Institut für Informatik, Universität Zürich

MSc Project Report

Implementing Learned Indexes on 1 and 2 Dimensional Data

Neeraj Kumar, Nivedita Nivedita, Xiaozhe Yao

Matrikelnummer: 19-765-189, 19-756-303, 19-759-570

Email: `{neeraj.kumar,nivedita.nivedita, xiaozhe.yao}@uzh.ch`

June 2, 2021

supervised by
Prof. Dr. Michael H. Böhlen and
Mr. Qing Chen

University of Zurich^UZH

Department of Informatics

D B T G

Databases use indexes to find records efficiently. Among these indexes, B-tree and KD-tree are two successful indexes used for 1-dimensional and 2-dimensional data. In this project, we implement both the learned index [1] for 1-D data and the learned index, named LISA [2] for 2-D data from scratch using Python. Afterwards, we have conducted sanity check to ensure that our implementations are correct. We then perform several experiments to compare the performance of learned indexes and compare them with classic tree-based indexes.

In addition to the implementation and evaluation, we have theoretically analyse some properties that the learned indexes hold. Beyond that, we also explore and discuss some common properties that the learned indexes should hold.

Contents

1 Introduction

Over the years, indexes have been widely used in databases to improve the speed of data retrieval. In the past decades, the database indexes generally fall into the hand-engineered data structures, such as B-Tree, KD-Tree, etc. These indexes have played a crucial role in databases and have been used widely in modern data management systems (DBMS) such as PostgreSQL. Despite their huge success, a shortcoming of these data structures is the lack of consideration of how the database records distributed. We use an example to demonstrate how distributions can affect the efficiency of database indexes.

Example 1.1 For example, if the dataset contains integers from 1 to 1 million, then the keys can be used directly as offsets. With the keys used as offsets, the value with a given key can be retrieved in $\mathcal{O}(1)$ time complexity while B-Tree requires $\mathcal{O}(\log n)$ time complexity for the same query. From the perspective of space complexity, we do not need any extra overhead by using the key as an offset directly, while the B-Tree needs extra $\mathcal{O}(n)$ space complexity to save the tree.

From the above example, we found that there are two promising advantages of leveraging the distribution of the data:

1. It may be faster when performing queries, especially when the number of entries in the database are rather huge.

2. It may take less memory space, as we only need to save the model with constant size.

Nowadays, to learn the distribution and apply it to database indexes, Tim Kraska et al. proposed learned indexes [1], [2], where machine learning techniques are applied to automatically learn the distribution of the database entries and build the data-driven indexes. In this project, we implemented both hand-engineered indexes and the learned index.

1. **Introduction**. In this chapter, we illustrate the organisation of this report and introduce the general information about database indexes.

2. **Implementation**. In this chapter, we present our implementation of the B-Tree for 1-D data and KD-Tree for 2-D data. We also present our implementation of the learned index for 1-D data [1] and 2-D data [2]. This includes a baseline learned index, a recursive model and all components of LISA framework.

3. **Evaluation**. In this chapter, we report evaluation of our implementation.

4. **Insights and Findings**. We demonstrate our findings in this chapter. Besides, we also discuss the advantages and disadvantages of different indexes.

5. **Conclusions**.

1.1 Notations

In this report, we will use the following notations:

Sets and Spaces	
$\mathbb{R}$	The set of real numbers
$\mathbb{R}^d$	The set of d dimensional real space
Random Variables	
$\boldsymbol{X}$	A vector or matrix
x	A single value in **X**
(x, y)	A tuple that contains two values
Hyper-Parameters	
N	A pre-set hyper parameter
Functions	
$\mathcal{LR}$	Linear Regression Function
$\mathcal{P}$	Polynomial Function
$\mathcal{M}$	Mapping Function $\mathbb{R}^n \to \mathbb{R}$
$\mathcal{O}$	Big-O notation of complexity
$\mathcal{SP}$	Shard Prediction Function
$\mathcal{Q}(\boldsymbol{l}, \boldsymbol{u})$	Range Query with the bounds $(\boldsymbol{l}, \boldsymbol{u})$
$\mathcal{K}(x, y, k)$	KNN Query for kth nearest neighbours around (x, y)

1.2 Terminologies

In the following chapters, we will use the following terminologies

Index model is a function that maps the index of a row of data into the location (e.g. page index) of the data. For example, in one-dimensional case, the index models include B-Tree, Linear Regression models, etc.

Key is a special attribute in the database that could identify a record. In our work, the key could be a scalar in one-dimensional case, or a (x, y) pair in two-dimensional case.

Order of a tree is the maximum number of children that a node can have.

Internal node is any node of a tree that has child nodes and is not a root node.

Leaf node is any node that does not have child nodes.

Level of a node is defined as the number of edges between this node and the root node.

1.3 Assumptions

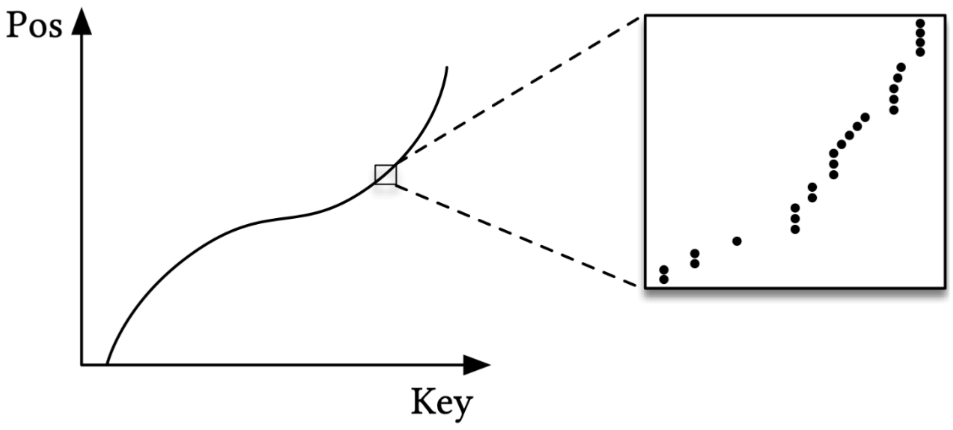


Figure 1.1: The illustration of indexes as CDFs, originally from [1]

Formally, we define the index of each record as x and the corresponding location as y and we represent the whole data as (X, Y) pairs with the total number of pairs defined as N. We could then normalise the Y into $\tilde{Y} \in [0, 1]$ so that the $\tilde{y}$ represents the portion of the y among the whole Y. With these definitions, we can then define a function $F : X \to \tilde{Y}$ that maps the index into the portion of the y. We have $y = F(x) * N$. As the output of this function can be considered as the probability of $X \leq x$, we can regard this function $F(x)$ as the cumulative distribution function (CDF) of X, i.e. $F(x) = \mathbb{P}(X \leq x)$. Now that N is determined by the length of data, we only need to learn such CDF and we called the learned CDF function as **learned index model**.

In Fig. 1.1, we illustrate the relationship between the key and its position. The raw keys and their positions are illustrated in the zoomed-in view and the zoomed-out view presents a shape of the relation. In this figure, we present why the position can be regarded as a CDF: the position of a key is always the position of previous key plus 1, i.e. the position describes how many keys are there before a certain key x. If we divide it by the total number of keys, we will have the result as the possibility of how many keys are smaller than the certain key x, i.e. $\mathbb{P}(X \leq x)$. The result is therefore the CDF of X.

Example 1.2 From the perspective of the distribution of keys, our previous example can be rephrased as following. Our data are key value pairs (X, Y) (we call the pairs as data points) with a linear relation, i.e. $y = x, \forall y \in Y$. We are looking for a function F such that $y = x = F(x) * N$, and hence we end up with $F(x) = \frac{1}{N} * x$. If we use this linear function $F(x)$ as the index model, then we could locate the data within $\mathcal{O}(1)$ time complexity and we only need to store the total number of data points as the only parameter. Compared with B-Tree whose query complexity is $\mathcal{O}(\log n)$, the potential of using learned index to handle huge amount of data is enormous.

In order to ensure the learned index model to be the desired CDF, we need to make the following assumptions:

1. All data points are stored statically. Hence we do not take insertion and deletion into consideration. If there is some insertion or deletion, then the total size of the data records, N, will be different. Therefore, if insertion or deletion are involved, we cannot calculate the position as we show above.

2. All data points are sorted according to theirs keys $\boldsymbol{X}$. Only when the data records are sorted according to the keys, we can regard the index model as CDF, i.e. $F(x) = \mathbb{P}(X \leq x)$.

3. For simplicity, we assume that our data points are stored in a continuous memory space. In other words, the indices of pages in this project is continuous integers and all the data records are loaded into memory.

2 Implementation

Summary In this chapter, we describe the implementation details of classic tree-based indexes and learned indexes.

1. In the Section 2.1, we present how to construct the B-Tree, Baseline model and Recursive Model Index (RMI) for one-dimensional data.
2. In the Section 2.2, we present how to construct the KD-Tree, LISA baseline and LISA model for two-dimensional data.
3. In the Section 2.3, we describe how to use these indexes to perform different queries. For the one-dimensional data, we show how to perform point query with B-Tree, Baseline model and RMI. For the two-dimensional data, we show how to perform point query, range query and KNN query with KD-Tree, LISA baseline and LISA model.

2.1 One Dimensional Data

2.1.1 B-Tree

B-tree and its variants have been widely used as indexes in databases. B-trees can be considered as a generalisation of binary search tree: In binary search tree, there is only one key and two children at most in the internal node. B-tree extends the nodes such that each node can contain several keys and children. The keys in a node serve as dividing points and separate the range of keys. With this structure, we make a multi-way decision based on comparisons with the keys stored at the node x.

In this section, we introduce the construction process of B-trees and then analyse its properties.

Attributes and Properties

Each node `x` in a B-tree has the following attributes:

- `x.n`: the number of keys currently stored in the node x.
- `x.keys`: the stored keys of this node.
- `x.leaf`: a bool value that determines if current node is a leaf node.

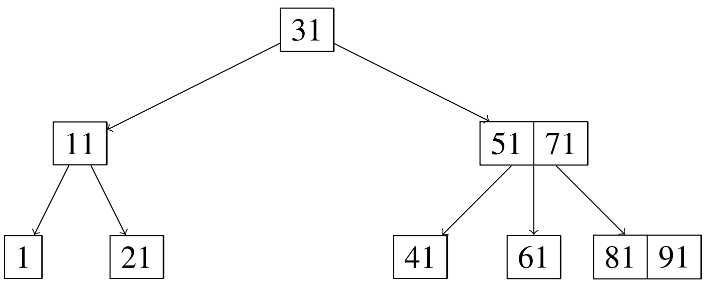


Figure 2.1: An example of B-tree with the minimum degree $t = 2$.

- `x.children`: a list of its children. If `x` is a leaf node who has no children at all, then the list will be empty. We assume the children are $\texttt{x.}c_1, \cdots, \texttt{x.}c_{x.n+1}$, i.e. there will be $\texttt{x.n} + 1$ children at most.

With these attributes, a B-tree has the following properties:

- The number of children of a node is always 1 bigger than the number of keys in a node.
- Nodes in this tree have lower and upper bounds on the number of keys they can contain. These two bounds can be expressed in terms of a fixed integer t, which we call the **minimum degree** of this tree.
  1. Each node, other than the root node, must contain at least $t - 1$ keys. The root of the tree must have at least one key if the tree is not empty.
  2. Each node can contain at most $2t - 1$ keys. A node is called **full** if it contains exactly $2t - 1$ keys.
- Inside each node, the keys are sorted in the non-decreasing order, so that we have $\texttt{x.keys}_1 \leq \texttt{x.keys}_2 \leq \cdots \leq \texttt{x.keys}_{\texttt{x.n}}$.
- The keys $\texttt{x.key}_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with a root $\texttt{x.c}_i$, then we have $k_1 \leq \texttt{x.keys}_1 \leq k_2 \leq \texttt{x.keys}_2 \leq \cdots \leq \texttt{x.keys}_n \leq k_{\texttt{x.n}+1}$.

In Fig. 2.1, we demonstrate an example B-tree whose minimum degree is 2. In the following section, we will illustrate how to construct and insert keys into a B-tree.

Insertion in a B-tree

With a B-tree, we cannot simply create a new leaf node and insert the new key as we do with a binary search tree, because the resulting tree will fail to be a valid B-tree. Instead, we need to insert the new key into an existing leaf node. If the node is not full, we can safely insert the new key. Otherwise, we will need to split the node around the median of its keys into two new nodes and promote the median key into its parent. In this process, we need to split the parent if its parent is also full.

In the insertion, we travel down the tree and search for the position where the key should be inserted. During the traverse, we split each full node along the way. By doing so, whenever we want to split a full node, we are assured that its parent is not full. The overall algorithm is shown in Algo. 1, which contains methods `splitChild` and `InsertNonFull` as described in Algo. 2 and Algo. 3 respectively.

Algorithm 1: Insert

```
input: T: The tree with the root T.root; k: The key to be inserted
Result: T: The tree with the inserted key k
r=T.root
if T.n==2t-1 then
    s = NewNode()
    T.root = s
    s.leaf = False
    s.n = 0
    s.$c_1$ = r
    SplitChild(s, 1)
    InsertNonFull(s, k)
else
    InsertNonFull(r, k)
```

In the Algo. 1, we first check if the root node r is full. If it is full, then the root splits and a new node s becomes the root. Then we insert the key k into the tree rooted at the non-full root node, i.e. s or r.

In the Algo. 2, the node y originally has $2t$ children (i.e. $2t - 1$ keys) and is full. We take the following steps to split it:

1. We first (from line 1 to line 11) create a new node z and give it the largest $t - 1$ keys and the corresponding t children of y.
2. Then we adjust the count of keys for y on line 12: after the split, y will have $t - 1$ keys.
3. After that, from line 13 to line 21, we insert z as a child of x, move the median key from y up to x, and adjust the key count in x.

Algorithm 2: SplitChild

```
input: x: The node whose children are being split; i: The index of x's child who is
       full originally
Result: x: The parent node whose children are not full
1  z = NewNode()
2  y = x.c_i
3  z.leaf = y.leaf
4  z.n = t-1
5  for $j \leftarrow 1$ to $t-1$ do
6  |   z.keys_j = y.keys_{j+t}
7  end
8  if not y.leaf then
9  |   for $j \leftarrow 1$to $t$ do
10 |   |   z.c_j = y.c_{j+t}
11 |   end
12 y.n = t-1
13 for $j \leftarrow x.n$ to $i+1$ do
14 |   x.c_{j+1} = x.c_j
15 end
16 x.c_{i+1} = z
17 for $j \leftarrow x.n$ to $i$ do
18 |   x.keys_{j+1}=x.keys_j
19 end
20 x.key_i = y.key_t
21 x.n = x.n+1
```

The Algo. 3 works as follows:

1. From line 3 to line 6, We first check if x is a leaf. If it is a leaf, then we insert the key k into x.

2. If x is not a leaf, then we must insert k into the appropriate leaf node in the subtree rooted at internal node x. From line 8 to line 11, we traverse the subtree rooted at x and determine the child of x to which the recursion descends. Then we check on line 12 if the child where the recursion descends is a full node.

3. If the child is a full node, we then split the child on line 13 into two non-full children. We then determine from line 14 to line 15 which of the two children is the appropriate node to insert.

4. At the last, on line 16 we look into the ith children of x and recursively insert the key k into it.

Algorithm 3: InsertNonFull

```
input: x: The node to be inserted; k: The key to be inserted
Result: x: The node with the inserted key k
i=x.n
if x.leaf then
    while $i \geq 1$ and $k < x.keys_i$ do
        x.key$_{i+1}$=k
        x.n = x.n+1
    end
else
    while $i \geq 1$ and $k < x.keys_i$ do
        i=i-1
    end
    i=i+1
    if x.c$_i$.n==2t-1 then
        SplitChild(x,i)
        if k>x.key$_i$ then
            i=i+1
    InsertNonFull(x.c$_i$, k)
```

2.1.2 Baseline Learned Index

Overview

The B-Tree can be regarded as a function $\mathcal{F}$ that maps the key x into its corresponding page index y. It is known to us that the pages are allocated in a way that the every S entries are allocated in a page where S is a pre-defined parameter. For example, if we set S to be 10 items per page, then the first page will contain the first 10 keys and their corresponding values. Similarly, the second 10 keys and their corresponding values will be allocated to the second page.

If we know the CDF of X as $F(X \leq x)$ and the total number of entries N, then the position of x can be estimated as $p = F(x) * N$ and the page index where it should be allocated to is given by

$$y = \lfloor \frac{p}{S} \rfloor = \lfloor \frac{F(x) * N}{S} \rfloor$$

Example 2.1 For example, if the keys are uniformly distributed from 0 to 1000, i.e. the CDF of X is defined as $F(X \leq x) = \frac{x}{1000}$ and we set $S = 10, N = 1001$. Then for any key x, we immediately know it will be allocated into $y = \lfloor \frac{1000}{10} * \frac{x}{1000} \rfloor = \lfloor \frac{x}{10} \rfloor$. Assume that we have a key 698, then we can calculate $y = \lfloor \frac{698}{10} \rfloor = 69$. By doing so, the page index is calculated in constant time and space.

In this example, we see that the distribution of X is essential and our goal of learned

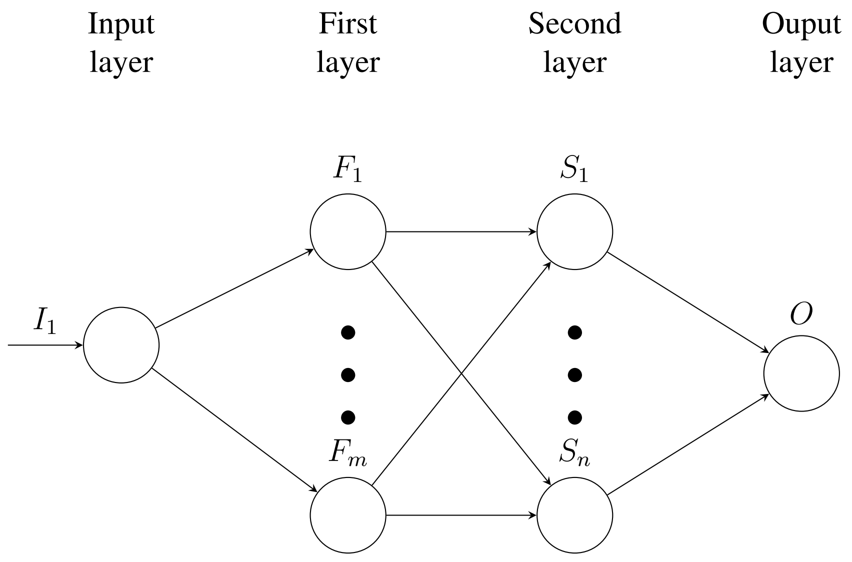


Figure 2.2: The architecture of the fully connected neural network used as baseline learned index. In this neural network, we use only 2 fully connected layers. The input of this neural network is only one neuron such that it represents the given query key. The output of this neural network is limited to 1 neuron such that it represents the predicted proportional position of the key-value pair.

index in one-dimensional data is to learn such distribution. To do so, we apply two different techniques as the baseline, the polynomial regression and fully connected neural network.

To train such a learned index, we first manually generate the X with respect to a certain distribution. We then save the generated X into a dense array with the length N. Then we use the proportional index, i.e. the index of each x divided by N as the expected output y.

Fully Connected Neural Network

After generating the training dataset X and its corresponding Y, we build a fully connected neural network as the baseline learned index. The architecture of the fully connected neural network is illustrated in Figure 2.2.

We apply the Rectified Linear Unit (ReLU) activation function at the end of F_i and S_i. Formally, assume the output of F_i is $\boldsymbol{a}$, then we define the output of $\boldsymbol{ReLU}(F_i)$ as $y = \max(\boldsymbol{a}, 0)$ where max returns the larger value between each entry of $\boldsymbol{a}$ and 0. Then we train this fully connected neural network with standard stochastic gradient descent (SGD), and we set the learning rate to be $\alpha = 0.001$. We use the mean square error (MSE) $\ell = \frac{1}{n}\sum(y - \hat{y})^2$ as the loss function.

Formally, we can induce the output of this fully connected neural network as following:

1. In the input layer, we have the input as a scalar value x.

2. The first fully connected layer has m nodes, and the output is defined as $\boldsymbol{y_1} = \boldsymbol{w_1}x + \boldsymbol{b_1}$ where $\boldsymbol{w_1}$ and $\boldsymbol{b_1}$ is a $m \times 1$ matrix. Hence, the output of the first fully connected layer is a $m \times 1$ matrix. Then we apply the ReLU activation function to $\boldsymbol{y_1}$ and we get $\boldsymbol{z_1} = \max(\boldsymbol{y_1}, 0)$.

3. The second fully connected layer has n nodes, and the output is defined as $\boldsymbol{y_2} = \boldsymbol{w_2}\boldsymbol{z_1} + \boldsymbol{b_2}$. Similarly, after the ReLU operation, we get $\boldsymbol{z_2} = \max(\boldsymbol{y_2}, 0)$.

4. For the output layer, in order to get a scalar as output, we apply a n node fully connected layer here. The final output is defined as $\hat{y} = \boldsymbol{w_3}\boldsymbol{z_2} + \boldsymbol{b_3}$ where $\boldsymbol{w_3}$ is a $1 \times n$ matrix.

In summary, the output of the fully connected neural network can be calculated as

$$\hat{y} = \boldsymbol{w_3}\max(\boldsymbol{w_2}\max(\boldsymbol{w_1}x + \boldsymbol{b_1}, 0) + \boldsymbol{b_2}, 0) + \boldsymbol{b_3} \tag{2.1}$$

In the above fully connected neural network, there are 6 parameters to optimise: $\boldsymbol{w_1}, \boldsymbol{w_2}, \boldsymbol{w_3}$ and $\boldsymbol{b_1}, \boldsymbol{b_2}, \boldsymbol{b_3}$ and we apply the gradient descent and back propagation to optimise them. Formally, the steps are illustrated below:

1. **Initialisation**. For $\boldsymbol{w_i}$ and $\boldsymbol{b_i}$ of the shape $m \times n$, we randomly initialise the values of each entry using a uniform distribution $U(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$.

