy is important. This idea of combining different models is proposed in [6], where large ML models are subdivided into smaller models which are responsible for subsets of the data.

In an optimal scenario, the type of model to use could be determined by building numerous different models, e.g. multi-class classification, regression, etc. The model providing the best performance could then be chosen based on metrics such as accuracy and inference cost when predicting. This approach has the advantage of being specific to the data, meaning the data chooses the models that are the best-performing. However, due to the vast number of options and the cost of building all these different types of models, it would quickly become infeasible, whenever an index on new data is to be built. Instead, looking at the data itself and the ordering of it, it becomes clear that when applying bucketing, we are essentially making the relationship between the spatial data and their positions linear by splitting the dimensions and sorting along these dimensions. This effectively creates a linear, or close to linear, relationship, since data points have been projected to a single dimension, which are the well-suited for linear regression models. In situations where a linear regression model over a specific subset of data is performing poorly, it is always a possibility to replace that model with another type of model, which has better accuracy, since all that is needed from a model is a prediction. This makes the Learned Spatial Bucket Index generic, since models can be replaced by other models as seen fit. This flexibility allows for the creation of a dynamic index, that is not limited to a single type of model.

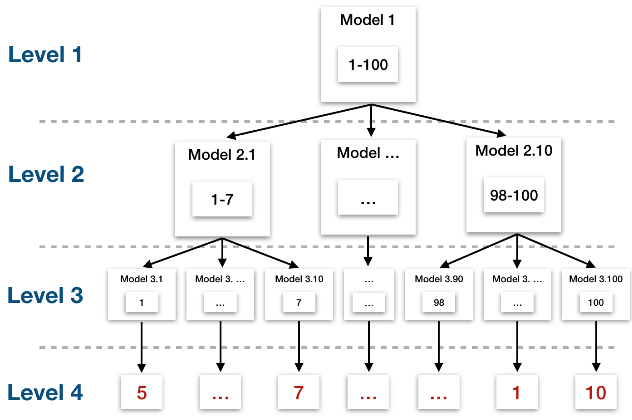

Fig. 7: Extending Spatial Bucketing to contain models.

In this paper, both linear regression models and a single neural network are applied, as both are useful in different scenarios; the single neural network is applied at Level 1 of the bucketing structure in Figure 5 to quickly narrow down the region of the relevant buckets by learning the relationship between the values within each bucket and their bucket key, while a linear regression model is created for each sub-bucket, learning the relationship between the data points within the bucket and their position within the bucket, to quickly locate the relevant values within each bucket.

Figure 7 shows the new structure, which has been expanded to contain models. Here, the single neural network model is introduced at Level 1, which learns the distribution of the points 1-100 into different buckets at Level 2. In turn, each of the buckets at Level 2 have a linear regression model covering data points of different ranges, e.g. Model 2.1 covers the range 1-7. This means when a key is input into Model 1, this model will predict the bucket, which covers that key. The model for that bucket is then leveraged again, to make a prediction as to which bucket at Level 3 the key resides in. When the correct bucket is found at Level 3, a new model at Level 3 is then utilized, to make a prediction as to where the key's y-values reside.

## 5. Spatial Queries

The spatial bucketing will allow execution of different types of spatial queries against the Learned Spatial Bucket Index. Due to the nature of spatial data, these types of queries can differ substantially from queries executed against one-dimensional or relational indexes. E.g. expressing and answering NN-queries is more natural in a spatial data context than expressing such queries on relational data. Section 5.1 will introduce and address the range query and Section 5.2 likewise for NN-queries.

### 5.1 Range Query

The range query is defined as follows: given a query window $q_r$, and a collection of spatial objects $P$, return any object from $P$ that intersects with $q_r$.

A range query $q_r$ for two dimensions, $x$ and $y$, is defined as $q_r = (x_1, x_2, y_1, y_2)$, where $x_1$ and $x_2$ are the upper and lower bounds for the $x$-dimension, and $y_1$ and $y_2$ for the $y$-dimension, respectively.

When executing the range query $(x_1 = 1, x_2 = 7, y_1 = 5, y_2 = 10)$ on the structure depicted in Figure 8, Model 1, given $x_1$, should return the first bucket at Level 2, since $x_1 = 1$ and the first bucket covers 1-7. However, in the case of producing a wrong prediction, a local search has to be utilized in order to locate the correct bucket. This is performed by checking the range of the predicted bucket and identifying the direction in which the correct bucket is found. This is possible due to the sorted order of the buckets and each bucket having an upper and lower bound for each dimension for the values it contains.

