Short paper

The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries

Angjela Davitkova
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
davitkova@cs.uni-kl.de

Evica Milchevski
Commerce Connector GmbH
Stuttgart, Germany
evica@commerce-connector.com

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

We present the ML-Index, a memory-efficient Multidimensional Learned (ML) structure for processing point, KNN and range queries. Using data-dependent reference points, the ML-Index partitions the data and transforms it into one-dimensional values relative to the distance to their closest reference point. Once scaled, the ML-Index utilizes a learned model to efficiently approximate the order of the scaled values. We propose a novel offset scaling method, which provides a function which is more easily learnable compared to the existing scaling method of the iDistance approach. We validate the feasibility and show the supremacy of our approach through a thorough experimental performance comparison using two real-world data sets.

1 INTRODUCTION

Processing queries on multidimensional data is a classical and thoroughly investigated problem. A plethora of index structures provides mechanisms for compactly storing and querying multidimensional datasets, applied in countless application scenarios. A recent idea proposed by Kraska et al. [4], suggests improvement and replacement of traditional index structures with machine learning and deep-learning models. They in particular successfully replace the B-Tree with a recursive learned model that maps a key to an estimated position of a record within a sorted array. With the use of the proposed learned models, they are able to utilize the patterns in the data distribution, resulting in an improvement of both the memory consumption and the execution time over the traditional index.

To provide a generalization of the B-Tree for multiple dimensions, the data needs to be sorted in an order which can be easily learned by supervised learning models. The ordering needs to be done in such a manner that the correctness guarantees are fulfilled when answering range and KNN queries. In practice, techniques, such as the Morton and the Peano-Hilbert order, can be exploited for sorting multidimensional data. However, directly mapping the multidimensional data points within the aforementioned orders cannot be easily learned by deep learning models. Kraska et al. [3] propose an approach for learning an order based on successively sorting and partitioning points along several dimensions into equally-sized partitions. However, choosing only a subset of dimensions may lead to performance degradation when the number of dimensions increases and the deduction of the partition neighbors may not be a time-efficient task.

Therefore, we create a novel Multidimensional Learned (ML) index which generalizes the idea of the famous iDistance scaling method [2] and uses the scaled ordering in combination with a two-layer learned index, to answer multidimensional queries. Unlike existing indexes, it captures the data distribution in two manners, by efficiently partitioning and scaling the data with respect to distribution-aware reference points and by learning the distribution of the sorted scaled values. Harnessing the power of the deep-learning models, the ML-Index is the first complete learned index, able to answer point, range and KNN queries efficiently while having a low memory consumption.

2 RELATED WORK

Until recently, limited research was present in improving data indices by interweaving them with machine learning. One of the first distribution aware indices [1] focuses on the combination of R-Trees with a self-organizing map. Another exact KNN algorithm [9] employs k-means clustering and triangle inequality pruning for efficient query execution.

Recently, Kraska et al. [4] draw the focus on a novel idea of substituting indices with deep learning models. Lead by the assumption that each index is a model, they draw a parallel between the indices and their respective analogue in the machine learning world. For instance, B-Tree Index and Hash-Index can be seen as models that map a key to a position within a sorted and unsorted array accordingly and can be easily replaced with neural network models. A learned database system called SageDB [3], extends the concepts to multidimensional data, by successively partitioning points along a sequence of dimensions into equal-sized cells and ordering them by the cell that they occupy. Although the order produces a layout noted as learnable, the complexity of directly learning the projection of n-dimensional points to an order position increases with the increase of dimensionality. Even though limiting the number of dimensions for partitioning avoids the added complexity, it results in slower execution of range queries including the missing dimensions.

Providing an inexpensive representation of multidimensional points which can be meaningfully sorted has been already widely explored, such as presorting the data by their Z-order [6], Hilbert order [5], or the respective distance to reference points [2]. A learned Z-order Model [8] focuses on combining the Z-order scaling with a staged learned model, for efficiently answering spatial queries. Although applicable for smaller dimensions, both the Z-order model and the UB-Tree are limited when dealing with a larger number of dimensions, which will be analyzed upon the experiments and the direct comparison.

3 THE ML-INDEX

The ML-Index is a compound of two main components, as illustrated in Figure 1. Its creation is carried out in two stages, guided and generalized by the idea of previously existent iDistance index [2]. The upper part consists of a set of reference points, responsible for scaling the multidimensional data to one-dimensional values, which can be easily sorted. The lower part is a Learned Model used for learning the distribution of the scaled values and a sorted array used for searching and storing the data.