2. **Forward Pass**. With the initialised $\boldsymbol{w_i}$ and $\boldsymbol{b_i}$, we calculate the output as formulated be the equation 2.1. We then calculate the error as $\ell = \frac{1}{n}\sum(y - \hat{y})^2$.

3. **Backward Pass**. After getting the error, we start from the last layer to perform the backward propagation operation. Formally, we do the following operations:

   a) We first calculate the partial derivatives: $\frac{\partial \ell}{\partial \boldsymbol{w_3}} = \boldsymbol{z_2}^T$, $\frac{\partial \ell}{\partial \boldsymbol{b_3}} = 1$ and $\nabla_3 = \frac{\partial \ell}{\partial \boldsymbol{z_2}} = \boldsymbol{w_3}^T$. Then we can update $\boldsymbol{w_3}$ and $\boldsymbol{b_3}$ as $\boldsymbol{w_3}^{\text{new}} = \boldsymbol{w_3} - \alpha * \frac{\partial \ell}{\partial \boldsymbol{w_3}}$ and $\boldsymbol{b_3}^{\text{new}} = \boldsymbol{b_3} - \alpha * \frac{\partial \ell}{\partial \boldsymbol{b_3}}$.

   b) Then we pass the ∇_3 to previous layer, and calculate the partial derivatives as $\frac{\partial \ell}{\partial \boldsymbol{w_2}} = \boldsymbol{z_2}^T\nabla_3$, $\frac{\partial \ell}{\partial \boldsymbol{b_2}} = \nabla_3$ and $\nabla_2 = \frac{\partial \ell}{\partial \boldsymbol{z_1}} = \nabla_3\boldsymbol{w_2}^T$. Then we update $\boldsymbol{w_2}$ and $\boldsymbol{b_2}$.

   c) After that, we pass the ∇_2 to the first layer, and calculate the partial derivatives as $\frac{\partial \ell}{\partial \boldsymbol{w_1}} = x^T\nabla_2$, $\frac{\partial \ell}{\partial \boldsymbol{b_1}} = \nabla_2$. Then we update $\boldsymbol{w_1}$ and $\boldsymbol{b_1}$.

4. **Loop between 2 and 3**. We perform the forward pass and the backward several times until the loss is acceptable or a maximum number of loops reached.

We will discuss more findings and insights about the baseline model in the *Chapter 4*.

2.1.3 Recursive Model Index

$\boxed{\textbf{Motivation}}$ With a single well-trained baseline model, we can reduce the errors from several millions to thousands. However, as the model is trained to minimise the overall error, it may behave badly in some intervals. If we illustrate the predicted output from a baseline model and the ground truth, as in Fig. 2.3, we will find that even though the baseline model has achieved a rather low error, it does not fit the data almost everywhere. In order to solve this problem, the recursive model index was proposed [1]. The idea is to split the large dataset into smaller pieces and assign each piece an index model. By doing so, each model is only responsible for a small range of keys.

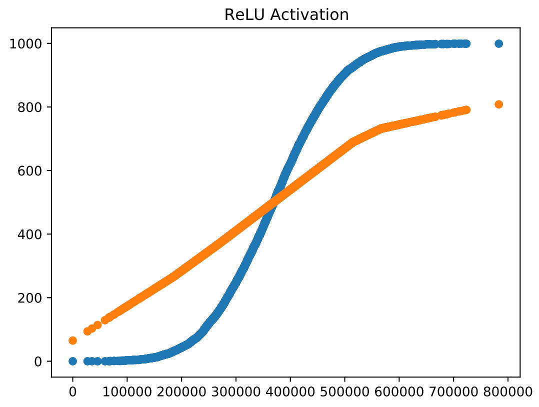


Figure 2.3: The predicted output and the ground truth from the baseline model

Ideally, in each smaller range, the keys are distributed in a way that is easier to be learned by our index models, such as polynomial model, fully connected model or even a B-Tree model.

As shown in Fig. 2.4, A recursive model is a tree structure, which contains a root model that receives the full dataset for training. Then the root model will split the dataset into several parts. Each sub-model will then receive one part of the full dataset. Then we train the sub-models one by one with the partial training dataset.

Example 2.2 For example, in the Fig. 2.4, the full dataset will be split into three parts and each sub-model receives one part. To train this recursive model, we first train the root model with the whole dataset. Then the root model will split the dataset into 3 parts according to the predicted value of each data point in the dataset. Then each sub-model will receive one part and we train the sub-model accordingly.

Properties

Similar to a tree, we define the following terms in a recursive model:

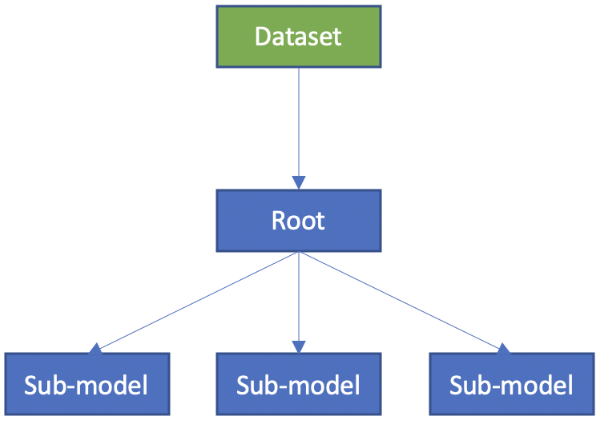


Figure 2.4: An example of recursive model index with one root model and three submodels.

1. **Node Model**. Every node is responsible for making decisions with given input data. In one dimensional case, it can be regarded as a function $f : \mathbb{R} \rightarrow \mathbb{R}, x \rightarrow y$ where x is the input index and y is the corresponding page block. In principle, each node can be implemented as any machine learning model, from linear regression to neural network, or a traditional tree-based model, such as B-Tree. In the Fig. 2.4, the root model and three submodels are all node models.

2. **Internal Node Model**. Internal nodes are all nodes except the leaf nodes. Every internal node receives a certain part of training data from the full dataset, and train a model on it.

3. **Submodel** Similar to the subtrees in tree structures, each internal node (i.e. except the leaf node) has several submodels. For example, in Fig. 2.4, the root model has three submodels.

In the following sections, we will use the notations defined below:

1. $N_M^{(i)}$ is the number of models in the ith stage.

Training

In order to construct a recursive model, we need to have several parameters listed below:

1. The training dataset, notated as (X, Y) with entries notated as (x, y).

2. The number of stages is notated as N_S. It is an integer variable.

3. The number of models at each stage, notated as N_M. It is a list of integer variable. $N_M^{(i)}$ represents the number of models in the ith stage.

The training process of recursive model is an up-bottom process. There will be only one root model that receives the whole training data. After the root model is trained, we iterate over all the training data and predict the page by the root model. After the iteration, we get a new set of pairs (X, Y_0). Then we map $\forall y_0 \in Y_0$ into the selected model index in next stage by `next` $= y_0 * N_M^{(i+1)}$`/max(Y)`.

Algorithm 4: Training of Recursive Model Index

```
input: $N_S$:  A scalar representing the number of stages;
       $N_M$:  An array representing the number of models at
each stage;
       x; y
trainset=[[(x,y)]]
stage← 0
model← 0
models← [[]]
while stage <$N_S$ do
    while model <$N_M$[stage] do
        model.train(trainset[stage][model])
        models[stage].append(model)
    end
    if stage<$N_S$-1 then
        for $i \leftarrow 0$ to $len(x)$ do
            next_model = 0
            for $j \leftarrow 0$ to stage do
                output = models[stage][next_model].predict(x)
                next = output * $N_M$[j+1]/max_y
            end
            trainset[stage+1][next].add((x[i],y[i]))
        end
    stage=stage+1
end
```

In the algorithm 4, we perform the following operations:

1. From the line 1 to 4, we prepare the training dataset to be a two-dimensional array `trainset` such that `trainset[i][j]` represents the training dataset use by the jth model at the ith stage. Hence, we first initialise the `trainset[0][0]` to be the full dataset. Besides, we create an empty two dimensional list `models` such that `models[i][j]` represents the jth model at the ith stage. We also let the current model and stage to be 0 as initialisation.

2. From the line 5 to 20, we iterate from 0 to N_S, i.e. the number of stages. At each stage, we perform the following operations:

    a) From the line 6 to 9, we iterate from 0 to N_M at this stage. During the iteration, we train all the models with the `trainset[i][j]`, i.e. the prepared dataset for the given model index at this stage. After that, we add the trained model into the model list at line 8.

b) From the line 10, we prepare the training dataset. If `stage<`N_S`-1`, then the current stage is not the last stage, i.e. we need to prepare the dataset for the next stage. Hence, we iterate over all the training data from line 11 to 21.

c) From the line 13 to 16, we iterate over all stages that are previous to the current stage (including the current stage). For each stage, we compute the prediction from the model and map the prediction into the index of model at the next level. Finally, we will get the index of the model that are supposed to handle the input data point x.

d) On line 17, we add the data point `x[i], y[i]` into the corresponding trainset.

Other types of models inside RMI

One of the promising properties of recursive model is the support of multiple types of models. If we have some prior knowledge of the data distribution, this property might help us train the model faster and better.

Example 2.3 For example, if we know the data is linear in most local regions, then we could use linear regression as internal models.

In addition, if we witnessed a huge error with RMI with learned models, we can replace the internal models to be B-Tree models as they are more accurate and do not care about the data distributions.

In thei project, we support the following types of models:

Type of models	Formulas
Linear Regression	$wx + b$
Quadratic Regression	$ax^2 + bx + c$
B-Tree	N/A
Fully Connected Neural Network	N/A

As linear regression and quadratic regression are special cases of polynomial models, they are implemented in a same manner in this project. Here we demonstrate how to fit a polynomial model. The polynomial regression model with degree m can be formalised as

$$\hat{y_i} = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m$$

and it can be expressed in a matrix form as below

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ \vdots & & & & \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix}$$

which can be written as $Y = \boldsymbol{X\beta}$.

Proof 2.1 Our goal is to find β such that the sum of squared error, i.e.

$$\mathrm{S}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (\hat{y} - y)^2$$

is minimal. This optimisation problem can be resolved by ordinary least square estimation as shown below.

First we have the error as

$$\begin{aligned} \mathrm{S}(\boldsymbol{\beta}) = ||\boldsymbol{y} - \boldsymbol{X\beta}|| &= (\boldsymbol{y} - \boldsymbol{X\beta})^T(\boldsymbol{y} - \boldsymbol{X\beta}) \\ &= \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{X\beta} + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta} \end{aligned} \tag{2.2}$$

Here we know that $(\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{y})^T = \boldsymbol{y}^T\boldsymbol{X\beta}$ is a 1×1 matrix, i.e. a scalar. Hence it is equal to its own transpose. As a result we could simplify the error as

$$\mathrm{S}(\boldsymbol{\beta}) = \boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{X\beta} \tag{2.3}$$

In order to find the minimum of $S(\boldsymbol{\beta})$, we differentiate it with respect to $\boldsymbol{\beta}$ as

$$\nabla_{\boldsymbol{\beta}} S = -2\boldsymbol{X}^T\boldsymbol{y} + 2(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{\beta} \tag{2.4}$$

By let it to be zero, we end up with

$$\begin{aligned} &-\boldsymbol{X}^T\boldsymbol{y} + (\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{\beta} = 0 \\ &\implies \boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \end{aligned} \tag{2.5}$$

■

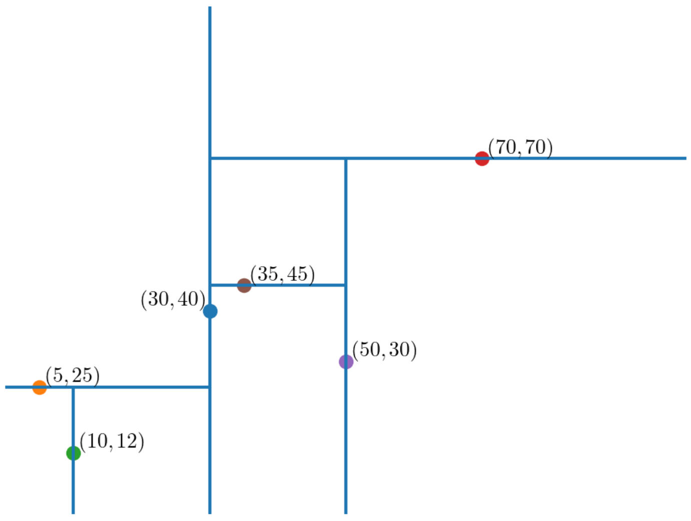
With the proof above, we showed that we can train the polynomial model with closed form.

$\boxed{\textbf{Summary}}$ In this section we showed the theoretical induction of the baseline model and the recursive model index. In the recursive model, we implemented four different types of models: B-Tree, fully connected neural networks, linear regression and quadratic regression.

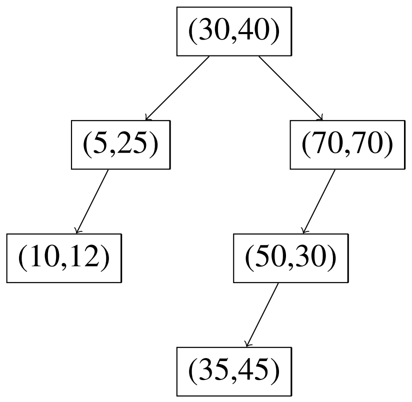
2.2 Two Dimensional Data

2.2.1 KD-Tree

KD-tree is a space partitioning structure that can be used to organise data points in k dimensional space. In this project, we limit the dimension k to be 2. We implement the KD-tree as a binary tree in which every node is a 2-dimensional point. Every non-leaf node is representing a splitting line that divides the space into two parts. Then every points to the left (or down) of this line are represented by the left subtree and points to the right (or up) of this line are represented by the right subtree. In Fig. 2.5 we illustrate an example of KD-tree.



(a) The 2D space and the partition of KD-tree



(b) The tree structure of the KD-tree

Figure 2.5: An example of KD-Tree

Insertion of KD-tree

Similar to a binary search tree, we need to traverse the tree when we need to insert a point to the KD-tree. The only difference is that we need to switch the axes when inserting into a KD-tree. For example, since the dimension is 2 in our case, we compare the x-coordinate at the root level. Then in the root's direct children, we compare the y-coordinate at that level. Formally, the insertion algorithm is expressed as in Algo. 5.

Algorithm 5: KD-tree Insertion

```
input: t: The node to be inserted; k: The key to be inserted; cd: Current dimension
Result: t: The node with the inserted key k
1 DIM=2;
2 if t==NULL then
3     t = NewNode(k)
4 else if x[cd]<t.data[cd] then
5     t.left=insert(x, t.left, (cd+1) % DIM )
6 else
7     t.right=insert(x, t.right, (cd+1)% DIM)
8 return N
```

To insert a key `k` into the KD-tree with `T` as its node, we only need to apply this function with the root node as `insert(T, k, 0)`.

In the Algo. 5, the insertion of a key is performed in the following steps:

1. On line 1, we specify the dimension to be 2.
2. Then we first check if the node is `NULL`. If it is `NULL`, which means we should have a new leaf node, then we create a new node and give it our key.

3. Otherwise, from line 4 to 7, we check the data at current dimension. If it is smaller than the data in the node `t` at current dimension, then we should insert into the left subtree. Otherwise, we should insert into the right subtree.

4. When we moves down to the left or right subtree, we switch the current dimension by calculating `(cd+1)% DIM`.

Example 2.4 In Fig. 2.5, we present an example of KD-tree. In this example, we will illustrate how it is constructed. Assume our data points is

$$[(30, 40), (5, 25), (10, 12), (70, 70), (50, 30), (35, 45)]$$

The construction of the KD-tree follows the steps below:

1. We start with $(30, 40)$ by creating a new node and give it the data point, which results in the root node in the figure.

2. After that we insert $(5, 25)$, since `x[0]<t.data[0]`, we insert this node as the left subtree to the root.

3. Then we insert $(10, 12)$. First we compare the x-coordinate at the root level. As $10 < 30$, we go to the left subtree. Then we compare at the root's children level and compare the y-coordinate. As $12 < 25$, we go to the left subtree and create a new node there.

4. Similarly, we insert other keys one-by-one.

2.2.2 Learned 2D Indexes Overview

Motivated by the performance benefits of learned indices for one-dimensional data, this section explores the application of learned index for spatial data. In this section, we present the implementation of LISA [2], which extends the learned indexes into two dimensional data.

Motivation

For the one-dimensional data, we can learn the CDF by using a recursive model as shown in the section 2.1.3. However, when the data is two-dimensional, the learned CDF (the marginal CDF) through the recursive model cannot be applied directly to predict the position of the key. Formally, we could learn the marginal CDF for each dimension by using the recursive model, i.e. $F(X)$ and $F(Y)$. However, to predict the position of a 2-dimensional key, we need the joint CDF $F(X, Y)$, which cannot be induced from the marginal CDFs. We show an example as below to illustrate this limitation.

Example 2.5 Assume that X and Y are distributed as shown in Fig. 2.6. In this example, we have three points A, B and C. For simplicity, we assume the page size is 1. Hence, the point A should be assigned into the first page and we have $F(x \leq A) = \frac{1}{3}$. With learned indexes in one-dimensional, then there comes the problem below:

1. There will be duplicate keys. In this example, if we only consider the X axis, we will get an array $[0.7, 0.7, 1.5]$ which contains duplicate keys.
2. If we remove the duplicate keys, then $F(x \leq A) = \frac{1}{2}$, which is not what we expect.
3. If we do not remove the duplicate keys, then $F(x \leq A) = F(x \leq B)$, which is still not we expect.

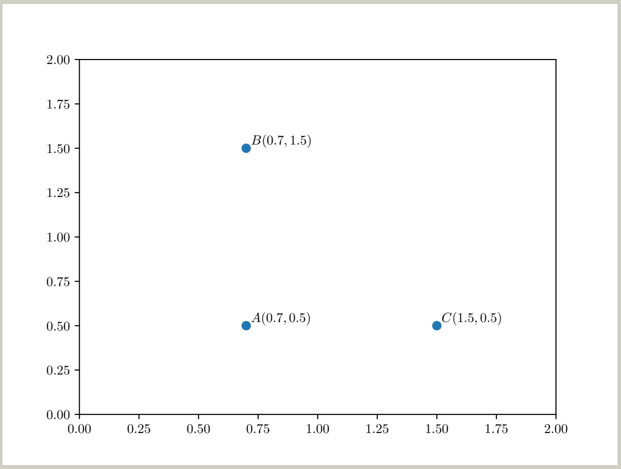


Figure 2.6: An example demonstrating the limitations of one-dimensional learned index in two-dimensional data. In this graph we have $F(x \leq A) = \frac{1}{3}$ but with learned index in one-dimensional, we cannot learn such joint CDF.

From the above discussion, we conclude that we cannot apply the one dimensional learned indexes directly, as any solutions to the duplicate keys will result in unexpected behaviour.

In order to extend the idea of learned indexes in one dimensional data but tackle the limitations that we face, LISA maps the keys in two dimensional space into a one dimensional sorted array and then apply piecewise linear function to learn the CDF.

In the following implementation of two dimensional learned indexes, we will use the following definitions.

1. **Key**. A key k is a unique identifier for a data record with $k = (x_0, x_1) \in \mathbb{R}^2$.

2. **Cell**. A grid cell is a rectangle whose lower and upper corners are points (l_0, l_1) and (u_0, u_1). Formally, we present cell as $[l_0, u_0) \times [l_1, u_1)$. Every key in the two dimensional space can be located in one cell. We use $a \in C_i$ to denote a key a in the ith cell.

3. **Mapping Function**. A mapping function $\mathcal{M}$ is a function on the domain $\mathbb{R}^2$ to the non-negative range in $\mathbb{R}$, i.e $\mathcal{M} : [0, X_0] \times [0, X_1] \to [0, +\infty)$.

   The mapping function should not mess up the original order of the cells of keys. Formally, assume we have two cells i, j and $i < j$, then $\mathcal{M}(a) < \mathcal{M}(b), \forall a \in C_i, \forall b \in C_j$.

2.2.3 Baseline Method

By using a mapping function, we can extend the learned index method on spatial data in the following steps:

1. First of all, we use the mapping function to map all keys and get a one dimensional array.
2. Then we sort all keys according to their mapped values and divide the mapped values into small intervals. We divide them in a way that each interval (except the last one) contains the same number of the keys. By doing so, we ensured that if the mapped value of a key (x, y) is larger than those of the keys in the first i intervals, then this key will be located in the $(i + 1)$th interval.

Training of Baseline Method

The training dataset for the baseline model can be notated as $(\boldsymbol{X}, Y)$ with entries notated as $(\boldsymbol{x}, y)$. $\boldsymbol{X}$ represents the two dimensional key coordinates, and Y represents the corresponding data item.

Algorithm 6: Training Algorithm for Lisa Baseline Method

```
input  : $N$:number_of_cells;
         trainset:[$(x,y); x \in \mathbb{R}^2; y \in \mathbb{R}$]
Output: cell:Array containing cells' metadata
1  for $i \leftarrow 0$ to $len(x)$ do
2      $x$[$i$].mapped_value = $x$[$i$][0]+$x$[$i$][1]
3  end
4  $K$ = len($x$)/$N$ // keys per cell
5  $x$ = $x$[argsort($x$.mapped_value)] //sort x based on mapped values
6  for $i \leftarrow 0$ to $N$ do
7      cell[$i$].lower = $x$[$i$* $K$].mapped_value
8      cell[$i$].upper = $x$[($i$+1)*$K$].mapped_value
9  end
10 return cell
```

In the Algo. 6, training of LISA baseline model is described in the following steps:

1. N, which represents the number of cells into which the key's mapped value space will be divided.
2. In lines 1 to 3, we calculate the mapped value of each item in the training set.
3. On line 4, we calculate the number of keys per cell.
4. On line 5, Sort train set according to keys' mapped values.
5. In lines 6 to 9, we divide the keys into equal sized cells. Per cell we need to store meta data for two keys, corresponding to first and last key in the cell

Prediction of Baseline Method

For prediction, we find the cell corresponding to mapped value of the query point using binary search, scan this cell sequentially and compare the values of keys in the cell against the query point, until a match is found.