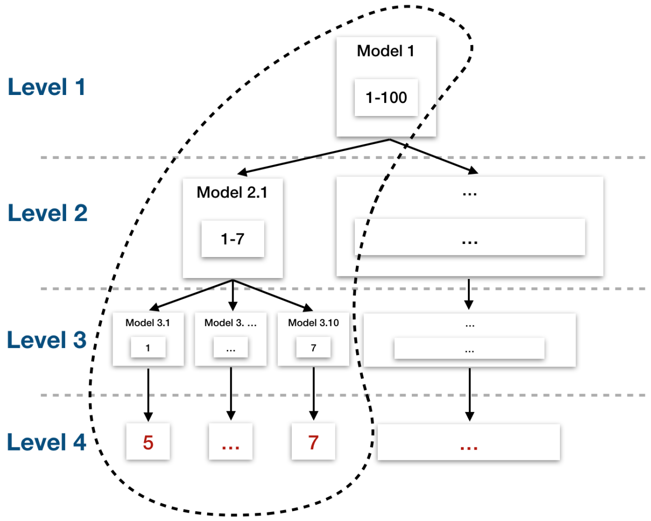

Fig. 8: Example of range query being executed, highlighted in the dashed circle

After the bucket containing $x_1$ is found, a scan towards $x_2$ is initiated, returning all buckets containing values in the interval $[x_1, x_2]$. Subsequently, the naive approach would be to produce all $(x, y)$-pairs of the returned buckets, and then check which pairs satisfy the range query. However, as mentioned in Section 3.1, the structure of the Learned Spatial Bucket Index allows for aggressive pruning. Each bucket that covers a range of $x$-values also has an upper and lower bound for the $y$-values it covers, e.g. the first bucket at Level 2 has the lower bound of 5 and upper bound of 7 for its $y$-values. This means, that when the buckets that satisfy the $x$-dimension of the range query are returned, each bucket is also checked on whether it contains $y$-values that are within the range query's $y$-dimension. This is performed by simply matching the bucket's upper and lower bound on the $y$-dimension with the range query's $y$-dimension. If the bucket's bounds for the $y$-dimension are not satisfied by the range query, the bucket is simply pruned. In the case that the bounds are satisfied, the entire bucket cannot be pruned. Therefore an extra measure is taken, where each bucket at Level 3, which is a sub-bucket of the unprunable bucket, is checked on the bounds of its $y$-values. Therefore, in the case where a bucket at Level 2 cannot be pruned, buckets at Level 3 are checked instead, to identify whether the sub-buckets of the unprunable bucket can be pruned. This process allows for the discarding of potentially a large amount of buckets, depending on the specific bucket configuration. After all possible pruning has been completed and all pairs have been found, they are returned.

### 5.2 Nearest Neighbor Query

A NN-query is defined as follows: given a collection of spatial geometric objects *P* and a query object $q_{nn}$, return the object in *P* that is closest to $q_{nn}$.

#### 5.2.1 Transforming the NN-query

The R-tree exploits its minimum bounding rectangle structure to maintain the spatial relationship between its indexed data. When a NN-query is executed, this structure enables the R-tree to avoid investigating unnecessary data and only visit the optimal number of candidates [7]. However, as explained in Section 3, the nature of the Learned Spatial Bucket Index splits the data into a sequence of dimensions and thereby, to some extent, loses the spatial awareness. Therefore, it might seem counterintuitive to use such an index for NN-queries. Yet, from a theoretical point of view an NN-query can be regarded as a range query, or a series of such, centered at the query object $q_{nn}$. Say an NN-query is issued, the spatial area surrounding $q_{nn}$ can be queried by establishing a bounding box centered at $q_{nn}$, with a distance-bound $d_b$ from $q_{nn}$ in each cardinal direction, and thereby effectively converting it to a range query. Figure 9 shows such a scenario, where a bounding box (*Bound Box*) centered at $q_{nn}$ is created at a distance of the bound $d_b$ from $q_{nn}$.

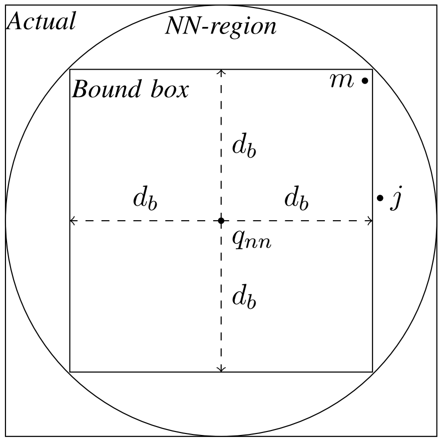

Fig. 9: Transformation of the NN-query to a range query.