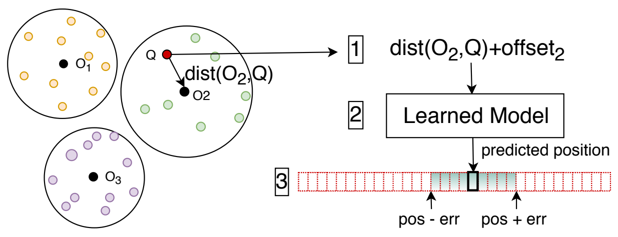

Figure 1: The ML-Index Approach

3.1 Scaling Methods

Consider a set of data points $d_l \in D$, where $d_l = (d_0, d_1, ..., d_n)$ is in an n dimensional metric space. The first stage, responsible for creating the upper part, maps the data points from a n into a one-dimensional data space. The scaling method aims to group points that are similar to each other and project them in one-dimension, such that the similarity is preserved by the proximity of the one-dimensional values. To efficiently do the scaling, m reference points O_i are chosen, each representing an identification or a centroid of the data in partition P_i. The partition P_i is formed from the points whose closest reference point is O_i. This means that the minimal distance of a point d_l to the reference points determines the appropriate partition.

Irrespective of the way to find reference points, we elaborate on the usage of different scaling methods and we discuss their advantages and disadvantages. The straightforward case is scaling d_l with respect to a single reference point, by mapping it to a one-dimensional value $key = dist(O_i, d_l)$. Although applicable for smaller datasets, the method creates a considerably large amount of false positives, produced by mapping multidimensional points that are far from each other to the same value.

The second scaling method is the iDistance method described by Jagadish et al. [2]. This method maps a data point d_l into a one-dimensional value key, based on $key = i * c + dist(O_i, d_l)$, where i is the index of the closest reference point O_i. The constant c serves to partition the points into predefined ranges and based on its value it stretches the ranges differently. Using the constant, the points belonging to a partition P_i will be mapped to a range $[i * c, (i + 1) * c]$. Although it provides well-scaled values and drastically reduces the number of false positives, it is highly dependent on the parameter c. A smaller value for c may create an overlap between the partitions, causing multidimensional points from different partitions to be mapped to the same scaled value. Hence, upon search, the number of unnecessarily examined points will be increased. On the other hand, having a larger c directly affects the creation of the second stage of the ML-Index which we thoroughly discuss in Subsection 3.2.

Finding the right value for the constant c, which will cause no overlap and have perfectly ordered partitions, with no gaps between them, is impossible. This comes as no surprise since reference points correspond to partitions of different sizes. Prompted by the previous, we propose a novel scaling approach that provides a perfect ordering between the ranges of different partitions and overcomes the problem of overlapping. The new method termed *offset method*, given a point d_l and its closest reference point O_i, calculates the scaling value as $key = offset_i + dist(O_i, d_l)$, where $offset_i$ is different for every partition P_i. Given an arbitrary ordering of the reference points $O_1, O_2, ..., O_m$ and their adequate partitions $P_1, P_2, ..., P_m$, the offset is calculated as the sum of the radii of their previous partitions:

$$offset_i = \sum_{j<i} r(j)$$

where r is the maximal distance from O_j to the points in partition P_j. This method assures no overlap and it reduces the gap between the partitions, which is crucial regarding the performance of the second stage of the ML-Index. Additionally, it omits the problem of tuning the parameter c, for a suitable range creation. The downside in comparison to iDistance is an unnoticeable memory increase, caused by storing the offsets. Following the scaling of the original data, each data point d_l is associated with a value key. Multiple points can have the same scaled value. Before the execution of the second stage of the ML-Index, we sort the points according to the value key. The resulting order is used as a starting point for creating a learned model, which efficiently predicts the position of a given key within a sorted array.

3.2 Learning the Order

The second stage of the ML-Index represents a *Recursive Learned Index*, similar to the one described by Kraska et al. [4]. The index, mimics the behavior of a traditional B-Tree, by mapping a given lookup key within a sorted array with the guarantee that the key is within proximity of the predicted position $[pos - err, pos + err]$. As observed [4], a model, which performs this task, effectively approximates the cumulative distribution function (CDF), modelled as $p = F(key) * N$, where N is the number of keys and $F(key)$ is the estimated CDF that estimates the likelihood to predict a key smaller or equal than the lookup key. The learned index is built in a top-down manner where a model in each stage provides a prediction used to choose the model in the next stage, or the position of the key when the final stage is reached. For model $f(x)$ from models M_l at stage l, with input x, the loss is calculated:

$$L_l = \sum_{x,y} (f_l^{\lfloor M_l f_{l-1}(x)/N \rfloor}(x) - y)^2 \quad [4]$$

Although Kraska et al. [4] suggest using multiple stages of the learned index, we use only two. The incentive behind this decision is that if the second stage index produces a larger than expected error, the increase in the number of models in the second stage is sufficient to reduce the error. Additionally, unlike their proposed learned index, the second stage of learned models in the ML-Index is constructed solely by using a regression, with the purpose of balancing the construction time and the performance of the learned index. By using a simpler model, the number of multiplications and additions upon search is reduced, leading to lower search time. The final prediction of the learned model is a predicted position within the sorted array where the key, in our case the scaled value is stored. Because a position is predicted with a certain error, one must also search within the error bounds around the prediction for the correct position.

As mentioned, the different scaling methods impact the second stage of the ML-Index. Since the naive scaling method is infeasible, we elaborate only on the impact of the iDistance and the offset scaling method. Both methods result in different functions that need to be estimated by the learned index. When considering the iDistance we distinguish two scenarios, one with overlap between the partitions and one without. When c is smaller than the maximum radius of all the partitions, then an overlap between the ranges of the partitions is inevitable. However, sorting the data with a smaller c creates a function which can be easily learned. This is a result of the overlapping that leads to a larger density of the scaled values which results in a fairly continuous function. The second case is having a larger c, that avoids overlaps, but creates large gaps within the function. Therefore, the neural network will learn the data present in the different ranges. However, upon search time it is possible to search for a point that is not present within these ranges, but it is located within the gaps of the function. Vast research has been proposed for filling the missing values in a function to be learned, however, this leads to a pre-processing step which can be easily avoided by using the offset method. The offset method circumvents the gaps created by the different maximal sizes of the partitions and provides a better "learnable" function in which the missing values may only appear due to sparsity in the clusters and not the scaling method.

4 QUERY PROCESSING

Point Query: The point query identifies the existence of a multidimensional point within the index and it is executed in three steps as shown in Figure 1. The first step includes searching for the closest reference point O_i to the query q and calculating the scaled value $key = offset_i + dist(ClosestO_i, q)$, where $offset_i$ is calculated as $i * c$ for the iDistance scaling and appropriately for the offset method. Once calculated the key is used to predict the position of the point q, within the sorted array of points. The predicted value is the position pos of the key within the sorted array, and it is used in its exponential search with bounds $[pos - err, pos + err]$. The complexity of the first step, for m reference points and d dimensions, is $O(m * d)$. The learned model complexity depends on the architecture of the neural network. A neural network with a single hidden layer with width h and input size N will have $O(hN)$ multiplications and additions.

KNN Query: Given a query q and a parameter k, the KNN query finds the closest k points S_k, to the query such that $\forall d_i \in S_k,\ \forall d_j \in D \setminus S_k,\ dist(q, d_j) \geq dist(q, d_i)$. Since the iDistance was initially created for executing KNN queries, we adapt the algorithm for the ML-Index. The algorithm creates several one-dimensional range queries, whose selectivity expands until the result is complete. The major modification is locating the start of the range. For this purpose, the Algorithm 1 is used, that exploits the learned model to predict the position of a given key within a sorted array. However, since a key which is not present within the array can be provided, we need to search for the key with the smallest difference to the initial key. Nonetheless, searching for the closest key to the initial key is not only dependent on the bounds provided by the error of the learned model but also the bounds of the ranges occupied by the reference points. Therefore, we modify the binary search to also include the $offset_i$ and $offset_{i+1}$, which further reduces the search space. The method *closest* returns the position of the value closest to the key in the range $[offset_i, offset_{i+1}]$. To describe the need for the second bounds, we consider having two reference points and their respective ranges within the array $[1, 5][6, 10]$. Let's further take the assumption that the first range has the keys $[1, 3, 5]$ and the second $[7.5, 8, 10]$. Upon search, we want to find the closest key to $key = 6$ for the reference point O_2. If we do not consider that the reference point O_2 has a lower bound 6 we would retrieve the key 5 as closest which does not belong to the region 2 and thus, in reality, may not be even close to the query point.