Algorithm 7: Prediction Algorithm for LISA Baseline Model

```
input  : x_test:query_point;
         cell:cell_metadata_array;
         trainset:[$(x,y); x \in \mathbb{R}^2; y \in \mathbb{R}$]
Output: x_test.value:query_point_value
cell_found = False
x_test.mapped_value = x_test[0]+x_test[1]
for $i \leftarrow 0$ to len(cell) do
    if x_test.mapped_value$\in$ [cell[i].lower, cell[i].upper) then
        cell_found = True
        break
    end
end
if cell_found==True then
    $K$ = cell.keys_per_page
    cell_offset = $K \star i$
    for $i \leftarrow cell\_offset$ to $K$+cell_offset do
        if (x_test[0] == x[i][0]) and (x_test[1] == x[i][1]) then
            return $x$[$i$].value
        end
    end
end
return -1
```

Limitations of Baseline Method

The baseline method can be used to perform point query and range query, but suffers from a severe problem. We demonstrate this problem with a range query in the following. In the two dimensional case, a range query is defined as searching for all data points that fall into a rectangle range in the two dimensional space.

For a range query, represented by the query rectangle $qr = [l_0, u_0) \times [l_1, u_1)$, we only need to predict the indices of (l_0, l_1) and (u_0, u_1) namely i_1 and i_2 respectively. Then we scan the keys in $i_2 - i_1 + 1$ cells, and find those keys that fall in the query rectangle qr.

Example 2.6 An example of range query with LISA baseline method is illustrated in Fig. 2.7. In this figure, the key space are divided into 3 cells with equal number of keys. With the query range, we find the Cell 2 contains the query rectangle. After that, we need to iterate over all the points that fall into the Cell 2 (all the red points in this figure).

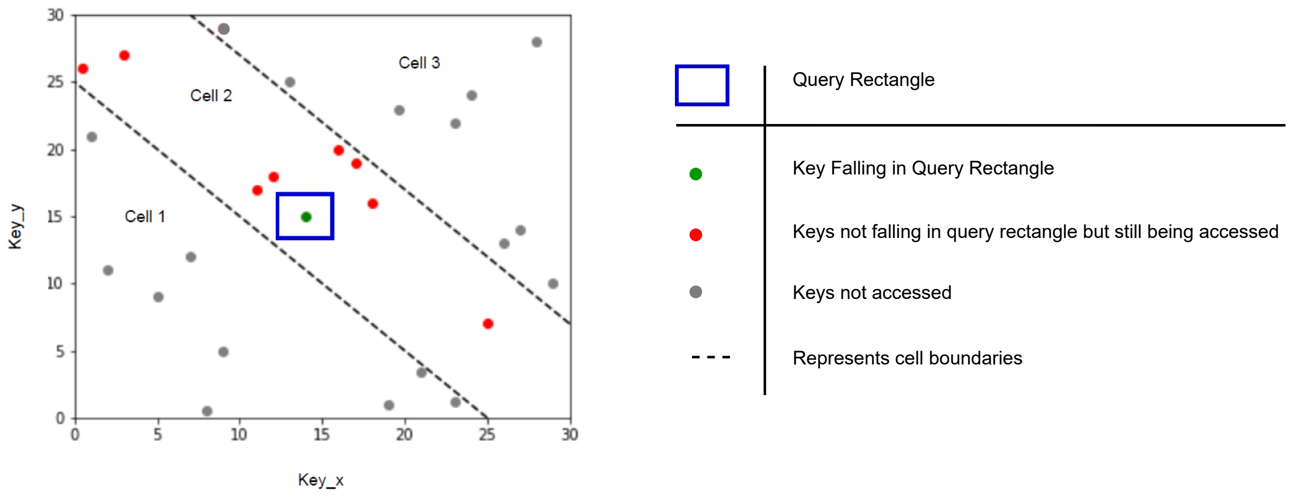


Figure 2.7: An Example of Range Query with LISA Baseline Method

During prediction, we need to find out the cell to which our query point belongs (the 2^{nd} cell in our example).

1. The two dimensional key space is divided into 3 cells using the mapping function $\mathcal{M}((x, y)) = x + y$. Each section in Fig. 2.7 represents one cell.

2. The query point is represented by the blue rectangle. It consists of only 1 key and falls inside the second cell.

3. Identify the cell to which the query rectangle belongs by doing a binary search based on query point mapped value.

4. Once the cell 2 is identified, we need to compare the two dimensional key value of the query point, against all the possible keys in that cell 2 until a match is found. This can results in maximum of 8 irrelevant points being accessed for the point query.

In the Fig. 2.7, the dataset is partitioned into three cells. The query rectangle falls inside the second cell and caused all the data points in the second cell to be checked if they are inside the query rectangle. This results in 8 irrelevant data points (the red points in the graph) accessed for the query range that only contains 1 relevant key.

2.2.4 LISA Approach

$\boxed{\textbf{Motivation}}$ LISA solves this problem by dividing the cells into smaller quadrilateral regions, called shard. Based on the mapped value, LISA builds a shard prediction function and this function will divide the whole space into different shards. Then we can predict the shard where the query key should be located to perform the query.

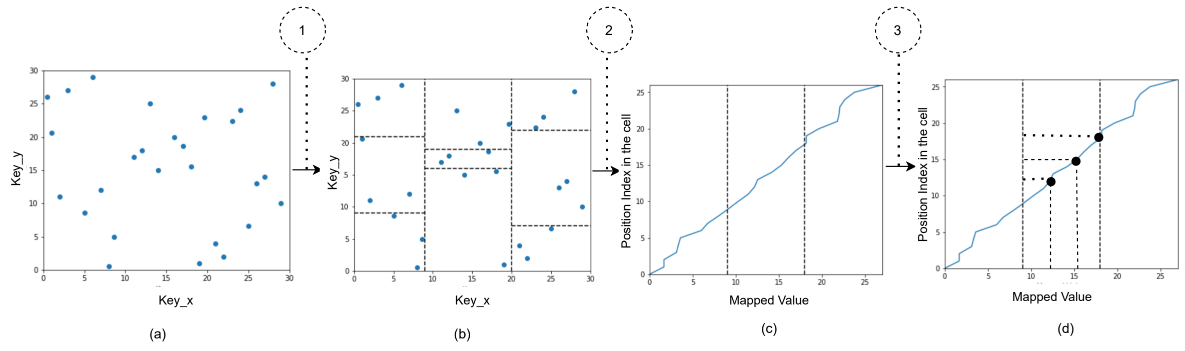


Figure 2.8: LISA Framework.

Given a spatial dataset, we generate the mapping function $\mathcal{M}$ and the shard prediction function $\mathcal{SP}$. Based on them, we build our index structure, LISA, to process point, range and KNN queries. The LISA model consists of four parts: the representation of grid cells, the mapping function $\mathcal{M}$, the shard prediction function $\mathcal{SP}$ and the local models for all shards. As illustrated in the Fig. 2.8, the procedure of building LISA is composed of four parts.

1. Grid cell partition.
2. Mapping spatial coordinates into scalars, i.e. $\mathbb{R}^d \rightarrow \mathbb{R}$.
3. Build shard prediction function $\mathcal{SP}$.
4. Build local models.

Definitions

This section presents the additional definition specific to LISA implementation.

4. **Shard and Shard Prediction**. The shard S is the pre-image of an interval $[a, b) \subseteq [0, +1)$ under the mapping function $\mathcal{M}$, i.e. $S = \mathcal{M}^{-1}([a.b))$. Shard can be regarded as a quadrilateral region in the two dimensional space and a certain shard is mapped into a certain range of mapped values with the mapping function. Given a mapped value, our goal is to predict which shard it belongs to.

   After getting the mapped values of all keys, we learn a monotonic shard prediction function $\mathcal{SP}$ to partition the mapped values into different shards. Assume that $m_i = \inf \mathcal{SP}^{-1}([i, i+1))$ and $m_{i+1} = \inf \mathcal{SP}^{-1}([i+1, i+2))$ for mapped values m_i and m_{i+1}, i.e. m_i is the smallest mapped value in the i to $i+1$ shard, then we define the ith shard as $S_i := \mathcal{M}^{-1}([m_{i-1}, m_i))$.

5. **Local Model**. Local model L_i is a model that processes operations within a shard S_i. It keeps dynamic structures such as the addresses of pages contained by S_i. Local models are not relevant to our implementation as full data-set is loaded in the main memory.

In the following sections, we present the construction process of LISA model step by step.

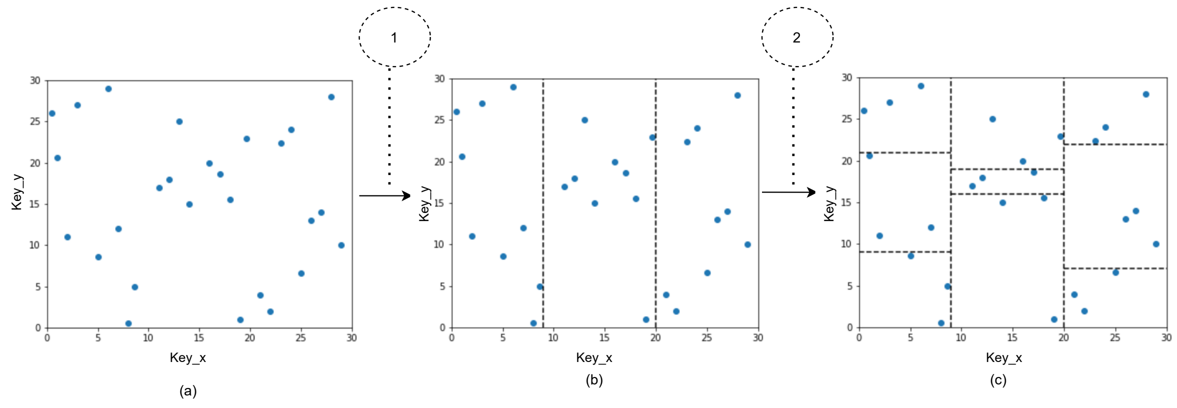


Figure 2.9: Cell Partition Strategy.

Grid Cells Generation

The first task in LISA implementation is to partition the two dimensional key space into a series of grid cells based on the data distribution along a sequence of axes. Then we number the cells along these axes as well. The goal of this partition process is to evenly divide all the data points into smaller cells.

Example 2.7 Consider the example shown in the figure 2.9.

1. Plot A shows distribution of 27 keys in the two dimensional space.
2. In plot B, we first sort Keys on 1^{st} dimension and divide into 3 vertical columns each containing 9 keys.
3. Then for each vertical column of 9 keys, we sort the keys again according to 2^{nd} dimension, and divide the keys in each vetical column into 3 new cells.
4. The total number of cells into which the keys space is divided, N_{cells}, is a hyper-parameter and found empirically using grid search.

We need to sort the key space along the sequence of axis before we partition the keys value along that axis to make sure that cells do not contain overlapping keys.

Algorithm 8: Grid Cell Generation Algorithm for LISA Method

```
input: $N$: length_of_grid_cell;
       trainset: [$(x, y)$; $x \in \mathbb{R}^2$; $y \in \mathbb{R}$]
1  $K$ = len($x$)/($N * N$) // K: Keys per cell
2  $x$ = $x$[argsort($x$[0])] // Sort x based on $1^{st}$ dimension
3  for $i \leftarrow 0$ to $N$ do
4      for $j \leftarrow 0$ to $N$ do
5          cell[$i$+$j \star N$].lower[0] = $x$[$i$ $\star K$ $\star N$][0] // Store keys's x
             coordinate for first key in cell.
6          cell[$i$+$j \star N$].upper[0] = $x$[($i + 1$) $\star K$ $\star N$][0] // Store keys's x
             coordinate for last key in cell.
7      end
8  end
9  for $i \leftarrow 0$ to $N$ do
10     $x$[$i \star K \star N$:($i + 1$)$\star K \star N$] = $x$[argsort($x$[$i \star K \star N$:($i + 1$)$\star K$
         $\star N$])+$i \star K \star N$][1] // Sort x based on $2^{nd}$ dimension
11 end
12 for $i \leftarrow 0$ to $N$ do
13     for $j \leftarrow 0$ to $N$ do
14         cell[$i$+$j \star N$].lower[1] = $x$[$j$ $\star K$ $\star N$][0] // Store keys's y
             coordinate for first key in cell.
15         cell[$i$+$j \star N$].upper[1] = $x$[($j + 1$) $\star K$ $\star N$][0] // Store keys's y
             coordinate for last key in cell.
16     end
17 end
```

In the Algo. 8, segregation of key space into cells is performed in the following steps:

1. Row length of the grid cell N is passed as input. In our implementation, grid cell is constrained to be a square and total number of cells is given by N*N.

2. Then we calculate number of keys per cell (K) and sort x based on 1^{st} dimension.

3. In lines 3 to 8, we divide the key space into N vertical columns each containing K*N keys. Since our cell grid is N*N, for each cell along x dimension, we store the same values of keys's x coordinates for N cells along y dimension, thereby creating a vertical column of N cells for each cell along x dimension. Per cell we need to store meta data for two keys, corresponding to first and last key in the cell.

4. In lines 9 to 11, for each vertical column of K*N keys, we sort the keys again according to 2^{nd} dimension.

5. In lines 12 to 17, we divide the keys in each vertical column into N new cells and store the keys's y coordinates.

Mapping Function

A mapping function $\mathcal{M}$ is a function on the domain $\mathbb{R}^2$ to the non-negative range, i.e $\mathcal{M} : [0, X_0] \times [0, X_1] \to [0, +\infty)$ such that $M(x_i) < M(x_j)$ if $i < j$, where $x_i \in C_i$ and $x_j \in C_j$. That means the mapped value of a key in cell i will always be less than mapped values of a key in cell j, if $i < j$.

Suppose $x = (x_0, x_1)$ and $x \in C_i = [\theta^{(0)}_{i_0}, \theta^{(0)}_{i_0+1}) \times [\theta^{(1)}_{i_1}, \theta^{(1)}_{i_1+1})$ then we define

$$\mathcal{M}(x) = i + \frac{\mu(H_i)}{\mu(C_i)}$$

where $H_i = [\theta^{(0)}_{i_0}, x_0) \times [\theta^{(1)}_{i_1}, x_1)$ and μ is the Lebesgue measure on $\mathbb{R}^2$.

In two dimensional cases, the Lebesgue measure is exactly the area. Hence, the above formula can be interpreted as: the area of the point x divided by the area of the whole cell, plus the index of the cell.

As shown in figure 2.10, in 2-dimensional case, $\frac{\mu(H_i)}{\mu(C_i)}$ represents the fraction of the area covered by the key(x_0, x_1) to the total area of the cell. Since we are adding i, the index of the cell, to this fraction, the mapped value of a key in cell i will always be less than mapped values of a key in cell j, if $i < j$. After calculating the mapped values of the data set, we sort the keys in each cell according to the mapped value. This results in the whole key space to be sorted according to the mapped value. Figure 2.11 shows the mapping of 2 dimensional key space to one dimensional CDF.

In the example below, we demonstrate the calculation of mapped values of two data points that are located in different cells.

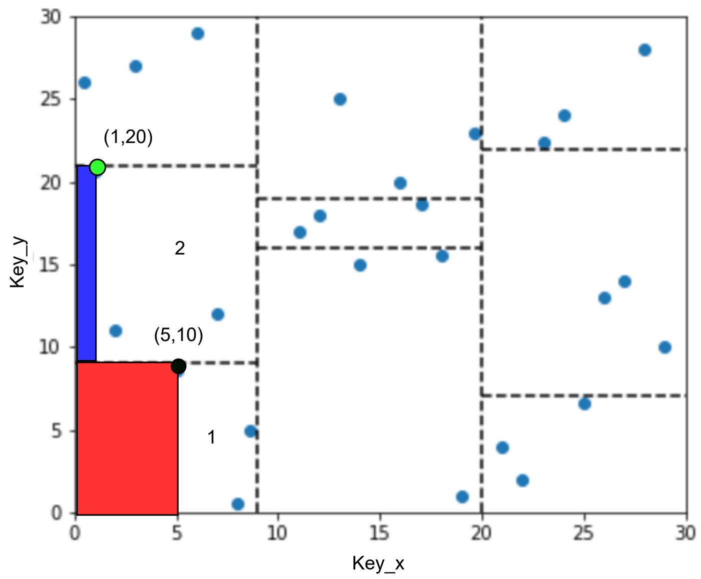


Figure 2.10: Lebesgue Measure Representation for 2 dimensional data.

Example 2.8 In 2.10, we calculate the Lebesgue measure for the black and green points as examples.

1. For the black point in the first cell, the Lebesgue measure will be ratio of area of red rectangle divided by the total area of 1^{st} cell, i.e. $50/100 = 0.5$.
2. For the green point in the second cell, the Lebesgue measure will be ratio of area of blue rectangle divided by the total area of 2^{nd} cell, i.e. $20/100 = 0.2$

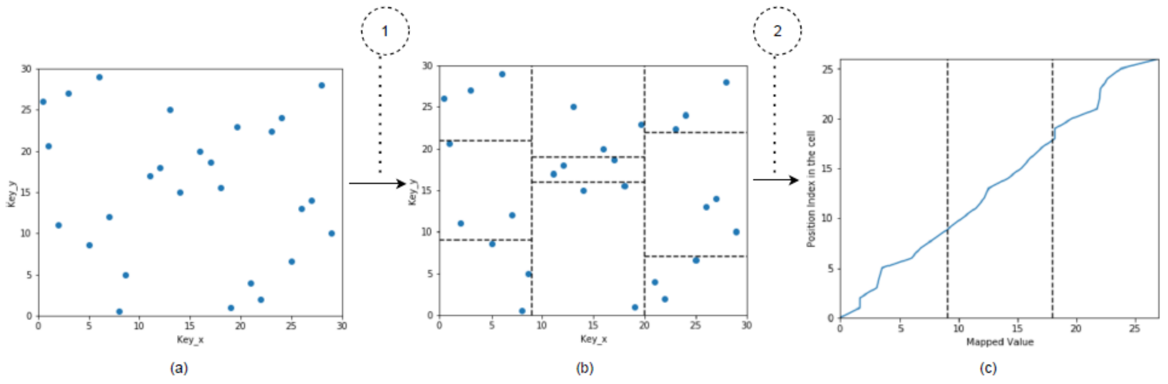


Figure 2.11: Mapping 2 dimensional data points to one dimensional cdf.

Shard Prediction Function

After the mapping function, we get a dense array of mapped values. Then we partition them evenly into U parts and let $\boldsymbol{M}_p = [m_1, \cdots, m_U]$. We train linear regression functions $\mathcal{F}_i$ on each interval and suppose $V+1$ is the number of mapped values that each $\mathcal{F}_i$ needs to process and D is the number of shards per interval. $\Psi = \lfloor \frac{V+1}{D} \rfloor$ is the number of keys falling in a shard.

Example 2.9 For example, assume we have a dense array of 9 mapped values as

$$[1, 1.2, 2, 2.2, 3, 3.3, 3.4, 4, 4.5]$$

and U and D are initialized as 3. So we have $\boldsymbol{M}_p = [9]$ which is divided into 3 equal intervals, $\boldsymbol{M}_p = [m_1, m_2, m_3]$, each containing 3 keys. In this case we have $V+1 = 3$ and will train 3 linear regression functions, 1 for each interval. Each $\mathcal{F}_i$ generates D shards and number of keys falling in a shard will be $\Psi = \lfloor \frac{V+1}{D} \rfloor = 1$.

Then with a given x, the predicted shard is given by $\mathcal{SP}(x) = \mathcal{F}_i(x) + i \times D$, where $i =$ binary-search$(\boldsymbol{M}_p, x)$. More specifically, we first determine i by using binary search. The result tells which interval this x should belong to. Then we find the corresponding linear regression function $\mathcal{F}_i$ and calculate $\mathcal{F}_i(x)$, which is the predicted shard.

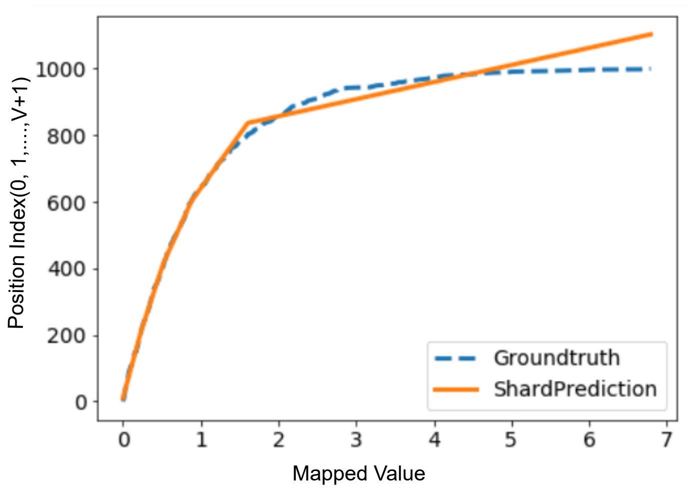


Figure 2.12: Piecewise linear regression functions learnt by the piecewise linear function in the shard prediction training algorithm, $V+1$ is the number of keys per mapped interval.