The bounding box can be used to execute a range query $q_r$ in the same manner as explained in Section 5.1, but this time centered at $q_{nn}$. This effectively means that the NN-query is transformed to a range query; hence, when an NN-candidate set of objects are returned by the range query, this set must contain the true nearest neighbor to $q_{nn}$. However, due to the shape of the bounding box, this might not actually be the true nearest neighbor. In Figure 9, the bounding box illustrates the range of a given NN-query transformed to a range query. However, if only the area covering the bounding box is queried, it can potentially return a point, which is covered by the bounding box, but is not the true nearest neighbor. Figure 9 illustrates such a scenario where the point $m$ is contained by the bounding box, and therefore will be returned by executing a range query using this bounding box. However, the point $j$ is actually the true nearest neighbor to $q_{nn}$, but $j$ is not contained within the bounding box and therefore not returned by the range query. To correct this behavior, the bounding box will have to be expanded such that it at least covers the points contained in the circular region *NN-region*, as depicted in Figure 9. The correct bounding box is depicted in Figure 9 as *Actual*, which wraps the *NN-region* in a range query region ensuring the true nearest neighbor is returned. This effectively means that when an NN-query is transformed to a range query, the correct bounding box, *Actual*, will be utilized and the returning set of NN-candidates, covered by *Actual*, will have to be filtered to ensure that only points within the *NN-region* are considered during the NN-search. It might be the case that no NN-candidates are found within the queried region, which means that the query region will have to be expanded until at least a single candidate is found residing within the *NN-region*. Algorithm 1 shows the execution of an NN-query.

**Algorithm 1** $nearestNeighborQuery$

```
Input: Query object $q_{nn}$, Initial bound $d_b$
Return: Nearest neighbor to $q_{nn}$
1: $q_r \leftarrow convertToRangeQuery(q_{nn}, d_b)$
2: $candidates \leftarrow rangeQuery(q_r)$
3: while $candidates = \emptyset$ do
4:     $q_r \leftarrow expandBound(q_r)$
5:     $candidates \leftarrow rangeQuery(q_r)$
6: return closest object to $q_{nn}$ in $candidates$
```

First the NN-query is converted to a range query by establishing the query region surrounding $q_{nn}$ (line 1).

Secondly, range queries are executed, as explained in Section 5.1, until at least a single candidate is found residing within the *NN-region* by expanding the search area at each iteration (line 2-5). Lastly, when a non-empty candidate set is returned by the range query, the set is filtered to locate the closest point to the query object $q_{nn}$ (line 6).

The efficiency of the NN-query depends heavily on the size of the bounding box and thereby the bound distance $d_b$. A large bound means that the bounding box will cover a larger area of the dataset and therefore potentially return a large number of NN-candidates. As the distance from $q_{nn}$ to each candidate must be calculated in order to determine the actual nearest neighbor, the size of the candidate set will affect the performance of the NN-query. In contrast, a bound too small, such that the bounding box is empty, will force the bound to be expanded potentially many times, until at least a single candidate is found, where for each expansion a new range query must be executed to retrieve the new candidate set. Therefore, the choice of bound is essential and there exists a trade-off between a large candidate set and executing multiple range queries.

### 5.2.2 Bound Estimation

Choosing a bound is no trivial task and is highly dependent on the query, since an efficient bound for one query might prove extremely inefficient for another. Therefore, it is important that the strategy for choosing a bound takes into account the data distribution in the area surrounding the query object such that smaller bounds are chosen for high-density areas and vice versa for low-density areas. The one chosen with the needs of the estimation is Geohashing [8]. Geohashing hierarchically encodes geographic locations into a short string consisting of letters and digits, following a Z-order space filling curve, and provides arbitrary precision in terms of longer hashes. Spaces are divided into a grid shape which in turn are subdivided into smaller grids, thereby gradually increasing precision. All points within the same grid section have the same hash-value effectively meaning that points in proximity of each other will have similar hashes. An example of the hierarchical structure of Geohashing can be seen in Figure 10 where the hash region $9q8$ contains the regions $9q80$, $9q81$, etc. which in turn will contain hash regions one character longer.

In order to estimate a bound given a dataset, the dataset is hashed recursively until a single grid section

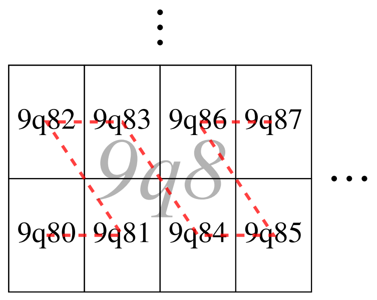

Fig. 10: Geohash hierarchically Z-order structure.

contains at most a specific number of points. Each grid section is assigned a bound based on the specific strategy chosen for the dataset e.g. best-case distance between points within the grid. This means that when a given NN-query is issued, the query object $q_{nn}$ is hashed to the maximum precision of the hashed dataset. The bound is retrieved by looking up the bound assigned to the grid section with the same hash as $q_{nn}$, this bound $d_b$ is then used to create the bounding box shown in Figure 9. If $q_{nn}$'s hash is not present, meaning that no point in the dataset was hashed to this grid, the hierarchical structure of Geohashing enables the removal of a character from $q_{nn}$'s hash thereby gradually decreasing precision until $q_{nn}$'s truncated hash is present. The truncating of $q_{nn}$'s hash makes intuitively sense, since if a hash with some given precision is not present, then the dataset contains no points in that grid meaning that the distance to a nearest neighbor is larger and the bound for the parent of the grid is a more appropriate approximation.