Range Query: Widely applicable for smaller dimensions, the range query $q = q_1, q_2, ..., q_n$, where $q_j = [bound_{min}, bound_{max}]$, defines bounds of a dimension j, retrives the data points $d_i \in D$

Algorithm 1 Predict Closest Position

```
Require: scaled value $key$ (if outside of range, set to $offs_i$ or $offs_{i+1}$),
    range for $O_i$ $[offs_i, offs_{i+1}]$
 1: $mid = predict(key),\ s = mid - err,\ t = mid + err$
 2: while $s \leq t$ do
 3:     $mid = \lceil (s + t)/2 \rceil$
 4:     if $data_{mid} == key$ then
 5:         return $mid$
 6:     $withinRange = True$
 7:     if $data_{mid}$ outside $[offs_i, offs_{i+1}]$ then
 8:         $withinRange = False$, set $s$ or $t$ to $mid$
 9:     if $withinRange$ or $|s - t| \leq 1$ then
10:         if $key < data_{mid}$ then
11:             if $key \geq data_{mid-1}$ then
12:                 return $closest(data, key, mid - 1, mid, offs_i)$
13:             $t = mid - 1$
14:         else
15:             if $key \leq data_{mid+1}$ then
16:                 return $closest(data, key, mid, mid+1, offs_{i+1})$
17:             $s = mid + 1$
```

where $\forall j \in n,\ d_{ij} \geq q_{j0}$ and $d_{ij} \leq q_{j1}$. For the execution of the range query within the ML-Index, we adapt the Data-Based Method for range approximating suggested by Schuh et al. [7]. The algorithm iterates over the reference points and for each reference point calculates the closest and the furthest point from the given range q. For the computed furthest and closest point, the algorithm issues a range query of the form $[dist(O_i, point_{closest}) + offset_i,\ dist(O_i, point_{furthest}) + offset_i]$. Since the closest and furthest points can also have keys which are not present within the sorted array, the method described in Algorithm 1 is used.

5 EXPERIMENTS

For evaluation, all indices are in main memory and implemented in Java. The learned model is implemented with TensorFlow and it is extracted to omit the overhead. Its width is set according to the dataset. The reference points selection is done using the KMeans algorithm, due to better results for the real-world data. GMeans was not considered, since the branching factor of the M-Tree is set to k, for a fair comparison. We compare the following structures: the ML-Index, the iDistance index [2] with keys computed by our offset method, the M-Tree, and the index based on learning a Z-order, ZM-Index [8]. Two real-world datasets were used, Color Histogram (dim: 32, points: 68040, size: 19.5 MB) and Forest Cover Type (dim: 10, points: 581012, size: 68.7 MB).

Scaling Methods Comparison: Figure 2a shows the absolute error of learning the order produced by the different scaling methods, by varying the width of a single layer neural network. For the iDistance, c is once set to produce 10% and once 0% range overlap between the different reference points. The data is directly learned in several epochs, without preprocessing. The error produced by the offset scaling method is much lower from the error by the iDistance method when no overlap occurs. Differently, due to the density of the values, the approach with an overlap performs slightly better. However, an overlap between the ranges will result in slower query execution, therefore, the offset method performs the best when considering both aspects.

Memory: Figure 2b (Note the log scale 2) shows the memory consumption of the M-Tree, the iDistance and the ML-Index for both datasets. As observed, the ML-Index has a drastic reduction in memory and has only small storage required for the learned model and the offset distances. Increasing the number of clusters

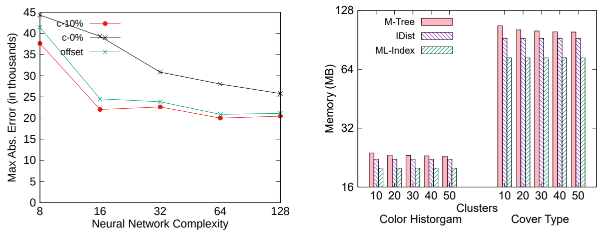

Figure 2: Scaling methods comparison (left) and memory consumption when varying datasets and clusters (right)

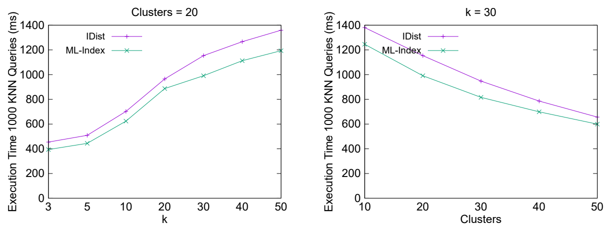

Figure 4: KNN search when varying k (left) and clusters (right) for Forest Cover Type dataset