Example 2.10 In the above example, given a key $x = 1.2$, we first perform binary search in $\boldsymbol{M}_p$ and we found $i = 1$. Then we find the first linear regression function $\mathcal{F}_1$ and calculate $\mathcal{F}_1(x)$. Since each linear regression function will yield $D = 3$ shards, the shards that the first linear regression function generates will be from 0 to 2 and the shards that the second linear regression function generates will be from 3 to 5. Hence, the predicted shard index is given by

$$\mathcal{SP}(x) = \mathcal{F}_i(x) + i \times D$$

Then the problem left is to train the linear regression functions $\mathcal{F}_i$. Let $\boldsymbol{x} = (x_0, \cdots, x_v)$ be the keys' mapped value that fall in $[m_{i-1}, m_i)$. Suppose that $\boldsymbol{x}$ is sorted, i.e. $x_i \leq x_j, \forall 0 \leq i < j \leq v$. Let $\boldsymbol{y} = (0, \cdots, V)$. Then we build a piecewise linear regression function f_i with inputs $\boldsymbol{x}$ and ground truth $\boldsymbol{y}$. For a given point with mapped value $m \in [m_{i-1}, m_i)$, its shard index is given by $\lceil \frac{f_i(m)}{\Psi} \rceil + i \times D$, i.e. $\mathcal{F}_i(x) = \frac{f_i(m)}{\Psi}$.

Example 2.11 In our previous example, in the interval $[0, 2]$, we have $\boldsymbol{x} = (1, 1.2, 2)$ and $\boldsymbol{y} = (0, 1, 2)$. Then for a point with the mapped value $m = 1.2$, the expected output will be $f_i(m) = 1$ and the shard index is given by $\lceil \frac{1}{1} \rceil + 0 \times 2 = 1$. Hence, the point with mapped value $m = 1.2$ will be allocated to the second shard with shard index 1. Then the problem is to train a continuous piecewise linear regression function in each interval. We constrain the piecewise linear regression function to be continuous so that it is guaranteed be monotonic as shown in Figure 2.12.

Formally, a piecewise linear function can be described as

$$f(x) = \begin{cases} b_0 + \alpha_0(x - \beta_0) & \beta_0 \le x < \beta_1 \\ b_1 + \alpha_1(x - \beta_1) & \beta_1 \le x < \beta_2 \\ \vdots \\ b_\sigma + \alpha_\sigma(x - \beta_\sigma) & \beta_\sigma \le x \end{cases} \tag{2.6}$$

In order to make this piecewise linear function continuous, the slopes and intercepts of each linear region depend on previous values. Formally, let $\bar{a} = b_0$, then Eq. (2.6) reduces to

$$f(x) = \begin{cases} \bar{\alpha} + \alpha_0(x - \beta_0) & \beta_0 \le x < \beta_1 \\ \bar{\alpha} + \alpha_0(x - \beta_0) + \alpha_1(x - \beta_1) & \beta_1 \le x < \beta_2 \\ \cdots \\ \bar{\alpha} + \alpha_0(x - \beta_0) + \alpha_1(x - \beta_1) + \cdots + \alpha_\sigma(x - \beta_\sigma) & \beta_\sigma \le x \end{cases} \tag{2.7}$$

Then to make Eq. (2.7) monotonically increasing, we only need to ensure that

$$\sum_{i=0}^{\eta} \alpha_i \ge 0, \forall 0 \le \eta \le \sigma$$

Let $\boldsymbol{\alpha} = (\bar{\alpha}, \alpha_0, \cdots, \alpha_\sigma)$, the square loss function $L(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{i=1}^{V}(f(x_i) - y_i)^2$. We then optimise $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ iteratively.

Assume that $\boldsymbol{\beta} = \hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \hat{\beta}_1, \cdots, \hat{\beta}_\sigma)$ is fixed, then $\boldsymbol{\alpha}$ can be regarded as the least square solution of the linear equation $\boldsymbol{A\alpha} = \boldsymbol{y}$, where

$$\boldsymbol{A} = \begin{bmatrix} 1 & x_0 - \hat{\beta}_0 & \left(x_0 - \hat{\beta}_1\right) 1_{x_0 \ge \hat{\beta}_1} & \cdots & \left(x_0 - \hat{\beta}_\sigma\right) 1_{x_0 \ge \hat{\beta}_\sigma} \\ 1 & x_1 - \hat{\beta}_0 & \left(x_1 - \hat{\beta}_1\right) 1_{x_1 \ge \hat{\beta}_1} & \cdots & \left(x_1 - \hat{\beta}_\sigma\right) 1_{x_1 \ge \hat{\beta}_\sigma} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_V - \hat{\beta}_0 & \left(x_V - \hat{\beta}_1\right) 1_{x_V \ge \hat{\beta}_2} & \cdots & \left(x_V - \hat{\beta}_\sigma\right) 1_{x_V \ge \hat{\beta}_\sigma} \end{bmatrix}$$

where $1_{x_0 \ge \hat{\beta}_1}$ equals to 1 if $x_0 \ge \hat{\beta}_1$, otherwise it equals to 0.
We have

$$\begin{aligned} L(\boldsymbol{\alpha}, \boldsymbol{\beta}) = (\boldsymbol{y} - \boldsymbol{A\alpha})^T(\boldsymbol{y} - \boldsymbol{A\alpha}) &= \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{A\alpha} + \boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{A\alpha} \\ &= \boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{y} + \boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{A\alpha} \end{aligned} \tag{2.8}$$

and if we let

$$\begin{aligned} \frac{\partial L(\boldsymbol{\alpha}, \boldsymbol{\beta})}{\boldsymbol{\alpha}} &= 2\boldsymbol{A}^T\boldsymbol{A\alpha} - 2\boldsymbol{A^T y} = 0 \\ &\implies \boldsymbol{\alpha} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{Ay} \end{aligned} \tag{2.9}$$

we get the $\boldsymbol{\alpha}$ with the given fixed $\boldsymbol{\beta}$. Clearly, different $\boldsymbol{\beta}$ give rise to different optimal parameters. Let $\boldsymbol{\alpha}^\star(\boldsymbol{\beta})$ be the optimal $\boldsymbol{\alpha}$ for a particular $\boldsymbol{\beta}$, then we want to find $\boldsymbol{\beta}$ such that

$$L(\boldsymbol{\alpha}^\star(\boldsymbol{\beta}^\star), \boldsymbol{\beta}^\star) = \min\{L(\boldsymbol{\alpha}^\star(\boldsymbol{\beta}), \boldsymbol{\beta}) | \boldsymbol{\beta} \in \mathbb{R}^{\boldsymbol{\sigma}+\mathbf{1}}\} \tag{2.10}$$

For $\boldsymbol{\beta}$, we define $\boldsymbol{r} = \boldsymbol{A\alpha} - \boldsymbol{y}$ and

$$\boldsymbol{K} = \text{diag}(\bar{\alpha}, \alpha_0, \cdots, \alpha_\sigma), \boldsymbol{G} = \begin{bmatrix} -1 & -1 & \cdots & -1 \\ p_0^{(0)} & p_0^{(1)} & \cdots & p_0^{(V)} \\ p_1^{(0)} & p_1^{(1)} & \cdots & p_1^{(V)} \\ \vdots & \vdots & \ddots & \vdots \\ p_\sigma^{(0)} & p_\sigma^{(1)} & \cdots & p_\sigma^{(V)} \end{bmatrix}$$

where $p_i^{(l)} = -1_{x_l \geq \beta_i}$. Then

$$\boldsymbol{KG} = \begin{bmatrix} -\bar{\alpha} & -\bar{\alpha} & \cdots & -\bar{\alpha} \\ 0 & \alpha_0 p_0^{(1)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \alpha_\sigma p_\sigma^{(V)} \end{bmatrix}$$

then we have

$$g = \frac{\partial L(\boldsymbol{\alpha}, \boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 2\boldsymbol{KGr}, Y = \frac{\partial g}{\partial \boldsymbol{\beta}} = 2\boldsymbol{KGG}^T\boldsymbol{K}^T \tag{2.11}$$

As $g = \nabla_{\boldsymbol{\beta}} L$, $-g$ specifies the steepest descent direction of $\boldsymbol{\beta}$ for L. However, the convergence rate of $-g$ is low as it does not consider the second order derivative of L. Hence, we use Newton's method to perform the update along the direction of second derivative, $s = -\boldsymbol{Y}^{-1}g$. Newton's method assumes that the loss L is twice differentiable and uses the approximation with Hessian The geometric interpretation of Newton's method is that at each iteration, it amounts to the fitting of a paraboloid to the surface of $L(\boldsymbol{\alpha}, \boldsymbol{\beta})$ at the trial value β_k, having the same slopes and curvature as the surface at that point, and then proceeding to the maximum or minimum of that paraboloid. Hessian matrix, Y in our case is positive semidefinite and hence can be inverted.

$$Y = \frac{\partial g}{\partial \boldsymbol{\beta}} = 2\boldsymbol{KGG}^T\boldsymbol{K}^T = 2(\boldsymbol{KG})(\boldsymbol{G}^T\boldsymbol{K}^T) = 2(\boldsymbol{G}^T\boldsymbol{K}^T)^T(\boldsymbol{G}^T\boldsymbol{K}^T) = 2(\boldsymbol{M}^{\boldsymbol{T}}\boldsymbol{M}) \tag{2.12}$$

Y is a full rank matrix as columns of Y are linearly independent (all keys are independent of each other). To prove that Y is positive definite, we need to show that $x^TYx > 0, \forall x \neq 0$. $x^TYx = x^TM^TMx = (Mx)^T(Mx) = \|Mx\|_2^2 \geq 0, \forall x \neq 0$

In the beginning, we set $\beta^{(0)} = x_0$ and $\beta_i^{(0)} = x_{\lfloor i \times \frac{V}{\Psi} \rfloor}, \forall i \in [1, \sigma]$. Then we can obtain $\boldsymbol{\alpha}$ by solving Eq. (2.9). Then at each step, we perform a grid search to find the step $lr^{(k)}$ such that the loss L is minimal. Then at the next iteration, we increase k by one and set

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} + lr^{(k)}s^{(k)}$$

As described in Algorithm 9, we perform following operations during shard training, :

1. Divide the sorted mapped values into equal sized U intervals. We found empirically that training algorithm generalizes better if mapped intervals are aligned with grid cell boundaries. U is initialized to numbers of grid cells.
2. Suppose $V + 1$ is the number of mapped values in each interval and D is the number of shards learned per mapped interval.
3. For each interval, we want to build a monotonic regression model $\mathcal{F}_i$ whose domain is $[m_{i-1}, m_i]$
4. Each $\mathcal{F}_i$ generates D shards and every such shard contains $\Psi = \lfloor \frac{V+1}{D} \rfloor$ number of keys
5. $x = [x_0, \cdots, x_V]$ specifies the keys' mapped values in interval i, $[m_{i-1}, m_i]$
6. Given $V + 1$ sorted mapped values $x = [x_0, \cdots, x_V]$ and their indices $y = [0, \cdots, V]$, each $\mathcal{F}_i$ is built and trained with the procedure mentioned in the algorithm 9.

Algorithm 9: Shard Training Algorithm

```
 input: $M_p$:sorted mapped value array,U: number of mapped intervals, $D$:number of
        shards per interval
1 Partition $M_p$ into equal length U intervals $\boldsymbol{M}_p = [m_1, \cdots, m_U]$
  for $i \leftarrow 0$ to $U$ do
2     $x = [x_0, \cdots, x_V]$ be the keys' mapped values in interval $i$
3     $y = [0, \cdots, V]$
4     Initialize $\beta^{(0)}$ as $\beta^{(0)} = x_0$ and $\beta_i^{(0)} = x_{\lfloor i \times D \rfloor}, \forall i \in [1, \sigma]$
5     while $k <$ iter do
6         Initialize $A^{(k)}$ according to (2.7)
7         $\alpha^{(k)} = ((A^{(k)})^T A^{(k)})^{-1} (A^{(k)})^T y$
8         Calculate $g^{(k)}, Y^{(k)}$
9         $s^k = -(Y^{(k)})^{-1} g^{(k)}$,
10        Find update step $lr^{(k)}$ such that
           $\mathrm{L}(\alpha^\star(\beta^k + lr^{(k)} s^k), \beta^k + lr^{(k)} s^k) = \mathrm{min}\{L(\alpha^\star(\beta^k + lr^{(k)} s^k), \beta^k + lr^{(k)} s^k)\}$
11        $\beta^{k+1} = \beta^k + lr^{(k)} s^k$
12    end
13 end
```

Local Models for Shards

Local models are not relevant to our implementation as full data-set is loaded in the main memory.

2.3 Queries

2.3.1 Point Query

A point query is a database operation that finds the records that exactly match our query conditions. In this project, we perform point query on 1-dimensional data and 2 dimensional data. We assign the database records into pages, predict the page index with the index models and then perform sequential search on the predicted page. In order to evaluate the errors that different index models are making, we focus on predicting the page indices and ignore the sequential search operation on a specific page.

Example 2.12 For example, assume we have an 1-dimensional array $[1, 2, 3, 4]$ and two pages such that $[1, 2] \in P_0$ and $[3, 4] \in P_1$. A point query for $x = 2$ is expected to return 0 as the page index.

Point Query with B-Tree

Searching in a B-tree is similar to searching in a binary search tree. In a binary search tree, we traverse the tree and make a binary decision at each node. Similarly in order to perform point query with a B-tree, we traverse the tree and make a **multi-way** decision at each node.

In our implementation, the point query method with B-tree takes the root node `x` of a subtree and a key `k` to be searched for in that subtree. If `k` is in that subtree, the method returns the node `y` that contains the key `k` and an index `i` such that $\texttt{y.key}_i$=`k`. Otherwise the method will return `-1`. The point query algorithm for a B-tree is illustrated in Algo. 10.

Algorithm 10: B-tree Point Query

```
input: x: The node of the subtree to be searched; k: The key to be searched
Result: y: The node that contains the query key in its keys; i: the index of the query
        key
i=1
while $i \leq x.n$ and $k > x.keys_i$ do
    i=i+1
    if $i \leq x.n$ and $k == x.keys_i$ then
        return x, i
    else if x.leaf then
        return NULL, -1
    else
        return BTreeSearch(x.$c_i$, k)
end
```

In the point query algorithm of B-tree as illustrated in 10, the search is performed with the following steps:

1. From line 1 to 3, we use linear search to find the smallest index i such that `k`$\leq$`x.key`$_i$. If there is no such i, we set i to be `x.n+1`.

2. Then we check whether we have found the key in this node on line 4 to 5. If we have, then the method returns current node and the index of the query key.

3. Otherwise, we check if current node is a leaf node. If it is a leaf node, then we know there is no such query key in this subtree. Hence, this method returns a null node and -1 to indicate there is no such key.

4. If current node is not a leaf node, we then recursively search the appropriate subtree of `x`.

Example 2.13 For example if we were to search for 41 in the Fig. 2.1, we would first compare query key 41 and the keys in root node, which is 31. Hence we go to the second subtree, whose root node contains two values 51 and 71. By comparison, we should go the first subtree of this node. Then we reach the leaf node, which contains our query key 41 and hence the query will return this leaf node and the index 1 as output. If there is no such key, then the method will return `NULL` and -1.

Point Query with KD-Tree

Similar to search with binary search tree, we also need to traverse the tree in order to perform point query. However, we need to switch the dimensions when we compare the values between the query key and the values in the nodes.

Algorithm 11: Point Query with KD-Tree

```
Input  : t: The node being searched; x: The query key; cd: Current dimension
Output: n: the node that contains the query key
1 DIM=2
2 if t==NULL then
3     return NULL
4 if x[0]==t.data[0] and x[1]==t.data[1] then
5     return t
6 else if x[cd]<t.data then
7     return pointSearch(t.left, x, (cd+1) % DIM)
8 else if x[cd]>t.data then
9     return pointSearch(t.right, x, (cd+1) % DIM)
```

The point query works in the following steps:

1. From line 2 to 3, we first check if current node is `NULL`. If so, that means that we have already traversed all the possible nodes and found nothing. In this case, the query returns `NULL`.

2. From line 4 to 5, we check if the current node contains the same key as the query key. If so, the current node is the node that we are looking for. Hence, we return the current node in this case.

3. Otherwise, from line 6 to 9, we check if the current dimension of the query key is smaller, larger or equal to the current dimension of the data in the node.
   a) If it is smaller, then we search on the left subtree of current node, with the same query key and switched dimension.
   b) If it is larger, then we search on the right subtree of current node, with the same query key and switched dimension.

Example 2.14 In the previous figure 2.5, we showed an example KD-tree. If we want to search for $(50, 30)$ in this tree, we would follow the following steps:

1. We first check the root node and compares the x-coordinate. As $50 > 30$, we go to the right subtree of the root node.

2. Then in the subtree, we compare the y-coordinate. As $50 < 70$, we go to the left subtree of this node.

3. Then in the left subtree, the termination condition is reached, hence we return this node as result.

Point Query with Baseline Index Model

The point query with baseline model is the same with forward pass in the training process. As the baseline model is a two-layer fully connected neural network with `ReLU` activation functions, we calculate the output of a given input x with the equation below:

$$\hat{y} = \boldsymbol{w_3}\text{max}(\boldsymbol{w_2}\text{max}(\boldsymbol{w_1}x + \boldsymbol{b_1}, 0) + \boldsymbol{b_2}, 0) + \boldsymbol{b_3} \tag{2.13}$$

As we assumed, the baseline model is approximating the CDF of X. Hence, for a certain x, the output is the probability that $F(X \leq x)$. Since we are working with a static array without insertion and deletion, we can assume that we know the total number of records as N. We also define the page size to be S as a parameter. Then we can calculate the position of this key as $\hat{p} = \lfloor \hat{y} * N \rfloor$.

After knowing the position of the key in the static array, we then calculate the page where it should be allocated to as below

$$\hat{o} = \lfloor \frac{\hat{p}}{S} \rfloor = \lfloor \frac{\hat{y} * N}{S} \rfloor \tag{2.14}$$

Complexity Analysis

For any key, the computation complexities of $\hat{o}$ are the same, as there are only fixed number of computations needed. Hence, the time complexity of query with the baseline model is $\mathcal{O}(1)$, i.e. constant for any training data size.

Point Query with Recursive Model Index

The point query of recursive model is a top-down process. With a given x, the root model will first predict an output that represents the probability that $F(X \leq x)$. Then we map this output into the index of models in the next stage. Afterwards, we use that model to predict an output with the given x. We iterate these steps until the last stage in which we use the output as the final output. The above process is described in Algorithm. 12

Algorithm 12: Point Query With Recursive Model Index

```
   input: x; models; num_of_stages
1  stage← 0
2  next_model← 0
3  while stage <num_of_stages do
4      model=models[stage][next_model]
5      output = model.predict(x)
6      if stage==num_of_stages-1 then
7          y = output
8      else
9          next_model=output*len(models[stage+1])
10         stage = stage+1
11     end
12 end
13 return y
```

In the query algorithm, we have three inputs: x as the query key, trained models and the number of stages.

On line 1-2, we first initialise the *stage* and *next_model* to be 0, so that we use the root model at the very beginning. Then on line 3, we iterate over all stages. In each stage, we perform the following actions:

1. On line 4, we access the model at *stage* whose index is *next_model*.

2. On line 5, we perform the prediction with the query key x and the model selected by the previous step.

3. On line 6, we check if current stage is the last stage.

   a) If it is, then we get the final output, which equals to the output from line 5.

   b) If it is not the last stage, then we map the output from previous step into the index of the model in the next stage. As there are `len(models[stage+1])` models in the next stage and the output represents some probability (hence, `output` $\in [0, 1]$), we multiply them and find the `next_model`. In the meanwhile, we add 1 to the stage.

4. At the end, we return the final output, which is the output from the model in the last stage.

After calculating the output as described in the Algorithm. 12, we calculate the page index in a same way as we described in the baseline model.

Point Query with LISA

Algorithm 13: Prediction Algorithm for LISA Point Query

```
   input  : q:query_point;
            $D$:number_of_shards_per_mapped_interval;
            cell :cell_metadata_array;
            $x$ :training_database_array
   Output: q.value:point_query_value
1  cell_found = False
2  for $i \leftarrow 0$ to len(cell) do
3      if q[0] $\in$ [cell[i].lower[0], cell[i].upper[0]) then
4          if q[1]$\in$ [cell[i].lower[1], cell[i].upper[1]) then
5              cell_found = True
6              break
7          end
8      end
9  end
10 if cell_found==True then
11     q.map_v=cell[$i$].id +(q.area/cell[$i$].area)
12     q.map_i=binary_search(cell.map_v_array, q.map_v)
13     shard_id = q.map_i$\star D$ + cell.shard_prediction(q.map_v)
14     $K$=cell.keys_per_shard
15     shard_offset=shard_id$\star K$
16     for $i \leftarrow$ shard_offset to shard_offset +$K$ do
17         if (q[0] == x[i][0]) and (q[1] == x[i][1]) then
18             return $x$[$i$].value
19         end
20     end
21     return -1
22 end
23 return -1
```

In the Algo. 13, Point query search is performed in following steps:

1. In lines 2 to 8, find the cell to which point query belongs by comparing the query key value with first and last key in each cell. First key in the cell represents the lower corner of the cell, whereas last key in the cell represents the upper corner. This search will be linear in the number of cells.

2. On line 11, calculate the mapped value of query point, as mentioned in the Section *Mapping Function* 17.

3. During model training, 2-D key space is mapped into a sorted one dimensional array. On line 12, find the mapped interval to which query point's mapped value belongs using binary search on this array.

4. On line 14, predict the shard index for calculated mapped interval. It is found empirically that predicted shard index can differ from ground-truth value by 1 for keys falling near the shard boundaries.

5. In lines 16 to 20, search for the query key in the predicted shard by sequentially comparing against all the keys in the shard until a match is found.

6. In case of no match, repeat the previous step in adjacent left and right shards as predicted shard index can have an error of 1.

2.3.2 Range Query

A range query is a database operation that retrieves all the records that lies in a range. In this project, we perform range query on 2-dimensional data only. In addition, we only consider a range query where the range is defined as a rectangle. Under these assumptions, a range query can be formalised as a query $\mathcal{Q}(\boldsymbol{l}, \boldsymbol{u})$ where $l, u \in \mathbb{R}^2$.