Multiple options exist regarding the determination of the bound assigned to each individual hash region. The optimal bound would exactly encapsulate the nearest neighbor of the query object, however this is impossible to know beforehand. Therefore, the bound will be an approximation of the distance to the nearest neighbor from the query object. As mentioned earlier, the bound has a substantial impact on the performance, since a bound too large results in a query region encapsulating many points and a bound too small would force the execution of multiple range queries.

Three different strategies were investigated, namely the best-, worst- and average-case distance between points residing in an individual hash region. The best-case strategy is optimistic, as it will approximate the nearest neighbor to the query object to be the distance between the two closest points in a given hash region away. This is rarely the case and will in most cases require multiple executions of the range query. In contrast, the worst-case strategy is pessimistic and will potentially result in a large number of points encapsulated in the range query. The average-case strategy will try to balance the two aforementioned strategies by calculating the bound to be the average distance between objects in a hash region. Section 6 will present more details with regards to which estimation strategy is chosen.

## 6. Evaluation

To assess the performance of the Learned Spatial Bucket Index, it is evaluated by three different types of tests. Firstly, the query performance of the index is evaluated on range and nearest neighbor queries on varying data distributions. Secondly, a scalability test is conducted to assess the index' performance as the data density and data size increases. Both evaluations are compared to an in-memory optimized R-tree [9]. Lastly, the accuracy of the models is evaluated by measuring the time needed to local search from the model prediction to the correct bucket. This is done to assess to what extent the model accuracy and local search impact the overall performance of the index.

**Datasets:** Three different datasets have been used throughout the evaluation. The scalability is evaluated on a dataset containing addresses and is split into random subsets of varying sizes; 5, 10, 20 and 40 million entries. The query performance is evaluated using the 10 million subset from the scalability test, a 10 million dataset containing locations of a different distribution and a synthetic dataset following a uniform distribution. An overview of the dataset can be seen in Table 1.

| Dataset | Size in mil. |
|---|---|
| OA5M | 5 |
| OA10M | 10 |
| OA20M | 20 |
| OA40M | 40 |
| Geonames | 10 |
| Uniform | 10 |

Table 1.: Datasets

**Training:** The learning component, as presented in the architecture on Figure 2, trains complex models using TensorFlow [10]. These models are able to approximate which bucket is covering a value supplied from an arbitrary query.

For each dataset, a neural network with 2 hidden layers using the ReLU activation function with 32 neurons is applied at Level 1, see Figure 7. The output layer uses a linear activation function and its output will be used to locate the bucket that covers the input of the neural network. On subsequent levels, linear regression models are trained, since they are fast to execute and cover a small chunk of data.

### 6.1 Experiments

The Learned Spatial Bucket Index is tested on three different configurations, which correspond to how many values a Level 2 bucket contains. The configurations are 2500, 5000 and 10000 values per bucket. Varying this size should give an indication of to which extent the performance is affected by having buckets of different sizes at the Level 2, and whether these splits are appropriate for a given dataset. Models have been trained at Level 1 and Level 2 for each of the datasets. The tests where conducted on a system with an i5 3.9GHz quad-core processor and 16GB RAM, running Windows 10.

#### 6.1.1 Range Query

Figure 11 illustrates the runtime of the different configurations for the Learned Spatial Bucket Index (LSBI) and the R-tree when testing how the range query manages different types of distributions.

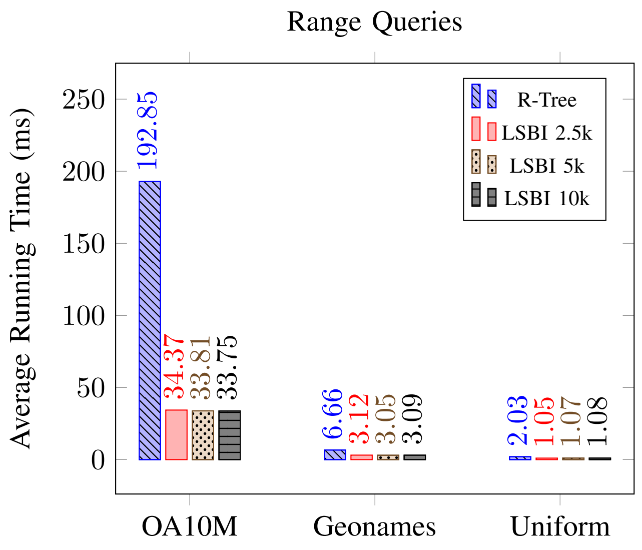

Fig. 11: Test results for running 1000 unique range queries on different configurations, where each query has a selectivity of 5%.