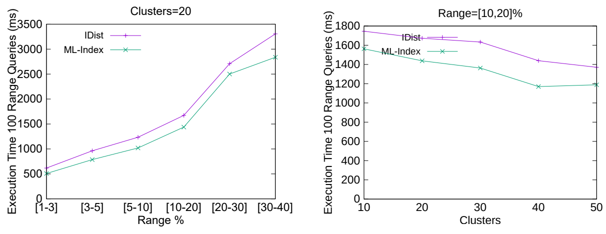

Figure 3: Range search when varying range selectivity (left) and clusters (right) for Forest Cover Type dataset

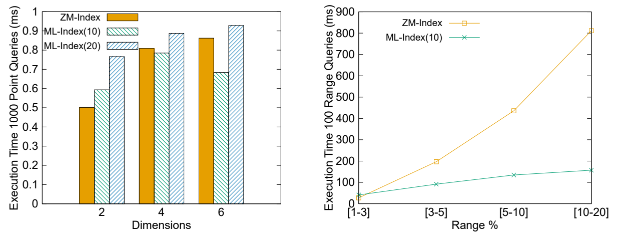

Figure 5: Point search (left) and range search (right) comparison with ZM-Index for Forest Cover Type dataset

results in an unnoticeably small increase in memory due to the number of offset values.

Query Execution: Since the M-Tree is drastically slower, we show only the comparison between iDistance and ML-Index. Figure 3 shows the execution time of 100 range queries by varying the query selectivity, representing the fraction of points within the query range from the total number of points, and the number of clusters used for choosing the reference points. Figure 4 shows the execution time of 1000 KNN queries by varying k and the number of clusters. The ML-Index is always faster than the iDistance, which is a result of the superiority of the search with the learned model over the search with B+Tree. This is especially visible for larger ranges and k values where the result may belong to more clusters, requiring multiple predictions from the learned model, causing a more visible difference in the execution time. Upon an increase of clusters, the difference between the search time of the ML-Index and iDistance is smaller. Assuming we search for the closest point, we need to first find the closest cluster among m clusters, which is performed the same for the iDistance and ML-Index. Hence, when m is large, the execution will be impacted more by the cluster iteration than the prediction.

Comparison with learned ZM-Index: For comparison with ZM-Index, we extract 2, 4, and 6 dimensions from the Forest Cover dataset, we scale the values to reduce the number of bits and the large gaps between successive Z-values, which produce a slower execution time. We always use our learned model, since having the number of neurons mentioned in [8], results in incomparably large execution time. ZM-Index outperforms ML-Index when searching for a two-dimensional point, as seen in Figure 5a. This is intuitive since a search through multiple clusters requires more time than a bit shifting operation, which is not the case for larger dimensions and a smaller number of clusters. When considering the range query comparison in Figure 5b, the ZM-Index performs far worse. Upon searching for a next Z-value which is within the range, we exchange every bit accordingly, resulting in a longer execution when dealing with both large numbers and dimensions. More importantly, in each step, the next Z-value within range is calculated, which may correspond to a value not present within the dataset and thus it will lead to unnecessary access.

6 CONCLUSION

We addressed the problem of replacing multidimensional indices with a learned, distribution-aware ML-Index. The ML-Index entails two core tasks: representing the dataset by one-dimensional values based on reference points chosen with respect to the data distribution, and a learned model capable of accurately learning the order of the values. Experimental results demonstrated the feasibility of the approach and its superior performance compared to state-of-the-art competitors. Future work includes rendering the index resilient to updates by observing degenerated performance and triggering retraining only when necessary.

REFERENCES

[1] G. Phanendra Babu. 1997. Self-organizing neural networks for spatial data. *Pattern Recognition Letters* 18, 2 (1997), 133–142.

[2] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.* 30, 2 (2005), 364–397.

[3] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*. www.cidrdb.org.

[4] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *CoRR* abs/1712.01208 (2017).

[5] Jonathan K. Lawder and Peter J. H. King. 2001. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *SIGMOD Record* 30, 1 (2001), 19–24.

[6] Volker Markl. 1999. MISTRAL: Processing Relational Queries using a Multidimensional Access Technique. In *Ausgezeichnete Informatikdissertationen*. Teubner, 158–168.

[7] Michael A. Schuh, Tim Wylie, Chang Liu, and Rafal A. Angryk. 2014. Approximating High-Dimensional Range Queries with kNN Indexing Techniques. In *COCOON (Lecture Notes in Computer Science)*, Vol. 8591. Springer, 369–380.

[8] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. IEEE, 569–574.

[9] Xueyi Wang. 2011. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *IJCNN*. IEEE, 1293–1299.