Example 2.15 For example, assume we have the points $[(1,2),(3,4),(3.5,4),(5,6)]$ and the range query $\mathcal{Q}((2,3),(5,5))$, as shown below:

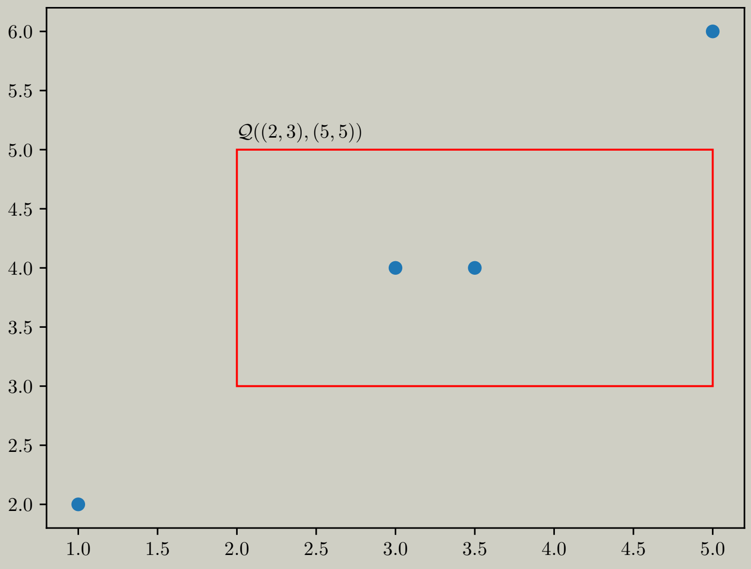
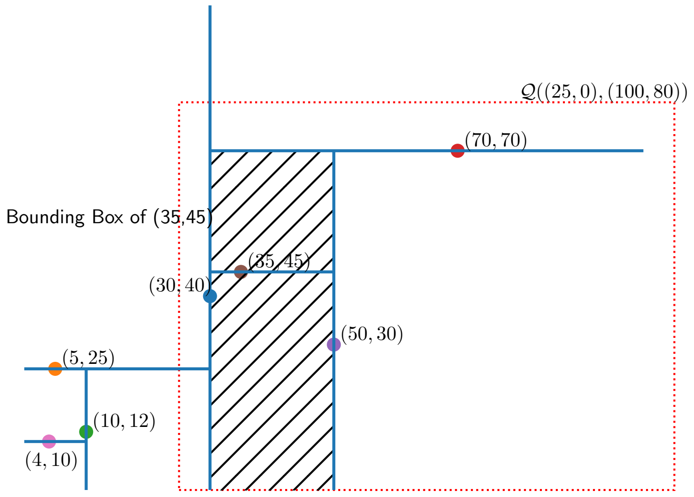


Figure 2.13: A Range Query Example where $\mathcal{Q}(\boldsymbol{l}, \boldsymbol{u}) = \mathcal{Q}((2,3),(5,5))$

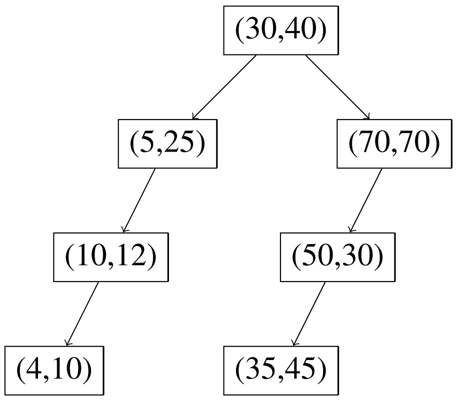
In this example, the range query should return the points that lies inside the red rectangle, i.e. $((3,4),(3.5,4))$.

Range Query with KD-Tree

Performing range query in a KD-tree needs to traverse the whole tree, but we can apply some pruning strategy to avoid useless searching for some nodes. We first present the **bounding box** of subtree in a KD-tree.



(a) The 2D space and the partition of KD-tree



(b) The tree structure of the KD-tree

Figure 2.14: An example of KD-Tree

Bounding Box

When we are traversing the KD-tree along the way, we are assured that a node is bounded in a rectangle region. Assume that a node `t` has k ancestors, then the node `t` is bounded in a rectangle in the following way:

1. We traverse from the root node whose coordinate is (x_r, y_r). If we go to the right subtree, then the node `t` is bounded in the right side of the root node. That means, the right subtree of the root node is bounded in a rectangle region determined by the lower bound $\boldsymbol{l} = (x_r, 0)$[1] and the upper bound $\boldsymbol{u} = (\infty, \infty)$. Similarly, we can determine the bounds of the left subtree as $\boldsymbol{l} = (0, 0)$ and $\boldsymbol{u} = (x_r, 0)$. We call the bounding box at this level as B_0^r and B_0^l for the right subtree and left subtree respectively.

2. We then traverse into the next level and determine the bounding box B_1^r and B_1^l of the subtrees rooted at the root's child node. At this time, we will need to switch the axis into the y-axis. If the child node is in the left subtree of the root node, the final bounding box is the intersection between the bounding box at this level and the bounding box of left subtree at the root level, i.e. $B_0^l \cap B_1^r$ and $B_0^l \cap B_1^l$. Similarly, if the child node is in the right subtree of the root node, the final bounding box are $B_0^r \cap B_1^r$ and $B_0^r \cap B_1^l$.

3. We traverse until the left node and determine the bounding box of each subtree and save this property into the root node of each subtree.

[1]For simplicity and clarity, we assume our keys are starting from $(0, 0)$ and are positive in both dimensions

Example 2.16 In the Fig. 2.14, we present the bounding box of the leaf node $(35, 45)$ as the hatched area. In this example we will demonstrate how we calculate this.

1. We first start with the root node and we go to the right subtree. Hence the bounding box of $(70, 70)$ will be $\boldsymbol{l} = (30, 0)$ and $\boldsymbol{u} = (\infty, \infty)$.

2. Then we traverse to the $(70, 70)$ and we go to the left subtree. Hence the bounding box of $(50, 30)$ will be $\boldsymbol{l} = (0, 0)$ and $\boldsymbol{u} = (0, 70)$. By intersecting with the bounding box from the first step, we get the final bounding box as $\boldsymbol{l} = (30, 0)$ and $\boldsymbol{u} = (0, 70)$, which refers to the right bottom area in the figure.

3. We then go to the left subtree and calculate the bounding box and get $\boldsymbol{l} = (0, 30)$, $\boldsymbol{u} = (50, 70)$, i.e. the hatched region in the figure.

With the bounding box, there are three conditions in our pruning strategy while traversing the tree:

- If the bounding box does not overlap with the query rectangle, we stop the recursion and traverse the subtree.

- If the bounding box is a subset of a query box, then we report all the points in current subtree.

- If the bounding box overlaps query box, then we recurse the left and right subtrees.

Formally, the algorithm for range query is illustrated as in Algo. 14.

Algorithm 14: KD-tree Range Query

```
input: Q: The query rectangle; T: The root node of a subtree to be range searched
Result: S: The set of all nodes that are in the query range
S=$\phi$
if T==NULL then
    return $\phi$
if T.range $\cap$ Q==$\phi$ then
    return NULL
if T.range $\subset$ Q then
    return AllNodesUnder(T)
if T.data $\in$ Q then
    S = S.union({T.data})
S = S.union(RangeQuery(Q, T.left))
S = S.union(RangeQuery(Q, T.right))
return S
```

In the above algorithm, we perform the range query with the following steps:

1. First we check if the node is NULL, if so, we simply return an empty set.

2. On line 4-5, we check if the bounding box is overlapping with the query rectangle, by comparing the bounds of the query rectangle and the bounding box.
3. On line 6-7, we check if the bounding box is a subset of the query rectangle. If it is, then we will traverse the subtree of `T` and simply return all nodes that are contained in the subtree.
4. On line 8-11, we cannot apply any pruning strategy. Hence, we first check if the data is inside the query rectangle by comparing the coordinates with the query rectangle. If it is inside, then we put the point into our result set. Then we recurse to the left and right subtree and append the results into our result set `S`.

Example 2.17 In Fig. 2.14, we present an example query rectangle $\mathcal{Q}((25,0),(80,80))$. In this example, we show how we perform range query with this rectangle. We assume that our space is from 0 to 100 on both dimension so that there is no ∞ bounds.

1. First we start with the root node, whose bounding box is the whole space. Hence we check if $(30,40) \in \mathcal{Q}$. Since it is inside, we add it to the result set $S = \{(30,40)\}$.
2. We then look at the left subtree, whose bounding box is $(l) = (0,0)$, $\boldsymbol{u} = (30,100)$. As there is some overlapping with the query rectangle, we check if $(5,25)$ is inside the query rectangle. Since it is not in the rectangle, we do not put it in the result set.
3. Then we go to $(10,12)$ whose bounding box is $(l) = (0,0)$, $\boldsymbol{u} = (30,25)$, which is overlapping with the query rectangle. Hence, we need to check if it is inside the query rectangle. Since it is not in the rectangle, we do not put it in the result set.
4. **No Overlapping** We then move to $(4,10)$ whose bounding box is $\boldsymbol{l} = (0,0)$, $\boldsymbol{u} = (10,25)$ which is not overlapping with the query rectangle. Hence, we prune the whole subtree rooted at $(4,10)$.
5. We then move to the right subtree of the root node, whose bounding box is $(l) = (30,0)$, $\boldsymbol{u} = (100,100)$. As there is overlapping between the query rectangle and the bounding box, we then check if $(70,70)$ is inside the query rectangle. We then put it in the result list as it is inside the query rectangle. $S = \{(30,40),(70,70)\}$.
6. **Subset** Then we go to the left subtree whose bounding box is $(l) = (30,0)$, $\boldsymbol{u} = (100,70)$. The bounding box is fully inside the query rectangle, and hence we add all the results under $(70,70)$ in to the result list. Finally we have

$$S = \{(30,40),(70,70),(50,30),(35,45)\}$$

Range Query with LISA

For a range query $\mathcal{Q}(\boldsymbol{l},\boldsymbol{u})$, we first find the cells that overlap with $\mathcal{Q}$. Then we decompose $\mathcal{Q}$ into the union of smaller query rectangles $\bigcup \mathcal{Q}_i$ such that each smaller query rectangles intersects only one cell, as shown in the Fig. 2.15.

Suppose that $\mathcal{Q} = \bigcup \mathcal{Q}_i$ where $\mathcal{Q}_i = [l_{i_0}, u_{i_o}) \times [l_{i_1}, u_{i_1})$, i.e. we have $\mathcal{Q}_i$ representing the ith smaller query rectangles of one cell C_j.

Then we can calculate the mapped values of $\mathcal{Q}_i$, i.e. $\mathcal{M}(l_{i_0}, l_{i_1})$ and $\mathcal{M}(u_{i_0}, u_{i_1})$. For simplicity, we use $m_l^{(i)}$ and $m_u^{(i)}$ to denote $\mathcal{M}(l_{i_0}, l_{i_1})$ and $\mathcal{M}(u_{i_0}, u_{i_1})$ respectively.

After creating corresponding mapped values, we then apply the shard prediction function $\mathcal{SP}(m_l^i)$ and $\mathcal{SP}(m_u^i)$ to predict the shard that could possibly contain keys that lie in the query rectangle $\mathcal{Q}_i$. Then in each shard, we perform a sequential search to find the desired keys.

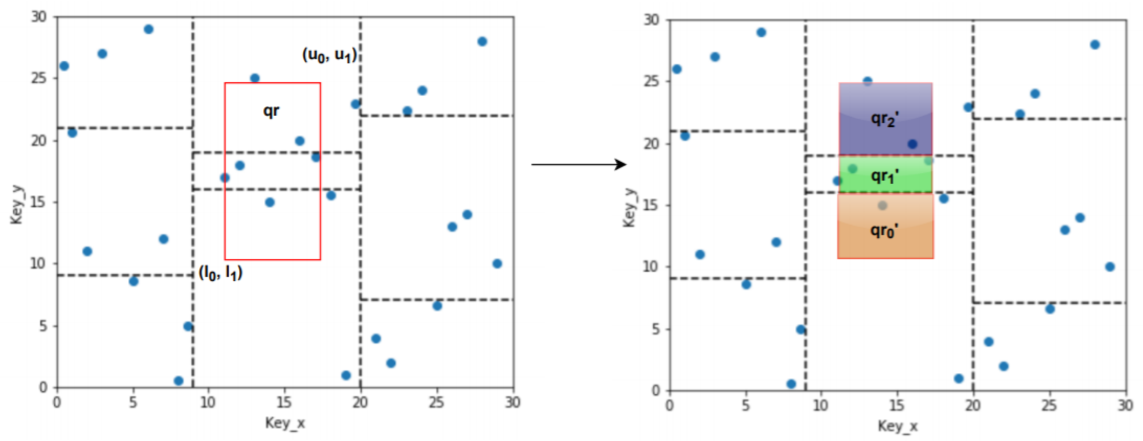


Figure 2.15: Range Query Search in LISA

Example 2.18 In Fig. 2.15, we present a range query example. Following steps show how we perform range query represented by the red rectangle.

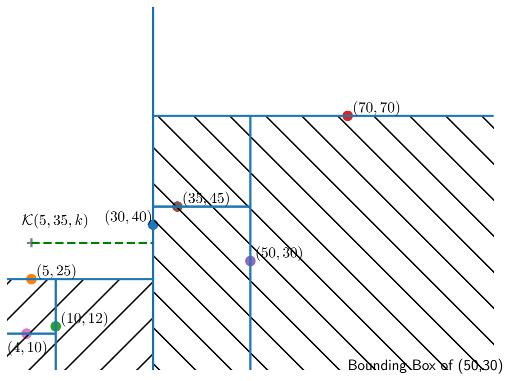
1. Find the cells that overlap with query rectangle qr. In our example, 3 cells are overlapping with the query rectangle.
2. Decompose qr into the unions of smaller query rectangles, each of which intersect one only one cell as represented by $qr_0^{'}$, $qr_1^{'}$ and $qr_2^{'}$ in Fig. 2.15.
3. Calculate mapped values of $qr_0^{'}$, $qr_1^{'}$ and $qr_2^{'}$ vertices represented by $m_l^0, m_u^0, m_l^1, m_u^1, m_l^2, m_u^2$.
4. Find shards corresponding to lower and upper coordinates for each smaller query rectangle represented by $\mathcal{SP}(m_l^0), \mathcal{SP}(m_u^0), \mathcal{SP}(m_l^1), \mathcal{SP}(m_u^1), \mathcal{SP}(m_l^2), \mathcal{SP}(m_u^2)$
5. Do a sequential search for each shard and collect the keys that fall in the query rectangle.

2.3.3 KNN Query

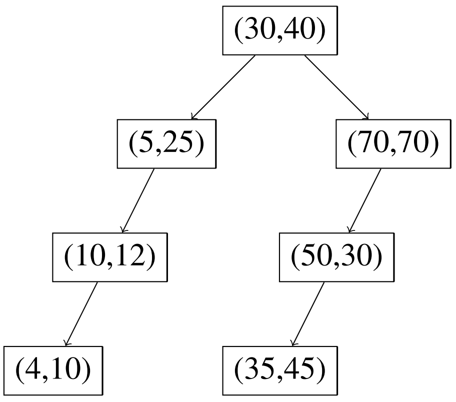
K-Nearest Neighbours (KNN), as the name suggests, is the process of finding K nearest neighbours to a given query point. In this project, KNN query is only performed on 2-dimensional data. We use ℓ_2 norm as the distance metric. A KNN query will be formalised as $\mathcal{K}(\boldsymbol{X})$ where $\boldsymbol{X} \in \mathbb{R}^2$.

KNN query with KD-Tree

In range query with KD-tree, we traverse the whole tree with a pruning strategy. Similarly, to perform KNN query with KD-tree, we also need to traverse the whole tree with a pruning strategy. We first present how to perform the nearest neighbour query (i.e. a KNN query with $K = 1$), and then extend it into general KNN query.



(a) The 2D space and the partition of KD-tree



(b) The tree structure of the KD-tree

Figure 2.16: An example of KNN Query with KD-Tree

Nearest Neighbours

During the query for the nearest neighbour, we use two global variables `best` and `best_dist` to store the nearest node and the nearest distance. During the traversing of the tree, we update the best points and the best distance if we find a closer node to the query point.

Similar to the pruning strategy in the range query with KD-tree, we can calculate the distance between the query point to a bounding box of a node. If the query point is inside the bounding box, then the distance will be 0. If the current shortest distance is larger than or equal to the distance between the query point and the bounding box, then there will not be any points that are closer to the query point than the current closes point. Therefore, we can avoid searching the subtree that will not contain any closer points.

With the pruning strategy, we traverse the tree in an order that maximises the change for pruning. We always start with the subtree that are closer to the query point. In other words, we search the subtree that would contain the query point if we want to insert the query point below the root of a subtree.

Example 2.19 For example, we demonstrate an example KNN query $\mathcal{K}(5, 35, k)$ whose query point is $(5, 35)$ in Fig. 2.16. We perform the nearest neighbour search in the following order.

1. First we start with the root node, which is $(30, 40)$. The bounding box of the root node is the whole space, and we cannot apply the pruning strategy. If we want to insert the query point $(5, 35)$ into the subtree rooted at the root node, we will insert it into the left subtree by comparing the x-coordinate. Hence, we go to the left subtree in the next step. At the root level, the best distance is $\sqrt{(30-5)^2+(40-35)^2} = 25.5$ and the closet point is $(30, 40)$.

2. Then we traverse to the left subtree and calculate the distance. The bounding box of this node is $\boldsymbol{l} = (0, 0)$ and $\boldsymbol{u} = (30, 100)$. The distance between the query point and the bounding box will be 0 as it contains the query point. Hence we cannot apply the pruning strategy. The distance is 10 and it is smaller than the old distance. Hence at this level, the best distance is 10 and the closest point is $(5, 25)$. Then if we want to insert the query point into the subtree under $(5, 25)$, we will compare the y-coordinate and then we need to first traverse the right subtree.

3. As there is no right subtree, we still go to the left subtree. The bounding box of this node is $\boldsymbol{l} = (0, 0), \boldsymbol{u} = (30, 25)$ and the distance between the bounding box and the query point is 10. The distance is the same with the best distance so far. Therefore we apply the pruning strategy and do not need to traverse the subtree.

4. Similarly, for the right subtree of the root node, $(70, 70)$ whose bounding box is $\boldsymbol{l} = (30, 0), \boldsymbol{u} = (100, 70)$, the distance between the query point and the bounding box is $\sqrt{(30-5)^2+(40-35)^2} = 25.5$, which is larger than the current best distance. Therefore we apply the pruning strategy and do not traverse the right subtree of the root.

K-Nearest Neighbours

For general KNN query, we follow the same traversal and pruning strategy. The only difference is that we use a max heap of size k to store the best K distance we have.

A max heap is a special binary tree that satisfies a property: if `P` is a parent node of `C`, then the data in `P` is greater than or equal to the data in `C`. We store the max heap as a binary tree and use a function `heapify` to make it a max heap.

Then we follow the steps below to construct and fill the max heap:

1. First we construct an empty binary tree.

2. We traverse the KD-tree with the pruning strategy as described in the nearest neighbour query. For each element, we decide if we need to push it into the max heap in the following way:

    - For the first K elements, they will be added to the binary tree no matter how close it is to the query point. We call the `heapify` function to ensure the resulting binary tree is a max heap.
    - After the Kth element, we compare the distance between the element and the query point with the root of the binary tree. Since we have used `heapify` and the root is the largest distance so far.
        - If the distance is smaller than the root, then we make it root and call `heapify`.
        - Otherwise, we ignore it.

3. Finally the max heap has K smallest elements and the root of the heap is the Kth smallest element.

Formally, we use the algorithms as illustrated in Algo. 15 to perform KNN query with KD-tree.

Algorithm 15: KD-tree KNN Query

```
   input: P: The query point; T: The root node of a subtree to be searched; K: The
          number of nearest neighbours H: the binary tree that store the result;
          cd: Current dimension
   Result: H: The binary tree that store the result
 1 DIM = 2
 2 H = empty max heap
 3 if T==NULL then
 4     return;
 5 if H.size()<K then
 6     H.insert(T)
 7     H.heapify()
 8     return;
 9 else if dist(P, T.range) > H.root.data then
10     return
11 else
12     if dist(P, T.data) < H.root.data then
13         N=NewBinaryTreeNode(dist(P, T.data))
14         H.root=N
15         H.heapify()
16     if P[cd]<T.data[cd] then
17         KNN(P, T.left, K, H, (cd+1)%DIM)
18         KNN(P, T.right, K, H, (cd+1)%DIM)
19     else
20         KNN(P, T.right, K, H, (cd+1)%DIM)
21         KNN(P, T.left, K, H, (cd+1)%DIM)
```

In the above algorithm, we perform the following operations:

1. From the line 1 to 2, we first initialise the `H` to be an empty max-heap and the maximal dimension is set to be 2.

2. From the line 3 to 4, we check if the current node is already the leaf node. If so, we do nothing and end the traverse process.

3. From the line 5 to 8, we insert the first K elements into the max heap. As the max heap is implemented as a binary tree, the insertion process is the same with the insertion into a binary tree. After the insertion of each item, we apply the `heapify` function and transform the tree into a max heap. In the heap `H`, the root contains the maximal distance that we found so far.

4. From the line 9 to 10, we compute the distance between the query point and the bounding box of current node. If the distance is larger than the root of the heap, then we stop the recursion.

5. From the line 12 to 15, we find the distance that we computed in the last step is smaller than the maximal distance. Hence it should be inserted into the max heap. In this case, we create a new node that contains the current distance, make it the new root of current max heap and perform `heapify` with the new max heap.