For all three datasets, the LSBI is able to outperform the corresponding R-tree with different speed-up factors. The LSBI is roughly twice as fast as the R-tree on the Geonames and Uniform datasets, while there is a much more significant speed-up on the OA10M dataset. As evident in the figure, the bucket sizes do not

| | Range Query | | | | | | Nearest Neighbor Query | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #Elements | OA5M | OA10M | OA20M | OA40M | Geo | Uni | OA5M | OA10M | OA20M | OA40M | Geo | Uni |
| 2500 | 3.89x | 4.38x | 4.36x | 2.02x | 6.68x | 5.83x | 3.60x | 1.66x | 1.08x | 0.23x | 2.06x | 4.05x |
| 5000 | 2.71x | 4.08x | 2.40x | 2.62x | 6.99x | 5.77x | 3.58x | 1.68x | 0.95x | 0.26x | 2.10x | 3.99x |
| 10000 | 2.10x | 3.80x | 2.56x | 1.68x | 6.62x | 7.73x | 3.58x | 1.59x | 0.87x | 0.25x | 2.06x | 3.97x |

Table 2.: Speed up factors for both Range and Nearest Neighbor queries run on the Learned Spatial Bucket Index, using the R-tree as a base case.

influence the running time of the queries. This indicates that the buckets produced are sufficiently linear despite the naive splitting, since if the overall data exhibits a linear relationship between the data points and their position, the data is still going to retain the linear relationship regardless of where the split occurs.

In Figure 12 the results for how the LSBI scales are depicted. The OA dataset is used in varying sizes to measure how the index scales compared to the scalibility of the R-tree.

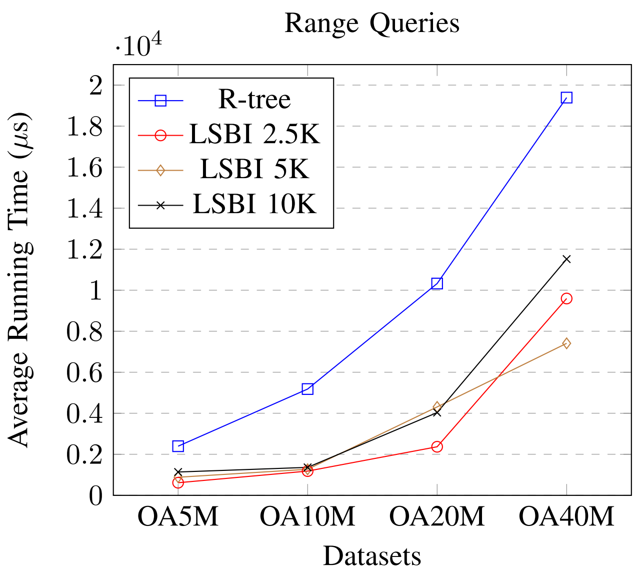

Fig. 12: Test results for running 5000 range queries on different configurations.

There is generally a similar relationship between the average running times and the data size for all the configurations and the R-tree; for all cases the average running time increases at the size of data increases. Every configuration of the LSBI, however, is able to outperform the R-tree on every dataset, which indicates the same may be the case for even larger datasets. An important note to make, is the gradual increase of the runtime of the configurations, as there is a single configuration that increases less than other configurations as the data size increases, namely the LSBI 5K configuration. This configuration is also the fastest of the three, which implies that a bucket size of 5k is better for larger datasets.

It is however generally difficult to make a conclusion based on the bucket sizes, as the LSBI 5K configuration is the slowest configuration on OA20M, but the fastest in OA40M. This can be due to a multitude of factors. Firstly, as evident in Figure 12 the bucket size influences the performance of LSBI more significantly when the data size increases. This indicates that when increasing the data size beyond 10 million, the overall relationship between the data points and their position is not as linear as for 5 and 10 million. Therefore, the naive splitting approach will influence the performance of LSBI. Secondly, having 5k data points per bucket may coincidentally fit well in OA40M, but not with OA20M. The buckets produced in OA40M may therefore be more learnable and the cost of the local search after leveraging the model for a prediction may therefore be lower. Lastly, having larger buckets means there is a higher likelihood of each bucket having smaller minimum values for each dimension and larger maximum values for each dimension. This means, that when a bucket is predicted, and its bounds are checked for whether the bucket satisfies the query range, there may be more cases where a bucket's bounds are satisfied by the range, but the bucket does not contain the correct combination of data points that satisfy the range. This means a bucket that should have been pruned, is not pruned. And due to the larger size of the buckets, the likelihood of the above-mentioned scenario happening may be larger, since more points per buckets increases the odds of having smaller and larger minimum and maximum bounds.

Trying to identify the factor which impacts the running times of the configurations in Figure 12 the most, is difficult, as there always is a trade-off to consider. Larger buckets gives the model more data to train on, but can make pruning harder. Reversely, smaller bucket may produce more prunable buckets, but instead give poorer model accuracy as each model now has less data to train on. As the buckets for each dataset is different, there are most likely cases where one factor impacts the running time more than the other factor, and vice versa. It is therefore difficult to ascertain which configuration works best when. A more advantageous approach would be to revise the bucketing strategy, such that this would produce buckets that are more dynamic in accordance to the data, and would therefore possibly balance the trade-off between having larger or smaller models. This is left as a future work. Nevertheless, in the current scenario, fixed size buckets still produce better running times than that of the R-tree. For this type and size of data, static bucket sizes are therefore a viable strategy.