6. From the line 16 to 21, we recursively traverse the left and right subtree of current node. If the value of the query point is smaller than the current node at the current dimension, we traverse the left subtree first and then the right subtree. Otherwise, we traverse the right subtree first and then the left subtree.

7.

KNN Query with LISA

$\boxed{\textbf{Motivation}}$ As there is no tree-structure in LISA model, we cannot perform the traversal and pruning operations as we do in the KD-tree. In order to perform KNN query, LISA model converts it into a range query and gradually adapts the size of the range, until the query is finished.

In the Fig. 2.17, we assume a KNN query at the point (x_0, x_1) and we want to find 3 nearest neighbours. We follow the following steps to complete this query.

1. We start with a query distance δ_0, then we perform range query within the circle of sky blue. As we do not have range query defined with a circular range, we then approximate it by converting the range into a square, which is shown in the same colour.

2. With the range query, only two points are reported, which is less than the given $K = 3$. To handle this case, we increase the query bound from δ_0 to δ_1, which results in the green circle in the figure. Similarly, we convert this circle into a square for performing the range query. After this query, we find 4 data points, which is more than what we need.

3. Then we take the average of δ_0 and δ_1 and get the new query distance $\delta = \frac{1}{2}(\delta_0 + \delta_1)$. At this time we get the black dashed circle and its corresponding query range. We can iterate this process for many times to determine the proper query range.

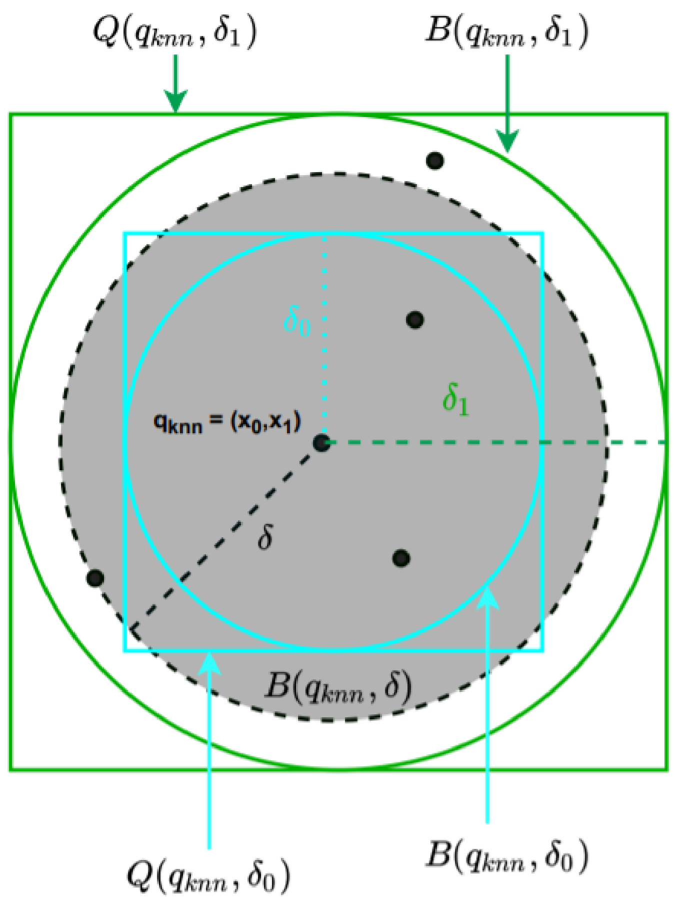


Figure 2.17: KNN Query Implementation in LISA (K=3). In this figure, q_{knn} represents the query point. $\mathcal{Q}(x, \delta) \triangleq [x_0 - \delta, x_0 + \delta) \times [x_1 - \delta, x_1 + \delta)$ represents query rectangle and $\mathcal{B}(x, \delta)$ represents the key space at distance δ containing K nearest keys.

Formally, for a query point $q_{knn} = (x_0, x_1)$, let $x^{'} \in V$ be the Kth nearest key to x in database at a distance value $\delta = \|x^{'} - q_{knn}\|_2$. Lets define $\mathcal{Q}(q_{knn}, \delta) \triangleq [x_0 - \delta, x_0 + \delta) \times [x_1 - \delta, x_1 + \delta)$ and $\mathcal{B}(q_{knn}, \delta) \triangleq \{p \in V \mid \|q_{knn} - p\|_2 \leq \delta\}$. We can create a query rectangle $qr = \mathcal{Q}(q_{knn}, \delta + \epsilon)$ where $\epsilon \to 0$. As shown in Fig. 2.17, K nearest keys to q_{knn} are all in $\mathcal{B}(q_{knn}, \delta)$ and thus in $\mathcal{Q}$. KNN query can be solved using the range query if we can estimate an appropriate distance bound δ for every query point.

In our experiments, we find the initial δ empirically. We try with different values of the initial δ and choose the one for which we get the best results.

2.4 Summary

In this chapter, we illustrate the construction of classic tree-based structures and learned indexes for both one and two dimensional data. Based on these indexes, we also show how to perform queries with these indexes.

For one dimensional data, we have three different indexes:

- B-Tree. As a classic tree-based index, it can be constructed in $\mathcal{O}(n \log n)$ time complexity and cost $\mathcal{O}(n)$ space. With B-Tree, we can perform point query in $\mathcal{O}(\log n)$ time complexity and always get an exact position.
- Baseline Learned Index. The baseline learned index is a simple fully connected neural network with a constant space usage. It can be used to perform point query with constant time complexity, but we are not assured that we can get an exact position.
- Recursive Model Index. It is a composition of several indexes and each internal model is responsible for a certain part of the whole data. It can also perform point query within constant time complexity. We are not assured that we can get an exact position with recursive model as well.

For two dimensional data, we showed two different indexes:

- KD-Tree. As a classic tree-based index, we can construct the KD-tree and perform point, range and KNN queries with it. The time complexity for construction, store and queries are all correlated to the number of data points.
- LISA model. It extends the idea of learned index from one dimensional to two dimensional by mapping the two dimensional keys into one dimensional scalars. We showed that it could also support three different types of queries.

3 Evaluation

Summary In this chapter, we describe how we evaluate the database indexes that we have implemented in previous chapter. For both one and two dimensional data, we use manually synthesised dataset that are generated from a certain distribution as our dataset. This chapter is organised into two sections, where the first section describes the experiment settings and results for one-dimensional data and indexes and the second section describes the two-dimensional data.

3.1 One Dimensional Data and Indexes

For one dimensional data, the evaluation covers the following tasks:

- Find a structure for recursive model index empirically.
- Compares the performance between baseline model, recursive model and traditional B-Tree.

3.1.1 Dataset

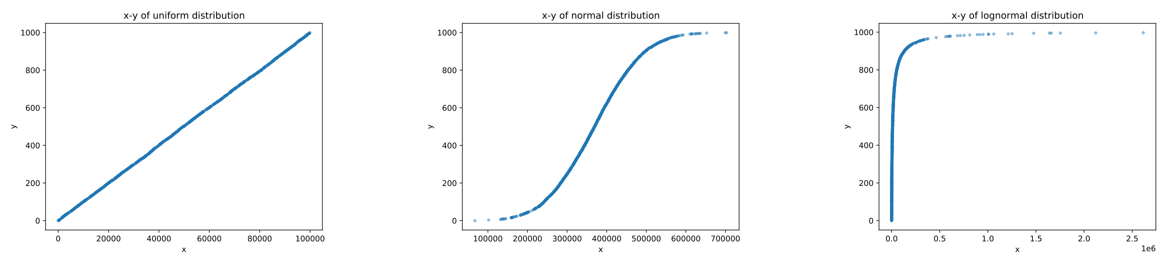
For one dimensional case, we manually generate two columns of the data:

- The first column contains the keys X, which is randomly sampled from a given distribution.
- Then we assign the keys into different pages according to a preset parameter N_{page} for page size. Specifically, the first N_{page} keys will be assigned to the first page, the second N_{page} keys will be assigned to the second page and so on so forth. After the assignments, we set the second column Y to be the page index of the corresponding x.

3.1.2 Hyper-parameters Search

We first generate $10,000$ data points where X is from a lognormal distribution Lognormal$(0, 4)$. In the Fig. 3.1, we illustrate the $x - y$ relations where X is randomly sampled from a uniform, normal or lognormal distribution.

We use three groups to find the best recursive model for lognormal data.



(a) Uniform **(b)** Normal **(c)** Lognormal

Figure 3.1: The x-y graph where x is randomly sampled from a certain distribution

- All models are fully connected neural networks. The numbers of second-level models are $200, 400$, and 600 respectively. The numbers of third-level models are $2000, 4000$ and 6000 for each number of second-level models.
- All models are linear regression models. The number of second-level models are 200, 400, and 600 respectively. The number of third-level models are $2000, 4000$ and 6000 for each number of second-level models.
- Models are combinations of fully connected neural networks and linear regression models. The numbers of second-level and third-level models are determined by the best settings in the previous two group.

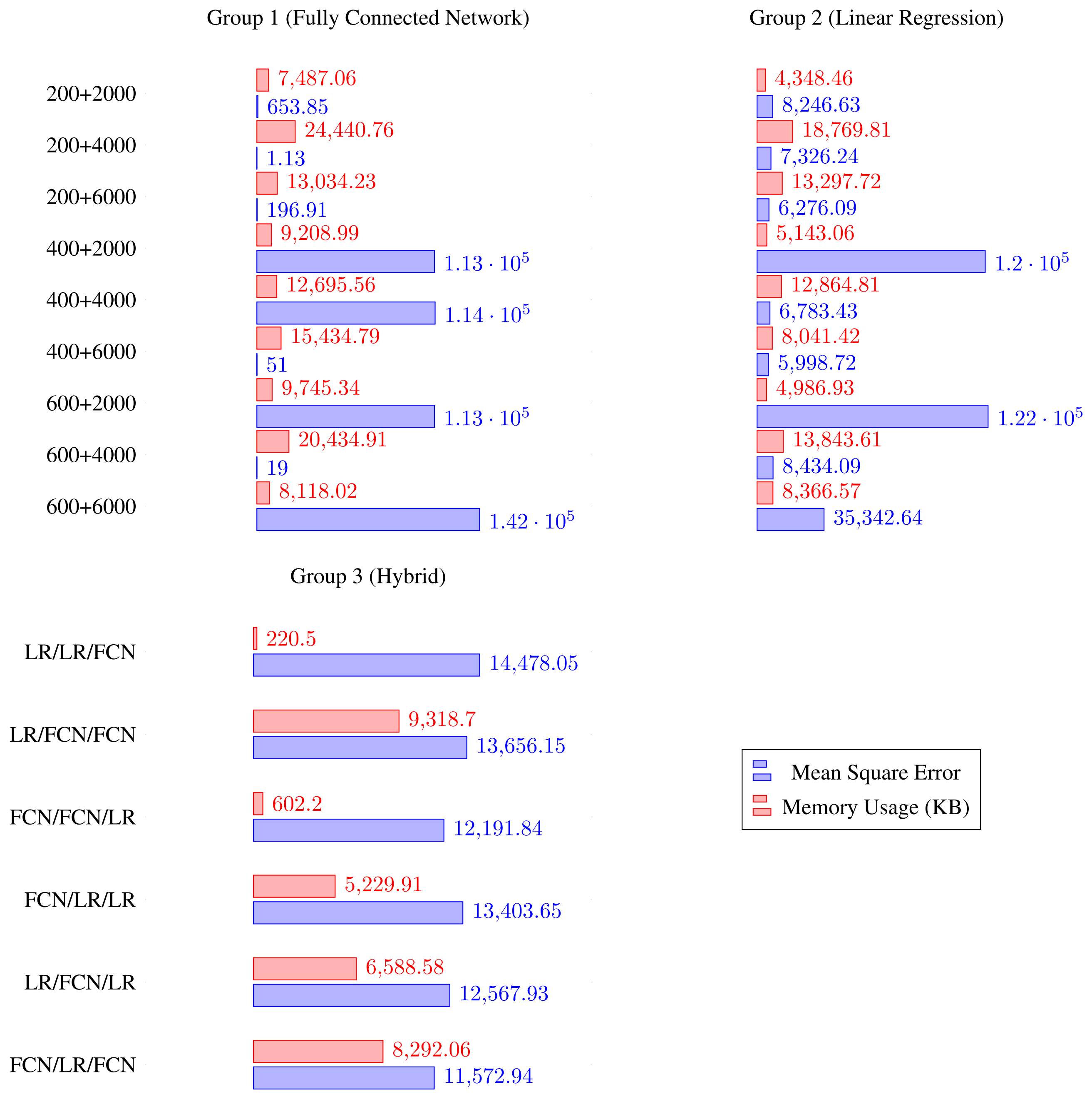
From the experiment results, we found that the second setting in group 1 (1 FCN model as root, 200 FCN models as second-level models and 4000 FCN models as third-level models) is the best regarding the mean square error. We also have the following findings in this searching process.

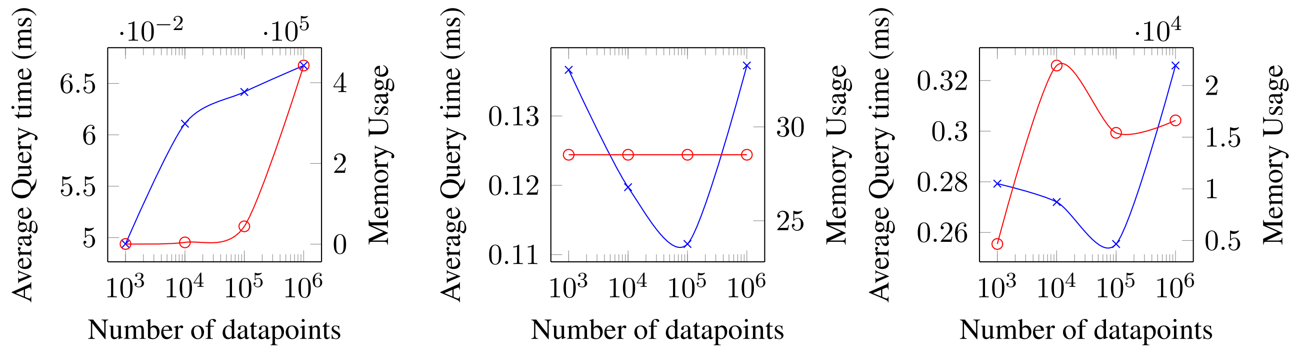
- Generally, the average error in the group 1, where all models are fully connected neural networks, is less than the error in the group 2. We conclude that the fully connected neural networks have the potential to be more accurate, i.e. it could achieve a small error if we tuned the parameters properly.
- Tuning a model is tedious and can be costly. There are lots of hyper-parameters to choose from, such as the number of models in each level, types of models in each level, number of levels, and the internal hyper-parameters in each model. Using grid search, as we did in this experiment, can be costly and time-consuming.

3.1.3 Comparisons across Models

After the search process for a recursive model, we then conduct experiments on several different distributions and sizes datasets. During this process, we use the following settings:

- The $\boldsymbol{X}$ is generated from *uniform*, *normal* and *lognormal* distribution.


Group 1 (Fully Connected Network)
Group 2 (Linear Regression)
Group 3 (Hybrid)
200+2000
200+4000
200+6000
400+2000
400+4000
400+6000
600+2000
600+4000
600+6000
7,487.06
653.85
24,440.76
1.13
13,034.23
196.91
9,208.99
$1.13 \cdot 10^5$
12,695.56
$1.14 \cdot 10^5$
15,434.79
51
9,745.34
$1.13 \cdot 10^5$
20,434.91
19
8,118.02
$1.42 \cdot 10^5$
4,348.46
8,246.63
18,769.81
7,326.24
13,297.72
6,276.09
5,143.06
$1.2 \cdot 10^5$
12,864.81
6,783.43
8,041.42
5,998.72
4,986.93
$1.22 \cdot 10^5$
13,843.61
8,434.09
8,366.57
35,342.64
LR/LR/FCN
LR/FCN/FCN
FCN/FCN/LR
FCN/LR/LR
LR/FCN/LR
FCN/LR/FCN
220.5
14,478.05
9,318.7
13,656.15
602.2
12,191.84
5,229.91
13,403.65
6,588.58
12,567.93
8,292.06
11,572.94
Mean Square Error
Memory Usage (KB)



(a) B-Tree **(b)** FCN **(c)** Recursive Model

Figure 3.2: The relations between the number of data points, average query time and the memory usage among three different indexes. The blue line represents the average query time and the red line represents the memory usage

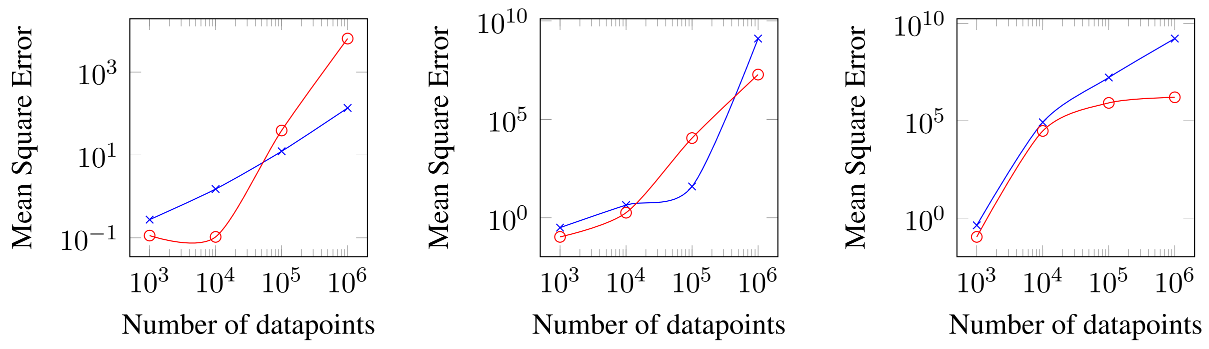
- For each distribution, we generate 1 thousand, 10 thousand, 100 thousand and 1 million data points. We then assign the generated data points into pages where $N_{page} = 10$.
- We use a B-Tree with degree $t = 20$ (i.e., at least 19 keys and at most 39 keys), a fully connected neural network with two layers and 32 nodes per layer, and a recursive model with 200 second-layer models and 4000 third-layer models.

We compare the following performance metrics:

- The query time per key and the memory usage among three index models.
- The mean square error caused by the fully connected network and the recursive model across different distributions.
- The construction time among three index models.

Conclusion 3.1 From Fig. 3.2, we analyse the time complexities for query and the space complexity for storing three different index models.

1. From Fig. 3.2a, we verified that the average query time per key for a B-Tree is growing as the number of data points is increasing. It grows with a complexity of $\mathcal{O}(\log n)$, i.e. it grows slower when there are more data points.
2. From Fig. 3.2b, we found that the memory usage of a fully connected neural network is significantly less than the memory usage of B-Tree. Meanwhile, the fully connected neural network takes constant memory usage, as there is a fixed number of nodes in the neural network. Similarly, the average query time is also a constant in theory. In the experiments, the average query time is changing but likely caused



(a) Uniform **(b)** Normal **(c)** Lognormal

Figure 3.3: The relations between the number of data points and the mean square error in three different distributions. The blue line represents the fully connected network and the red line represents the recursive model

by turbulence.

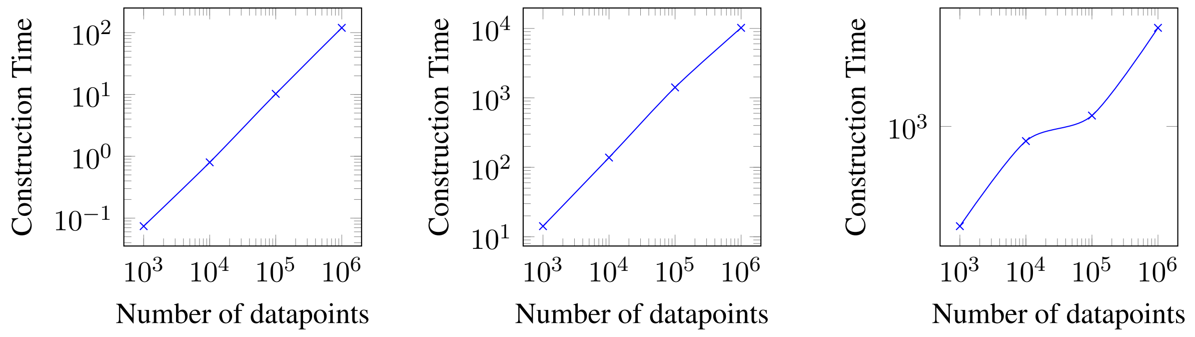
3. From Fig. 3.2c, we found that the memory usage of a recursive model is significantly higher than the memory usage of a fully connected neural network, but still less than a B-Tree, which is because the recursive model consists of thousands fully connected network. The memory usage of a recursive model is fluctuating, as the actually used number of the fully connected network varies. The query time is higher than B-Tree and single fully connected network, but still a constant in theory, as there is only a fixed number of computations.

Conclusion 3.2 From Fig. 3.3, we analyse the errors of fully connected network and recursive model on several different distributed datasets.

1. From Fig. 3.3a, we found that both fully connected neural network and recursive model are capable of modelling uniformly distributed dataset with a rather low error. The fully connected neural network could achieve relatively less error, especially when there is a large amount of data.

2. From Fig. 3.3b, we found that the error is increasing exponentially as the number of data points is increasing. The error in the fully connected neural network is significantly higher than the recursive model.

3. From Fig. 3.3c, we found that the error from the recursive model is significantly less than the error in the fully connected neural network. Combined with 3.3b and 3.3a, we conclude that the recursive model could surpass fully connected network when the data is not uniformly distributed. That means the fully connected network is suitable for uniformly distributed data. We will analyse this property in more



(a) B-Tree **(b)** FCN **(c)** Recursive Model

Figure 3.4: The relations between the number of data points and the construction time among three different index models.

detail in the chapter *Insights and Findings*.

Conclusion 3.3 In 3.4, we analyse the construction time of different models.

1. As shown in 3.4a and 3.4b, the construction time of both B-Tree and fully connected neural network is increasing almost linearly as the number of data points is increasing. Theoretically, the construction time for B-Tree is $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$ for fully connected neural network.

2. In 3.4c, we found that the construction time of recursive model is increasing as well. The time in construction varies by two factors:
   - The number of data points will affect the construction time.
   - As we need to iterate over all possible models in each layer to assign training set, the number of models in each layer will affect the construction time as well.

3.1.4 Comparisons on Large Dataset

The last and largest dataset that we used is a large dataset that contains 190 million key-value pairs that are distributed under lognormal distribution. There are two challenges in this task:

1. The training set is too large to be trained and tuned. As our implementation only supports a single process, it would take a tediously long time to train the recursive model.

2. It takes a super long time (several days in our settings) to evaluate on the very large dataset, as only one CPU thread will be used.

To tackle these challenges, we take the following strategies:

Training on Sampled Dataset We first randomly and uniformed sample from the whole training dataset. By sampling uniformly, we could keep the shape of the distribution unchanged. Assume we sampled S data pairs from the whole training dataset of size N, then we define the sampling ratio as $R = \frac{S}{N}$. We then map the output $\tilde{y}$ from our index model to its approximate position by $\hat{y} = \frac{\tilde{y}}{R}$. We illustrate an example in Example 3.1.

Example 3.1 Assume the **X** in training set is exponentially distributed as

$$[1, 2, 4, 8, 16, 32, 64, 128]$$

and we use a sample size $S = 4$. As we know the size of the fully training dataset is $N = 8$, we have $R = \frac{S}{N} = 0.5$. We uniformly sample from the training set and we will get $[1, 4, 16, 64]$. Then we train an index model based on

$$[(1, 0), (4, 1), (16, 2), (64, 3)]$$

For the key 32 as an example, ideally, we want our index model $\mathcal{F}$ to have an output such that $\tilde{y} = \mathcal{F}(32) = 2.5$. Then the original index of the key 32 can be calculated as $\hat{y} = \frac{\tilde{y}}{R} = \frac{2.5}{0.5} = 5$, which is the index of 32 in the original full training dataset.

Evaluation with Multiple Processes To evaluate our models on the full training set would take several days to complete because there is only one process working on it. As the query is independent of each other, we utilise multiple processes to work on it by taking the following steps:

1. We first train our model on the sampled dataset with $S = 100,000$.
2. Then we split the full training set into 10 pieces such that each piece contains only 19 million pairs.
3. Afterwards, we perform the point query with the trained model on each piece in parallel.
4. Finally, we collect the query time from 10 pieces and sum them to get the total query time of the full training set. Then we divide it by the number of pairs in total, i.e. 19 million and get the average query time per key. For the mean square error, we take the average of errors from each piece.

With these two approaches, we achieved the results as shown below:

[1]The memory usage of each node is slightly larger than previous experiments. It is because the tools for measuring memory usage (`pympler`) requires extra memory, and caused the program to be killed when there is not enough memory. Hence we use a different tool (`top`) to measure an approximate memory usage.

Model	Construction Time (s)	Avery Query Time (ms)	Memory Usage (MB)
B-Tree (degree=20)	26356 (1.00x)	**0.3489 (1.00x)**	96912 [1]
Recursive Model	**334 (0.013x)**	2.6505 (7.60x)	**8.836**

Table 3.1: The construction time, average query time and memory usage of a B-Tree (with a degree=20) and a recursive model.

Conclusion 3.4 From Table 3.1, we have the following conclusions:

1. The construction time of recursive model can be significantly less than the construction of B-Tree, for two reasons:
    - We sample from the training dataset and avoid iterating over all the data points. In contrast, B-Tree has to iterate all the data points and insert them one by one.
    - The recursive model trains relatively fast as it can converge in one to a few passes over the data points.
2. Then average query time for the recursive model is higher than B-Tree, but not significantly higher. The computation costs are mainly on the calculation in fully connected neural networks. The query time for the recursive model can be improved by either using a well-established library, such as PyTorch, that provides faster matrix computation or using faster hardware such as the GPU to improve the query speed.
3. The memory usage of B-Tree is significantly higher than the recursive model. As we showed above, the memory usage of B-Tree is $\mathcal{O}(n)$ and hence growing linearly. For the recursive model, the memory usage mainly depends on how many models in each layer. Therefore, the memory usage of a recursive model has an upper bound (if all models are used), and then will not grow as the number of data points is growing.

3.2 Two Dimensional Data and Indexes

For two dimensional data, the evaluation covers the following tasks:

- Find hyper-parameters for the LISA Baseline model empirically.
- Find hyper-parameters for the LISA model empirically.
- Compares the performance between KD-tree, LISA Baseline and LISA models for the point query.
- Compare the performance between KD-tree, LISA Baseline and LISA models for the range query.

- Compare the performance between KD-tree and LISA models for KNN query. KNN Query has not been implemented for LISA Baseline as there is no description of KNN Query for Baseline model in the paper.

3.2.1 Dataset

For two dimensional case, we manually generate three columns of the data:

- The first two columns contain the 2-dimensional keys $\boldsymbol{X} \in \mathbb{R}^2$, which are independently sampled from a lognormal Distribution. The dataset contains 190 million key-value pairs.
- Then we assign the keys into different pages according to a preset parameter N_{page} for page size. Specifically, the first N_{page} keys will be assigned to the first page, the second N_{page} keys will be assigned into the second page and so on so forth. After the assignments, we set the second column Y to be the page index of the corresponding x.

Our final data-set consists of 190 million key-value pairs that are distributed under lognormal distribution.

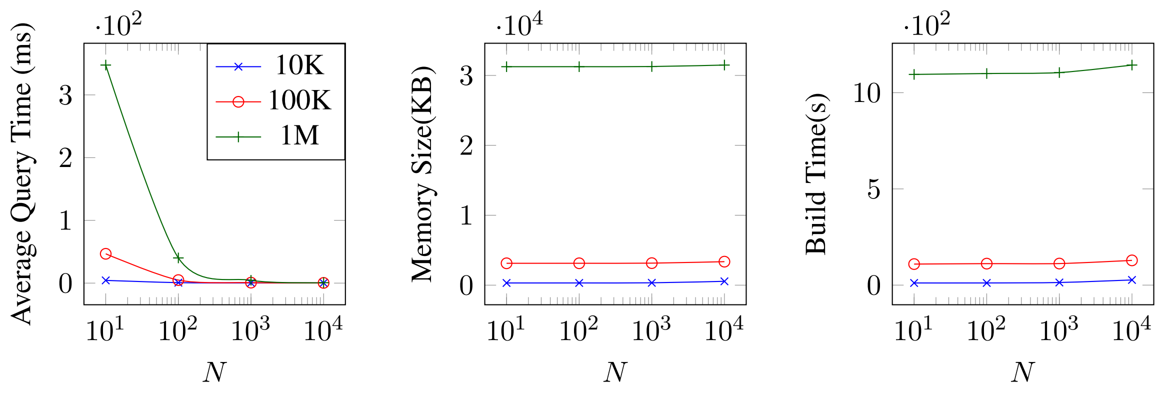
As discussed in previous section, there are multiple challenges in using the complete dataset for training and hyper-parameters tuning. Even on google cloud server, running experiments with the full data take considerable long times (LISA model took 26 hours to build), we had limited cloud server budget and a large number of experiments to run. Therefore, for two dimensional indexes evaluation, we have used sampling to generate smaller training datasets.

3.2.2 Hyper-parameters Search

After generating dataset as mentioned in previous section, we sample a smaller subset from it. We repeat our experiments for 3 different sample sizes of 10000, 100000 and 1000000 points. Test data is a copy of training data for all our experiments. For Baseline and LISA models, final prediction is given by linear search through a range of values (identified as a Cell for Baseline and Shard for LISA model) and mean square error (MSE) is zero as test points are already learned during training. This is where Learned Index models differ from traditional machine learning models where model performance is evaluated on unseen data.

Hyper-parameter search for the LISA baseline

Baseline model has one hyper-parameter: N (Number of cells specifying the number of equal length intervals into which mapped values are divided). The point query search consists of two parts, first is binary search to locate the cell into which the query key is located, followed by sequentially comparison of the query key value with keys in the found cell until a match is found. The time complexity of first search is $log_2 N_1$, where N_1 is the number of cells. The time complexity of second search is $\lceil N_2/2 \rceil$, where N_2 is the number of keys per cell.



(a) Average Query Time (ms) **(b)** Memory Size (KB) **(c)** Build Time (s)

Figure 3.5: Hyper-parameter search in LISA Baseline for training sizes $10K$, $100K$ and $1M$.

Conclusion 3.5 Following conclusions can be drawn from experimental results shown in table A.2 and Fig. 3.5

1. Optimum value of hyper-parameter N will be equal to number of points in the training data-set, resulting in 1 key per cell and search query time of O(log_2N).

2. Average Query Time: Average Query Time decreases with increase in value of N as number of keys per cell decreases.

3. Build time: Build time increases with increase in value of N, as metadata for additional cells needs to be calculated.

4. Memory Size: Memory requirements of the model increases with increase in value of N, as metadata for additional cells needs to be stored. Increase in memory size is not significant with increase in N as we maintain only two values per cell, mapped value of first key in the cell and mapped value of last key in the cell.

Hyper-parameter search for the LISA implementation

For LISA model, we have 3 hyper parameters:

1. G: The size of the grid cell. Number of grid cells into which the key space is divided. In our implementation, we use a square grid cell, and total number of cells is given by $G \times G$.

2. N: Number of equal length intervals into which mapped value range is divided. During our experiments, we found that shard prediction algorithm gives better performance if mapped interval boundaries are aligned to grid cell boundaries. Therefore this parameter is always initialised to N=$G \times G$.

3. S: Number of shards to learn per mapped interval.

Conclusion 3.6 Following conclusions can be drawn from experiments results shown in tables A.4, A.5 and A.6.

1. For a particular value of G, average query time decreases and memory size increases with increase in value of S. This is expected as increasing S, will result in lesser number of keys per shard, thereby reducing the sequential search cost of scanning the query key through the Shard.
2. Average query time decreases and memory size increases with increase in values of G and S.
3. We found emprically that value of S should be choosen such that there are at least 45 keys per shard. We see mean square errors(mse) if number of keys per shard are less than 45 for following reasons.
   a) For point query search, we first predict a shard and then sequentially compare the query point key values with all the keys in the predicted shard until a match is found.
   b) For query points near the shard boundaries, there can be a mismatch in the true index of the shard and predicted index. If the query point is not found in the predicted shard, we continue our search in adjacent left and right shards in an empirically found range.

   During test experiments, we found that if shard size is less than 45 keys, sometimes shard prediction error can be greater than 1 and point query search can fail resulting in MSE errors.

3.2.3 Comparisons across Models

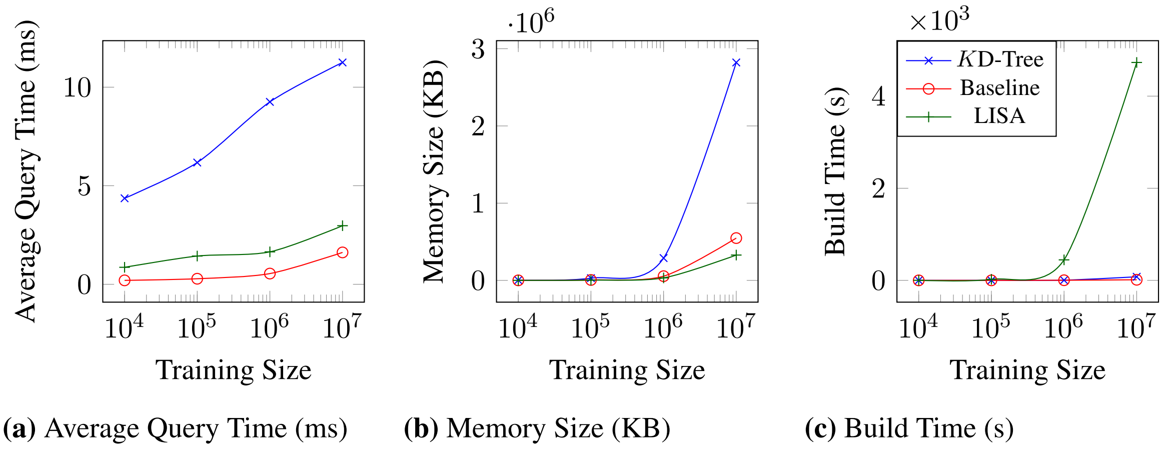
During following experiments, for each training data size, we have used hyper-parameters optimized for that particular data set size.

Point Query Comparison

Table A.7 and Fig. 3.6 shows the performance evaluation for KD-Tree, LISA-Baseline and LISA Models for different training data sizes. For a given training set, we perform point query evaluation for every point in the data-set and take the average.

Conclusion 3.7 The following conclusions can be concluded:

1. LISA outperforms KD-tree in terms of average query time. Search complexity of KD-Tree and LISA baseline(configured to keep 1 key per cell) is $\mathcal{O}(N)$, and $\mathcal{O}(\log_2 N)$ respectively where N is the number of points in the training data-set. On the other hand point query search cost in LISA is a combination of 4 costs.



(a) Average Query Time (ms) **(b)** Memory Size (KB) **(c)** Build Time (s)

Figure 3.6: Point Query experimental results for KD-Tree, Baseline and LISA models.

a) Search cost to find the grid cell to which point query belongs. Search complexity of this cost is $\mathcal{O}(\log_2 N_c)$), where N_c is equal to number of grid cells.

b) Search cost to find the mapped interval to which point query belongs. Search complexity of this cost is $\mathcal{O}(\log_2 U)$), where U is the number of intervals into which sorted mapped array is divided.

c) Find the index of the shard to which point query belongs. Search complexity of this cost is $\mathcal{O}(\infty)$ as shard prediction function weights are already learned during the build process.

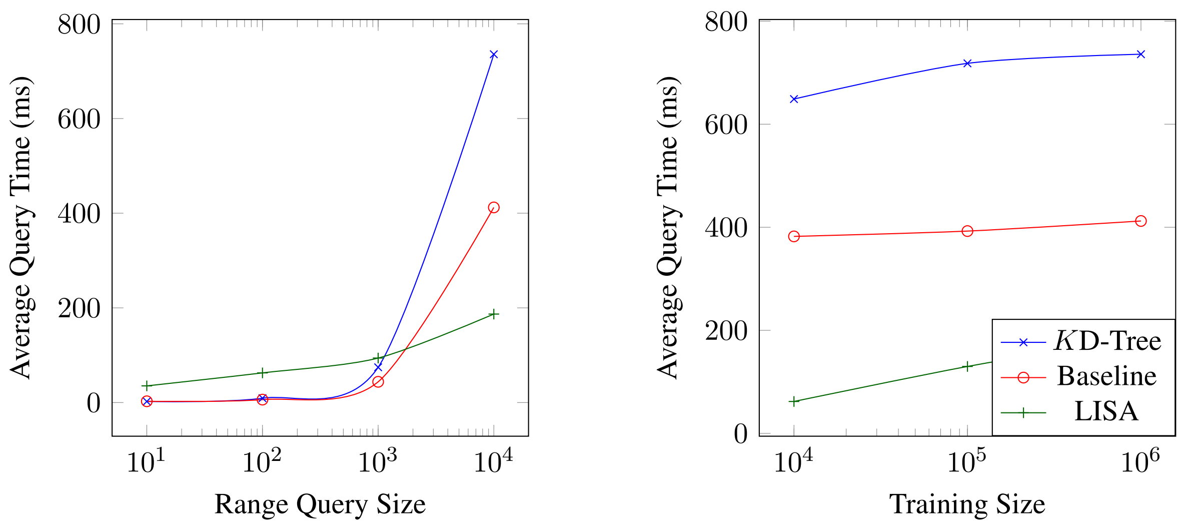
d) Once the index of the shard is found, search sequentially in the shard interval by comparing query point key value with all the keys in the shard until a match is found. Search complexity of this cost is $\mathcal{O}(\log_2 N_k)$), where N_k is equal to number of keys per shard.

2. LISA outperforms KD-tree in terms of memory size requirements. The storage consumption of LISA is considerably smaller than KD-Tree that has to construct a tree with all nodes and entries based on MBRs (minimum bounding rectangle) and parent-children relationships. In contrast, LISA only keeps the parameters of $\mathcal{M}$ and $\mathcal{SP}$. Specifically,$\mathcal{M}$'s parameters contain several numbers and a small list only, and $\mathcal{SP}$ is composed of a series of piecewise linear functions whose parameters are a number of coefficients.

3. LISA's build time is significantly higher than KD-Tree and LISA Baseline. The higher build time is caused by Shard Training Algorithm.

Range Query Experiments

Table A.8 shows evaluation results for LISA,Baseline and KD-tree models for range sizes of 10, 100, 1000 for different training sizes. For a given range query size, we perform 20 trials and take the average. For each trial, we sample a random point from the test set and find the



(a) Range Query Size **(b)** Training Size

Figure 3.7: Range Query experimental results for KD-Tree, Baseline and LISA models

range from sampled point to the range query size.

As shown in the Fig. 3.7, LISA outperforms KD-tree for range query size of 10000 for all training sizes, however its range query time for smaller range sizes is significantly higher than KD-Tree

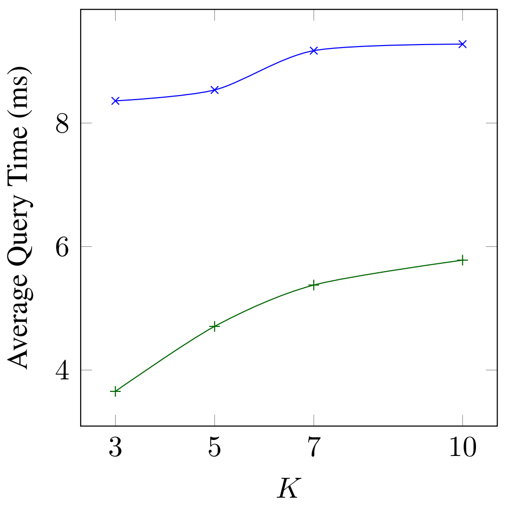
1. Plot A shows average range query time for a fixed training size of $1M$ points. LISA outperforms KD-Tree for larger range queries
2. Plot B shows average range query time for a fixed range query of size 10000 for various training sizes. LISA outperforms KD-Tree for all training data sizes for range queries of size 10000.

KNN Query Experiments

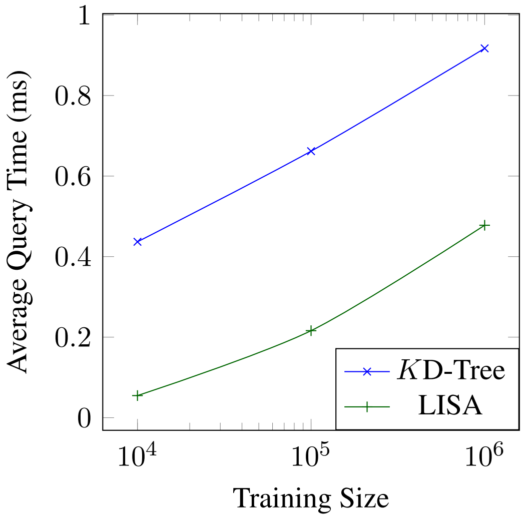
Table A.9 shows evaluation results for LISA and KD-tree models for KNN Queries for various value of K and training sizes. For a given K value, we perform 20 trials and take the average of query time. For each trial, we sample a random point from the test set and find K neighbours around that point.

In Fig. 3.8, we present the comparison of LISA and KD-tree models for KNN Queries.

1. Plot A shows average KNN query time (over 20 trials) for a fixed training size of 1M points and different values of K. LISA outperforms KD-Tree for all values of K.
2. Plot B shows average KNN query time for various training sizes with K = 10. LISA outperforms KD-Tree for all training data sizes.



(a) Number of Nearest Neighbours (K)



(b) Training Size (Fix $K = 10$)

Figure 3.8: KNN Query experimental results for KD-Tree and LISA models

4 Insights and Findings

4.1 General Discussions

Limitations

Though the learned index model, especially the recursive model has a potential to greatly reduce the memory usage and cost less time in making the query. It is still limited in several perspective.

- **Read-only Database**. Current recursive model index assumes that the data is a static, read-only array. Only when this assumption is hold, we can regard the database index as the CDF. However, in reality, we usually need to insert and delete the data in the array and violates this assumption.
- **Sorted Keys**. The recursive model and baseline model assume that the keys are sorted in ascending order, so that the CDF assumption applies.
- **In-Memory Database**. In our implementations, we only consider the case where all the keys are stored in the memory.

To apply the learned indexes into a general-purpose database, we will need to overcome these limitations. For example, the model needs to be trained again in order to support the read-and-write database.

4.2 One Dimensional Learned Index

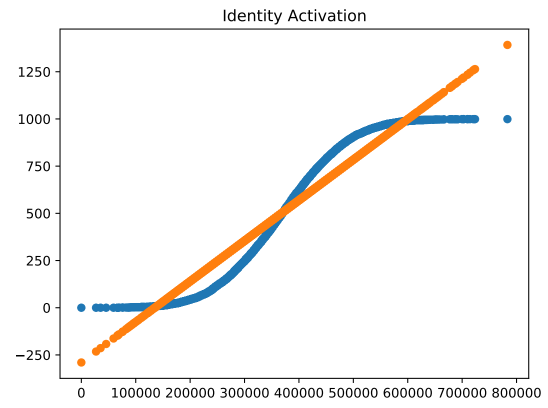
4.2.1 Baseline Learned Index

Activation Functions

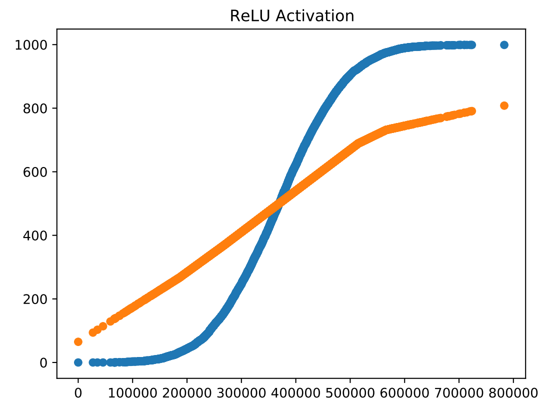
From our observations, activation functions determines the shape of the fully connected neural network. With the one-dimensional data, the input and output of a neural network is always a scalar, which reveals interesting relations between the activation function and the output of neural network. We use two different activations functions to describe this relation.