### 6.1.2 Nearest Neighbor Query

The NN-query implementation has been evaluated on 5000 queries with the same distribution as the data. Figure 13 shows the results of these queries on the same three different datasets of size 10 million as in the range query. Here, it is evident that the same relationship applies for the different configurations of the index, while all are still able outperform the corresponding R-tree for the dataset. This further proves that the arguments made about the bucket size for range queries, presented in regards to Figure 11, also apply for NN-queries; the bucket size is less relevant on smaller datasets.

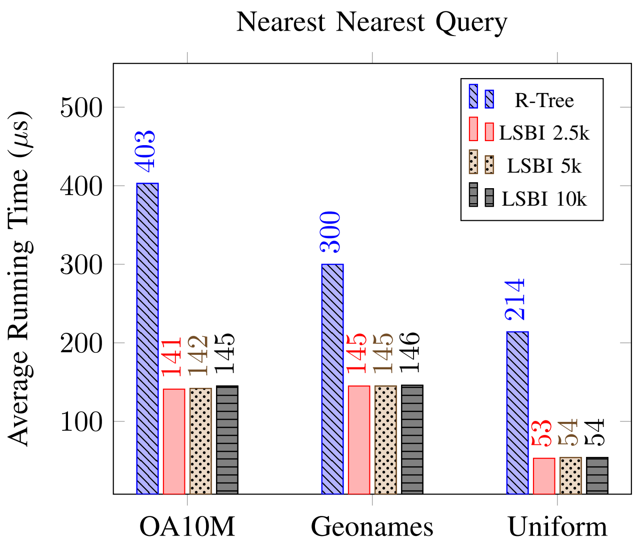

Fig. 13: Test results for running 5000 nearest neighbor queries on different configurations.

Figure 14, depicts how the Learned Spatial Bucket Index performs when scalability is investigated. The performance of the index is dependent on the size of the dataset. On both OA5M and OA10M, the index is outperforming the corresponding R-trees, but as the data size increases to 20 million and above, the performance of the R-trees begins to match the index, and eventually outperform the index. There are multiple factors as to why the performance of LSBI staggers in comparison to the R-tree. Many of these factors are the same as those described in the evaluation of the scalability of the range query: non-linear relationships with the buckets, the lack of prunable buckets, etc. In addition, as the size of the data increases so does the density, which means that the candidate set will grow thereby impacting the performance of the LSBI, as mentioned in Section 5.2.1.

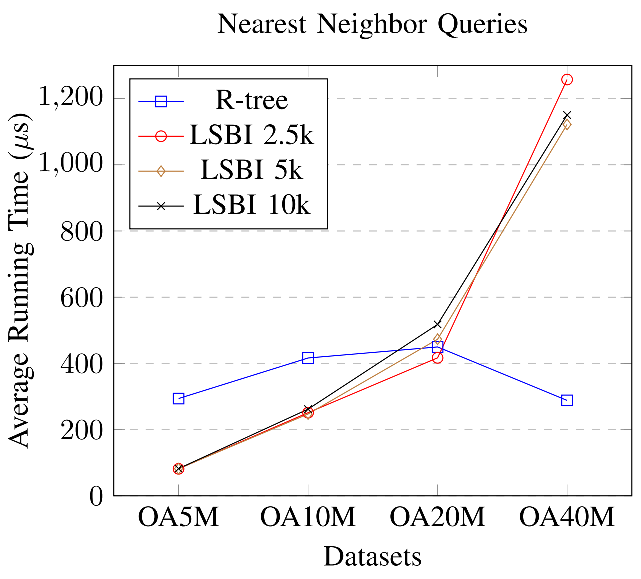

Fig. 14: Test results for running 5000 nearest neighbor queries on different configurations.

As evident in Figure 14 the performance of the R-tree exhibits an anomaly as it performs better on the 40 million dataset than on any of the other datasets. This can be due to different factors such as the structure of the tree which might prove more optimal with the 40 million dataset. This structure could enable the R-tree to more aggressively prune the search space.

The overall performance evaluation of the LSBI on NN-queries shows that in some scenarios it is able to outperform the R-tree. This indicates that it is possible to apply ML models to NN-queries, and open new directions for spatial indexes. However, in some cases considering the NN-query as a range query is too simple and here the R-tree will outperform LSBI, hence other strategies and optimizations are needed in order to completely outperform the R-tree for this type of query.

### 6.1.3 Model Accuracy

The performance of the index is directly tied to the accuracy of the top and second level models. To assess the efficiency of the models, these have been tested by executing 5000 range queries, and measuring the time it takes to correct the model predictions to the index of the correct bucket. Figure 15 shows the these times in nanoseconds for the model at Level 1.