- If we use identity activation function, i.e.$z^{(i)}(x) = x$, then no matter how many layers are there, the fully connected neural network falls back to a linear regression.

  Proof: The output of the first layer, with identity activation function, will be $o^{(1)} = z^{(1)}(w^{(1)}x + b^{(1)}) = w^{(1)}x + b^{(1)}$. Then the output will be the input of the next layer,



(a) Identity Activation



(b) ReLU Activation

Figure 4.1: The predictions of neural networks with different activation functions. The blue line represents the ground truth and the orange line represents the predicted output.

and hence the output of the second layer will be $o^{(2)} = z^{(2)}(w^{(2)}(w^{(1)}x + b^{(1)}) + b^{(2)}) = w^{(2)}w^{(1)}x + w^{(2)}b^{(1)} + b^{(2)}$. Hence if we use identity activation, the trained neural network will become a linear regression. The predicted output of a neural network with identity activation is illustrated in Fig. 4.1a, where we could verify that the predicted output is a line.

- With ReLU (Rectified Linear Unit) as activation function, i.e. $z^{(i)}(x) = \max(0, x)$, then the fully connected neural network becomes a piecewise linear function.

  Proof: In the first layer, the output of a neural network with ReLU activation will be $o^{(1)} = z^{(1)}(w^{(1)}x + b^{(1)}) = \max(w^{(1)}x + b^{(1)}, 0)$. There will be two cases for this function:

$$o^{(1)} = \begin{cases} w^{(1)}x + b^{(1)} & w^{(1)}x + b^{(1)} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

  The output of the first layer will be the input of the second layer, which uses the same ReLU activation function. Hence the output will be $o^{(2)} = z^{(2)}(w^{(2)}o^{(1)} + b^{(2)}) = z^{(2)}(w^{(2)}\max(w^{(1)}x + b^{(1)}, 0) + b^{(2)})$. If $w^{(2)}(\max(w^{(1)}x + b^{(1)}, 0)) + b^{(2)} < 0$, then the output will be 0. Otherwise, as the $o^{(1)}$ is a vector, we have to perform the multiplication element by element. Assume that $o^{(1)} = [\max(w_1^{(1)}x + b_1^{(1)}, 0), \cdots, \max(w_n^{(1)}x + b_n^{(1)}, 0)]$. By multiplying with $w^{(2)}$, we get

$$o^{(2)} = w_1^{(2)}\max(w_1^{(1)}x + b_1^{(1)}, 0) + \cdots + w_n^{(2)}\max(w_n^{(1)}x + b_n^{(1)}, 0) + b_2$$

  For each term in the equation above, there are two possibilities: 0 when $w_i^{(1)}x + b_1 < 0$ or $w_i^{(2)}w_i^{(1)}x + w_i^{(2)}b_i^{(1)}$. Without loss of generality, we consider a case where all of them are non-zero. In this case, we will have

$$o^{(2)} = \sum_i w_i^{(2)} w_i^{(1)} x + \sum_i w_i^{(2)} b_i^{(1)} + b_2$$

From the above induction, we find out that it is still a linear function when all values are positive. If there are some values to be zeros, then the function is still a linear function, but with different slope and intercept. Hence we conclude that the neural networks with ReLU activation function will become a piecewise linear function.

4.3 Two Dimensional Learned Index

Limitation of LISA Baseline model

Prediction cost in baseline method consists of following two parts.

1. Search cost for the cell which contains the key. This cost will be equal to $log_2 N_1$, where N_1 is the number of cells into which mapped values are divided.
2. Cost associated with sequentially comparing the query point key value against keys inside the cell found in previous search. On average this cost will be equal to $N_2/2$, where N_2 is the number of keys in a cell.

   If cell size is large, number of cells will be smaller, number of keys per cell will be higher, resulting in higher cost of sequential scan with in the cell.

Consider the example in Fig. 4.2. Dataset is divided into 3 sections based on the mapped values. Any point or range query in the second triangle(page) will result into a sequential scan through all 9 keys in the cells.

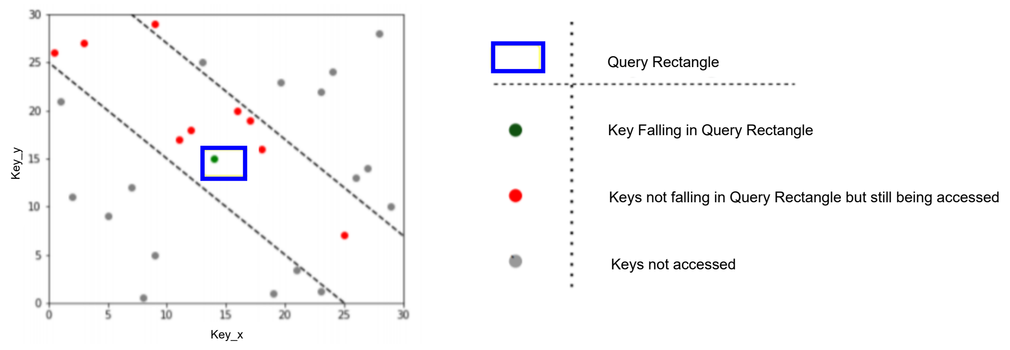
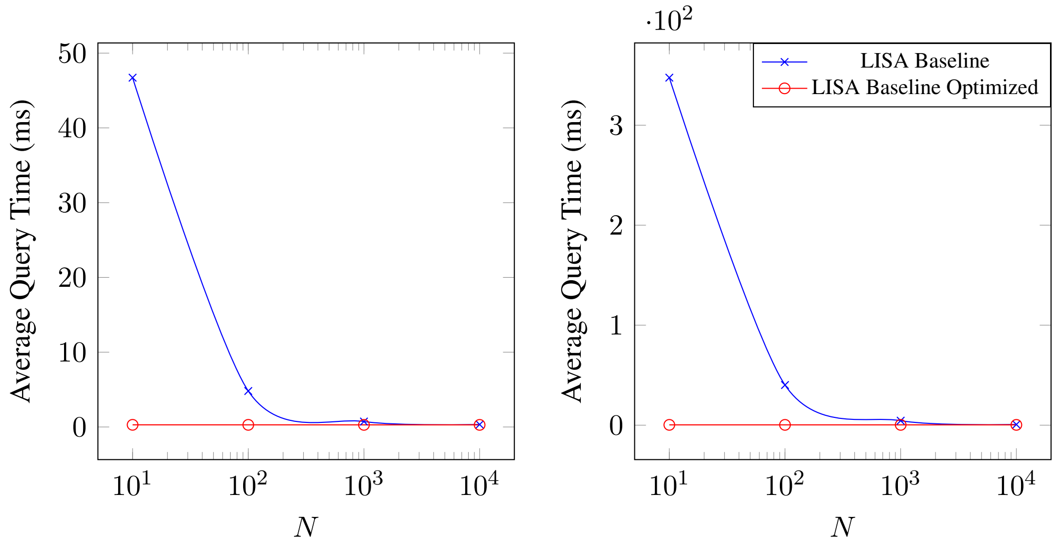


Figure 4.2: Baseline Method Limitation



(a) Training Size 100K **(b)** Training Size 1M

Figure 4.3: Point query results comparison between LISA Baseline and Optimized Model for different training sizes.

LISA Baseline model search optimization for smaller values of N

In case of high dimensional key values, key with in a cell can not be searched with mapped value, as a large number of keys can have the same mapped value. However for the 2 dimensional scenario, we can get considerable savings in search cost by replacing sequential scan based on keys values to binary search based on mapped value. As in the original method, search process will consist of two parts.

1. Find the cell which contains the query key based on mapped value using binary search.
2. With in the cell, replace sequential search based on query key value with the binary search based on query key mapped value. Once mapped value is found, do a lookup in the neighbourhood of the found key based on query key 2 dimensional value.

As shown in Fig. 4.3, we get significant savings in the query time with this approach for smaller values of N. As the value of N increases, number of Keys per cell decreases, and savings in avoiding sequential search gets normalized.

4.4 Future Work

Future work

In current work, we have implemented RMI and LISA, two novel learned index structures for one and two dimensional data respectively. This work opens up several directions for future research on learned indexes for database systems. We are listing some of them here.

1. Read only and in-memory database are two major constraints applicable to our LISA implementation that are supported by the original paper. Adding support for insertion, deletion and disk resident training data can be taken in next phase for both one dimensional and spatial databases.
2. LISA paper suggests Lattice Regression model to learn an appropriate distance bound from underlying training data for every query point and specific value of K. This distance bound is used to convert the KNN query to range query. It will be interesting to try different learning models like Lattice Regression, Neural Network and Bayesian Neural Network to learn this distance bound from underlying data.
3. It will be interesting to study other query types (e.g., spatial joins and closest pairs) using LISA
4. Our results show that learned index models outperform traditional databases by utilizing the distribution of data being indexed.. It will be interesting to develop functional databases using learned index models and investigate their performance on real data

Conclusion

In this work, we have implemented RMI and LISA, two novel learned index structures and B-Tree and KD-Tree two traditional database indexes for one and two dimensional data respectively. We have conducted a number of experiments using real and synthetic datasets. The experimental results demonstrate that learned index models outperforms traditional indexes in terms of storage and IO costs for point, range and KNN queries. Some of our learnings are listed below:

1. The key idea in our work has been to map the key space into a sorted one dimensional array and use learned models to approximate the cumulative distribution function (CDF).While RMI uses a hierarchy of linear regression models, LISA makes use of piecewise linear models to learn the cdf.

5 Conclusion

In this project, we reviewed and implemented two classic tree structures, B-Tree and KD-Tree, used as database indexes. The tree structures are capable of finding elements precisely as they will traverse all possible nodes. The shortcomings of these tree structures also come from this property: the tree needs to save and traverse the possible nodes, which yields a space complexity that is proportional to the number of records. In the meanwhile, it yields a query time complexity that has a positive correlation with the number of records. As the volume of data is increasing rapidly, the time and space complexity becomes huge and the tree structures become a bottleneck of applications.

We then implement two kinds of learned indexes: the recursive model index for one-dimensional data and the LISA model for two-dimensional data. We conclude that the recursive model and its baseline model have bounded time and space complexities.

5.1 Use Cases of Learned Index

From the aspect of the data volume, learned index is suitable for very large dataset. As we have seen, the most promising advantage of learned index is its constant time and space complexity, it could achieve less query time and occupies significantly less memory space when the data size is huge.

From the aspect of the data distribution, learned index is suitable for dataset that we have some prior knowledge about its distribution. Ideally, if we know the exact distribution of the data (e.g. normal, uniform or lognormal distribution), then we can use maximum likelihood estimation to fit the dataset use the estimator as an index.

However, in most cases, the data is not distributed in a certain distribution, i.e. we cannot use a single formula to fit the dataset. In this case, we can use the recursive model to split the dataset into smaller pieces and apply different models in different pieces.

From the aspect of applications, the learned indexes implemented in this project do not support write operation. Even though we can construct the model again once there is a new data point coming, it would take a much longer time to complete the write operation compared with classic tree-based indexes. Therefore, the learned indexes are not suitable for write-intensive applications. From the database perspective, learned indexes are more suitable for online analytical processing (OLAP) but not online transactional processing (OLTP).

5.2 Shortcomings of Learned Index

Having said these advantages of learned indexes, they all have their shortcomings.

1. Even though the learned indexes have a constant time complexity for queries, the constant is relatively large. Therefore, if the number of records is not huge, the learned indexes will not outperform classic tree structures.

2. The recursive model, baseline model and other models are prone to error. It may not be a big issue with the in-memory databases, but it will cost much more time when searching between different disk pages is needed, especially with the traditional hard disk drive (HDD).

Acknowledgement

We would like to express our sincere gratitude to Prof. Dr. Michael Böhlen, and Mr. Qing Chen for their commitment in supervising this project. Our appreciation extends to Dr. Sven Helmer in reading our report and arranging discussion and presentation of this project. Beyond them, there are lots of interesting discussions during our presentation of this project at the database technology group and we would like to thank all of them for their fruitful input.

Appendices

A Appendix

# id	Distributions	root model	second model	third models	Build Time (s)	Query Time (ms)	Evaluation Error (MSE)	Memory Size (KB)
1	log_normal	fcn	200 fcn	2000 fcn	418.9493798	0.970932583	653.8536667	7487.059896
2		fcn	200 fcn	4000 fcn	1141.521194	0.9675528	**1.134166667**	24440.75523
3		fcn	200 fcn	6000 fcn	688.8004486	1.07512705	196.9116667	13034.22656
4		fcn	400 fcn	2000 fcn	483.1734781	1.158343717	113246.196	9208.992183
5		fcn	400 fcn	4000 fcn	636.8463397	1.339095933	113652.3212	12695.55731
6		fcn	400 fcn	6000 fcn	742.0712694	1.243333667	51.00183333	15434.78905
7		fcn	600 fcn	2000 fcn	504.959355	1.06512235	113246.2647	9745.335942
8		fcn	600 fcn	4000 fcn	879.6010201	0.973031833	18.99766667	20434.90626
9		fcn	600 fcn	6000 fcn	373.6126809	1.11725315	142041.6877	8118.023442
10		lr	200 lr	2000 lr	262.5089284	1.280502367	8246.633985	4348.463542
11		lr	200 lr	4000 lr	869.7494701	1.304096217	7326.238372	18769.81252
12		lr	200 lr	6000 lr	655.0431077	1.318176683	6276.09111	13297.72135
13		lr	400 lr	2000 lr	275.3925674	1.31789575	120427.9247	5143.059892
14		lr	400 lr	4000 lr	601.7362665	1.453903583	6783.428749	12864.80731
15		lr	400 lr	6000 lr	388.5866734	1.623972083	5998.720313	8041.416654
16		lr	600 lr	2000 lr	267.8966881	1.861582733	121932.5051	4986.927088
17		lr	600 lr	4000 lr	558.531068	1.52717965	8434.091306	13843.60678
18		lr	600 lr	6000 lr	337.0881814	1.28034995	35342.6365	8366.570317
19		lr	200 lr	4000 fcn	34.86059083	1.63086885	14478.05283	**220.4973958**
20		lr	200 fcn	4000 fcn	410.2378013	1.653916983	13656.1507	9318.697933
21		fcn	200 fcn	4000 lr	38.131223	1.667702583	12191.8397	602.2005208
22		fcn	200 lr	4000 lr	238.9569197	1.663567483	13403.64758	5229.914058
23		lr	200 fcn	4000 lr	290.6430138	1.657428583	12567.93278	6588.58335
24		fcn	200 lr	4000 fcn	352.6849412	1.909310833	11572.93555	8292.059883

Table A.1

Training/Test Data Size	Model	N	Build Time (ms)	Avg Query Time (ms)	Memory Size (KB)
10,000	LISA Baseline	10	11.17	4.3426	313
10,000	LISA Baseline	100	11.25	0.7189	315
10,000	LISA Baseline	1000	13.54	0.3283	336
10,000	LISA Baseline	10000	26.83	0.2415	547
100,000	LISA Baseline	10	109.28	46.7173	3126
100,000	LISA Baseline	100	111.59	4.8086	3128
100,000	LISA Baseline	1000	111.97	0.7271	3149
100,000	LISA Baseline	10000	128.49	0.3301	3360
100,000	LISA Baseline	100000	272.93	0.2381	5469
1,000,000	LISA Baseline	10	1094.85	347.5613	31251
1,000,000	LISA Baseline	100	1099.38	40.1451	31253
1,000,000	LISA Baseline	1000	1104.65	4.4732	31274
1,000,000	LISA Baseline	10000	1143.65	0.6697	31485
1,000,000	LISA Baseline	100000	1273.56	0.2944	33594
1,000,000	LISA Baseline	1000000	2717.65	0.2436	54688

Table A.2: Hyper-parameters Search LISA Baseline Model for training sizes $10K$, $100K$ and $1M$

Training/Test Data Size	Model	N	Build Time(ms)	Avg Query Time(ms)	Memory Size(KB)
10,000	LISA Baseline Optimized	10	11.1208	0.2841	313
10,000	LISA Baseline Optimized	100	12.0108	0.2779	315
10,000	LISA Baseline Optimized	1000	12.7589	0.2765	336
10,000	LISA Baseline Optimized	10000	25.8732	0.2752	547
100,000	LISA Baseline Optimized	10	112.973	0.2855	3126
100,000	LISA Baseline Optimized	100	114.318	0.2823	3128
100,000	LISA Baseline Optimized	1000	116.699	0.2806	3149
100,000	LISA Baseline Optimized	10000	129.514	0.2794	3360
1,000,000	LISA Baseline Optimized	10	1116.51	0.2905	31251
1,000,000	LISA Baseline Optimized	100	1118.85	0.2858	31253
1,000,000	LISA Baseline Optimized	1000	1134.88	0.2844	31274
1,000,000	LISA Baseline Optimized	10000	1134.88	0.2831	31485

Table A.3: Experimental results for LISA Baseline model with search optimization

Training/Test Data Size	Model	G	S	Build Time(s)	Avg Query Time(ms)	Memory Size(KB)	mse
10,000	LISA	4*4=16	5	4.335	1.13135	324.72	0
10,000	LISA	4*4=16	10	3.370	0.96036	329.07	0
10,000	LISA	4*4=16	20	1.127	0.86184	337.85	0
10,000	LISA	4*4=16	30	3.478	0.74339	346.63	5729

Table A.4: Hyper-parameters Search LISA Model: Training Size:10,000 Points.
a) For the last row, Numbers of keys= 10000
b) Keys per cell= $10000 \setminus (4 \times 4) = 625$
c) Keys per shard = $625 \setminus 30 = 20$ keys per shard, resulting in mse errors

Training/Test Data Size	Model	GridCellSize	No of Shards	Build Time(s)	Avg Query Time(ms)	Memory Size(KB)	mse
100,000	LISA	4*4=16	50	122.64	1.51173	3176.6	0
100,000	LISA	4*4=16	100	30.211	1.44084	3220.3	0
100,000	LISA	4*4=16	150	142.13	1.15491	3264.1	297234
100,000	LISA	6*6=36	50	66.375	1.55903	3238.1	0
100,000	LISA	6*6=36	75	72.491	1.43043	3287.2	0
100,000	LISA	6*6=36	100	60.929	1.64881	3336.4	5.6e+07
100,000	LISA	8*8=64	20	35.638	1.54029	3218.7	0
100,000	LISA	8*8=64	50	45.014	1.52117	3323.6	0

Table A.5: Hyper-parameters Search LISA Model: Training Size:100,000 Points

Training/Test Data Size	Model	GridCellSize	No of Shards	Build Time(s)	Avg Query Time(ms)	Memory Size(KB)	mse
1,000,000	LISA	10*10=100	50	743.29	1.77751	31558.9	0
1,000,000	LISA	10*10=100	100	1077.89	1.63397	31832.3	0
1,000,000	LISA	20*20=400	25	365.49	2.53317	31930.8	0
1,000,000	LISA	20*20=400	50	609.32	1.44526	32477.6	0
1,000,000	LISA	25*25=625	25	240.22	1.56227	32779.8	0
1,000,000	LISA	30*30=900	25	205.18	1.79839	33010.3	0

Table A.6: Hyper-parameters Search LISA Model: Training Size:1,000,000 Points

Training/Test Data Size	Model	Build Time (s)	Avg Query Time (ms)	Memory Size (KB)
10,000	KD-Tree	0.023	4.363	2890
10,000	Baseline	0.026	0.198	547
10,000	LISA	1.127	0.861	337
100,000	KD-Tree	0.340	6.176	28906
100,000	Baseline	0.324	0.241	5469
100,000	LISA	22.491	1.43	3169
1,000,000	KD-Tree	4.124	9.254	289062
1,000,000	Baseline	2.718	0.343	54688
1,000,000	LISA	445.324	1.445	32477

Table A.7: Point Query experimental results for KDTree, Baseline and LISA models

Training/Test Data Size	Range Query Size	Avg Query Time (ms) (KD-tree)	Avg Query Time (ms) (Baseline)	Avg Query Time (ms) (LISA)
10,000	10	1.361	1.113	8.204
10,000	100	5.331	4.518	12.01
10,000	1000	43.88	39.92	29.45
10,000	10000	648.8	382.3	61.94
100,000	10	1.392	1.298	27.92
100,000	100	5.392	5.055	28.96
100,000	1000	43.97	42.83	56.38
100,000	10000	718.1	392.6	129.9
1,000,000	10	2.238	2.661	35.18
1,000,000	100	9.222	6.174	62.63
1,000,000	1000	74.46	43.78	93.97
1,000,000	10000	735.8	412.2	186.9

Table A.8: Range Query experimental results for KD-tree, Baseline and LISA models

Training/Test Data Size	K	Avg Query Time(ms)(KD-tree)	Avg Query Time(ms)(LISA)
10,000	3	4.2207	0.6020
10,000	5	4.3765	0.6084
10,000	7	4.4331	0.6129
10,000	10	4.3682	0.6493
100,000	3	6.0021	1.7601
100,000	5	6.0975	1.7745
100,000	7	6.1684	1.9453
100,000	10	6.6183	2.1617
1,000,000	3	8.3595	3.6549
1,000,000	5	8.5362	4.7073
1,000,000	7	9.2785	5.3767
1,000,000	10	9.1726	5.7799

Table A.9: KNN Query experimental results for KD-tree and LISA model

Bibliography

[1] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[2] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2119–2133, 2020.