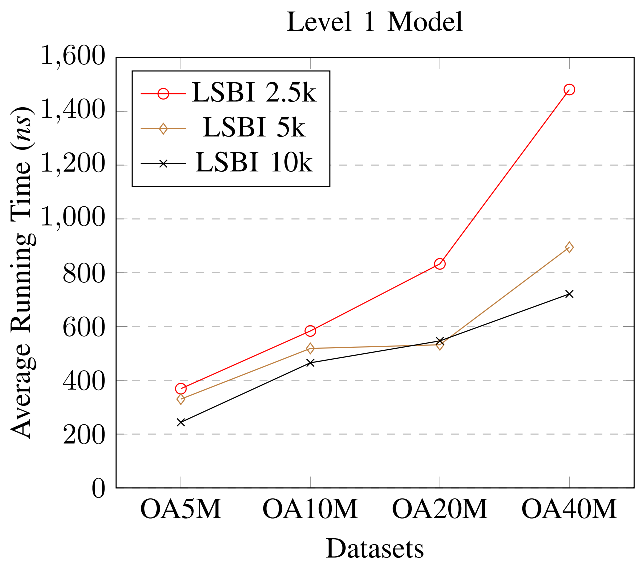

Fig. 15: Average running times of the Level 1 model's local search. Note that time is in ns.

Here, the average running time is illustrated for the correction of each bucket configuration on each dataset. In general, it takes longer time to correct the predictions of the model at Level 1 the less data each bucket covers. This intuitively makes sense, since the less data each model can train on, the lower the accuracy becomes, therefore resulting in worse predictions. However, the graphs of all configurations indicate that the average running time worsens as the data size increases. Since the bucket sizes do not change when the data size does, and the running time worsens, the issue is likely due to the fixed-size nature of the buckets, with the splits of the buckets occurring at every 2500, 5000 and 10000 data points. These fixed-size naive splits might prove to be suboptimal, since this strategy can produce buckets wherein similar data is split into multiple buckets, i.e. data residing in a given bucket might be dissimilar, making it difficult for the Level 1 neural network to approximate a function capable of separating the buckets. These splits can result in worse predictions and therefore also worse correction times. The best average correction time for the model at Level 1 is the configuration OA5M-10k with a time of 243.94 ns, whereas the worst is OA40M-2.5k with a time of 1481 ns. This again highlights the advantage of having larger buckets. However, as evident in Figure 12, smaller buckets produce better overall execution times of the Learned Spatial Bucket Index, indicating that the process of local search is but a small factor in the overall performance of the index.

Figure 16 shows the results of the same procedure as mentioned above, but for Level 2 models.

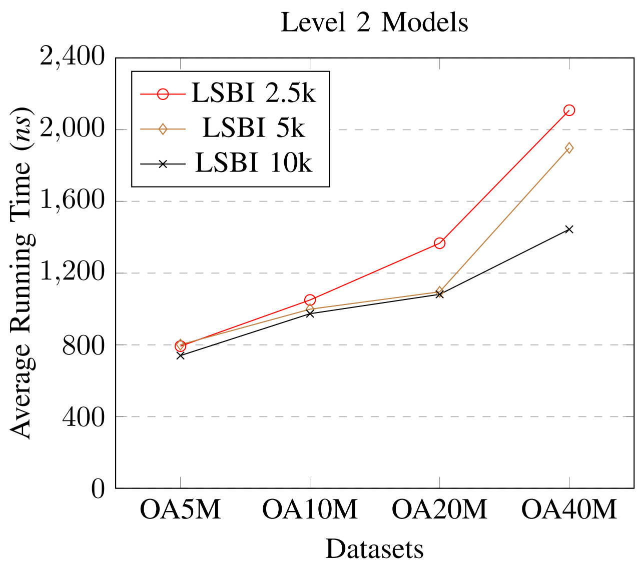

Fig. 16: Average running times of the Level 2 models' local search. Note that time is in ns.

The graph shows the same pattern as in Figure 15, but with a higher average running time. This indicates that the performance of the Level 2 models is also highly dependent on the bucket size, with larger buckets producing better predictions, but that the Level 2 models in general have a higher average running time of the correction, due to worse predictions. These predictions are again influenced by the bucketing strategy possibly producing buckets containing dissimilar data.

## 7. Related Work

In [2], the concept of R-trees is introduced. R-trees are highly relevant for this paper, as R-trees serve as the baseline index for the comparison of the learned indexes. Furthermore, the concept of MBRs is important to understand, since MBRs are a structure that is necessary to execute queries, since these are able to contain and conserve the spatial relationship between its data.

This paper is a spiritual continuation of our previous paper, [11], wherein we also seek to create indexes with the use of learned models. In that paper, we had a focus on point and nearest neighbor queries, where we were able to answer point queries at a faster rate than an R-tree. The index for the nearest neighbor queries, however, did not perform as well, and many of the findings from [11] has helped build the foundation of this paper.

The paper [4] is heavily relevant for the indexes described in this paper. Here they describe the process of ”sorting and partitioning points along a sequence of dimensions into equally-sized cells”. This is at the core of the bucketing strategy that is used in this paper, which we have altered to fit spatial data.

In [3], they introduce the original concept of using machine learning to create models that have better performance than index counterparts such as B-trees and Bloom filters. This paper is highly relevant for understanding the core concepts of learned models and the relevance of applying learned models in old computer science fields.

In [6], they show how large neural networks can be split into smaller gated networks thereby avoiding the invocation of the complete model each time. The structure of the Learned Spatial Bucket Index is inspired by this splitting of large models, such that expert models are trained on subsets of the data which might yield better accuracy than a large network. The notion of leveraging multiple models is also used in [3] to create their Recursive Model Index.

## 8. Conclusion

In this paper, we have investigated the application of learned indexes in a spatial context. Based on previous work, such as [3] and [4], we have taken principles proven true for one- and multidimensional relational data and extended these to spatial data. We describe the concept of bucketing, which makes it possible to create a learnable and correctable ordering, and use this strategy to build models over the data. Based on these models, we built an index, where we were able to execute range and nearest neighbor queries. We have shown that using ML to build spatial indexes can provide significant increase in performance. We have shown that this is true for range queries while varying the data size and distribution. However, while the Learned Spatial Bucket Index is able to perform NN-queries faster than the R-tree, there are still some cases where the index needs further investigation, to ensure the performance still applies for larger datasets. While our index currently supports fewer query types than the R-tree, we argue that our learned index for range and NN-queries showcases the viability and potential of learned indexes for spatial data.

## 9. Future Work

This section describes possible directions to which the research can be taken in the future.

**Additional Query Types:** The structure of the Learned Spatial Bucket Index may potentially allow for other query types to be possible such as spatial joins and kNN-queries. The same goes for other types of queries such as inserts, deletes and updates, however these may introduce complications, as the models trained on the data may have to be retrained to ensure correctness.

**Bucketing Strategies:** The current configuration of bucketing is to create equally sized bucket by setting a constant for the bucket size. This has the advantage of ensuring each bucket has enough data to train accurate models, however this may produce splits between relatively close data points. It could therefore be advantageous to investigate other bucketing strategies, that are able to produce learnable buckets that also happen to be optimal. One strategy could be to split the data on each dimension such that the learnability of the pattern within the data of each bucket is maximized.

**Estimators for NN-queries:** Currently, the range query is expanded every time it is not able to find a nearest neighbor. The expansion is based on the distance between the data points that are residing with the Geohash of the range query. However, the nature of data and the placement of the NN-query may result in many iterations of the range query until it is able to find a nearest neighbor. Investigating an improved estimator may therefore be able to heavily cut down on the number of range query iterations, when these occur.

## Acknowledgment

We would like to thank our supervisor Hua Lu for proposing this research direction and for his guidance, by providing feedback and helping us in resolving any issues we had during the course of project.

Furthermore, we want to thank Tim Kraska for providing us with details with regards to questions that arose when studying some of his work.

Lastly, we want to thank our friends and family for supporting us throughout the process of this paper.

## References

[1] R. Bayer and E. M. McCreight, “Organization and maintenance of large ordered indexes,” *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972. [Online]. Available: http://link.springer.com/10.1007/BF00288683

[2] A. Guttman, “R-trees,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD ’84*, vol. 14, no. 2. New York, New York, USA: ACM Press, 1984, p. 47. [Online]. Available: http://portal.acm.org/citation.cfm?doid=602259.602266

[3] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The Case for Learned Index Structures,” 2018. [Online]. Available: https://doi.org/10.1145/3183713.3196909

[4] 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,” Tech. Rep. [Online]. Available: http://alexbeutel.com/papers/CIDR2019_SageDB.pdf

[5] TPC, “TPC-H - Homepage.” [Online]. Available: http://www.tpc.org/tpch/

[6] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “OUTRAGEOUSLY LARGE NEURAL NETWORKS: THE SPARSELY-GATED MIXTURE-OF-EXPERTS LAYER,” Tech. Rep. [Online]. Available: https://arxiv.org/pdf/1701.06538.pdf

[7] H. Lu, “SW7 Data Intensive Systems (DIS) 2. Spatial Queries,” Tech. Rep., 2017.

[8] G. Niemeyer, “Geohash— Labix Blog,” 2008. [Online]. Available: https://blog.labix.org/2008/02/26/geohashorg-is-public

[9] M. Hadjieleftheriou, “libspatialindex libspatialindex 1.9.0 documentation.” [Online]. Available: https://libspatialindex.org/

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, and G. Brain, *This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16). Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX. TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning*. [Online]. Available: https://tensorflow.org.

[11] M. Folmer, R. Neumann, and T. Karunanithi, “Learned Indexes for Replacing Spatial Index Structures,” Tech. Rep.