Institut für Informatik, Universität Zürich

MSc Project Report

Implementing Learned Indexes on 1 and 2 Dimensional Data

Neeraj Kumar, Nivedita Nivedita, Xiaozhe Yao

Matrikelnummer: 19-759-570

Email: xiaozhe.yao@uzh.ch

April 20, 2021

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

University of Zurich[UZH]

Department of Informatics

(This page intentionally left blank)

Databases use indexes to find records efficiently. Among these indexes, B-tree and KD-tree are the two indexes used for 1-dimensional and 2-dimensional data. In this project, we first implement these two indexes from scratch and then we implemented the learned indexes, including a fully connected neural network and a recursive model for 1-dimensional data [KBC+18] and the LISA model for 2-dimensional data [LLZ+20]. Afterwards, we conduct several experiments to evaluate the performance of learned indexes. In addition to the implementation and evaluation, we then theoretically analyse some properties that the learned indexes hold. Beyond that, we also explore and discuss the properties that the learned indexes should hold.

As an extension to the existing learned indexes, we explore the possibilities of using convolution operation and convolutional neural network as learned indexes.

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, researchers proposed learned indexes [KBC+18], 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 hand-engineered indexes and the learned index. After that, we explore the possibilities of using complex convolutional neural networks as database indexes. This report is organised into the following chapters:

1. **Introduction**. In this chapter, we illustrate the organisation of this report. Besides, we go through the modern computer systems and introduce the general information about database indexes.

2. **Implementation**. In this chapter, we thoroughly describe the implementation of one and two dimensional indexes, including B-Tree, baseline learned index, recursive model, KD-Tree and LISA.

3. **Evaluation**. In this chapter, we perform evaluation among the indexes we implemented with different evaluation dataset.

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

5. **Convolution and CNN for Learned Indexes**. In this chapter we explore the possibilities of using convolution operation and convolutional neural network to build learned indexes.

6. **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 $\mathbf{X}$
(x, y)	A tuple 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
$\mathcal{O}$	Big-O notation for complexity
$\mathcal{SP}$	Shard Prediction Function
$\mathcal{Q}$	Range Query
$\mathcal{K}$	KNN Query
Others	
♣	End of Example
■	End of Proof
□	End of Conclusion

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

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 records, we only need to learn such CDF and we called the learned CDF function as **learned index model**.

Example 1.1 From the perspective of the distribution of data records, our previous example can be rephrased as following. Our data records are (X, Y) pairs 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 records as the only parameter. Compared with B-Tree and other indexes, the advantages are enormous. ♣

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

1. All data records are stored statically in database, 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 records 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 records 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

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 and search processes of B-Trees and then analyse their properties.

Attributes

Each node in a B-Tree has the following attributes:

- $x.n$ is the number of keys currently stored in the node x.
- Inside each node, the keys are sorted in non decreasing order, so that we have $x.keys_1 \leq x.keys_2 \leq \cdots \leq x.keys_{x.n}$.
- $x.leaf$, a Boolean value determines if current node is a leaf node.

With these properties, A B-Tree T whose root is $T.root$ have the following properties:

- Each internal node x contains $x.n+1$ children. We assume the children are $x.c_1, \cdots, x.c_{x.n+1}$.
- The nodes in the tree have lower and upper bounds on the number of keys that can contain. These bounds can be expressed in terms of a fixed integer t.

When inserting keys into a binary search tree, we search for the leaf position at which to insert the new key. However, with B-Tree, we cannot simply find the position, create a new node and insert the value because the tree will be imbalanced again. Hence, in this section, we illustrate an operation that splits a full node around its median key

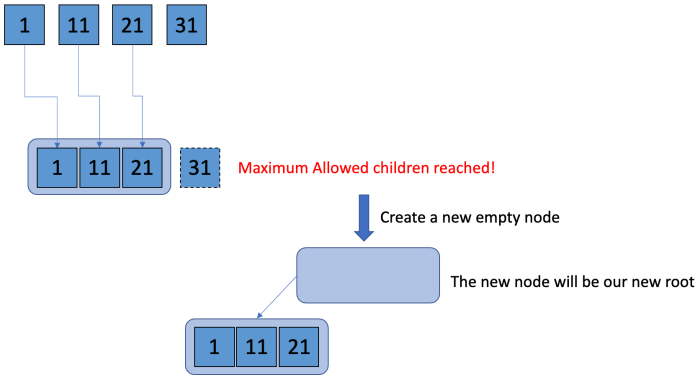


Figure 2.1.: B-Tree key insertion

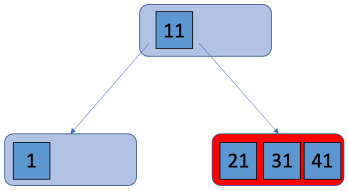


Figure 2.2.: B-Tree key insertion

Insertion in a B-Tree

Algorithm 1: Algorithm for B-Tree insertion

Input: m:`order_of_tree` ,(k, v):`(key, value)`, N:`Node;`
Output: `B-Tree`

if *N is a leaf and not yet full* **then**
  `insert` (k, v) `into` N
else
  `create new Node N'`
  `Find the median of the node`
  `Add the value at the median location to the new node N'`
end
if *N is root with no children and not full* **then**
  `insert` (k, v) `into` N
end

There are two conditions in insertion:

- **When the node is empty or not created at all**: In the algorithm, initially when the first key is inserted and there is no root to the tree it will check the condition if there are any nodes and if not create a new one. It will insert the new key in the node and keeps inserting the new keys until it is one less than the order of the B-Tree being created.
- **When the node is full**: As soon as the maximum number of allowed children has reached for the root node a new empty node is created. Suppose we have inserted

[1,11,21] to a node of a B-Tree with degree 4. Now if we want to insert a new value 31 into the tree, since it has reached the maximum number of children it will find the median of the existing node [1,11,21] which is 11 and increase the level of the B-tree to 2 and make [1] and [21] child nodes of [11]. So now if a new value is to be inserted it can be inserted. Splitting of the node happens each time it reaches it's maximum allowed keys. Now 31 will be compared with 11 and since 31 > 11 it will be inserted in the right child and it will have an updated value of [21,31]. More keys can then be inserted until it reaches it's maximum and splits again. Once the node is split it's parent is also updated to the median value i.e., [11] in this case for nodes [1] and [21,31,41] as can be seen in 2.2.

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 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 2.3.

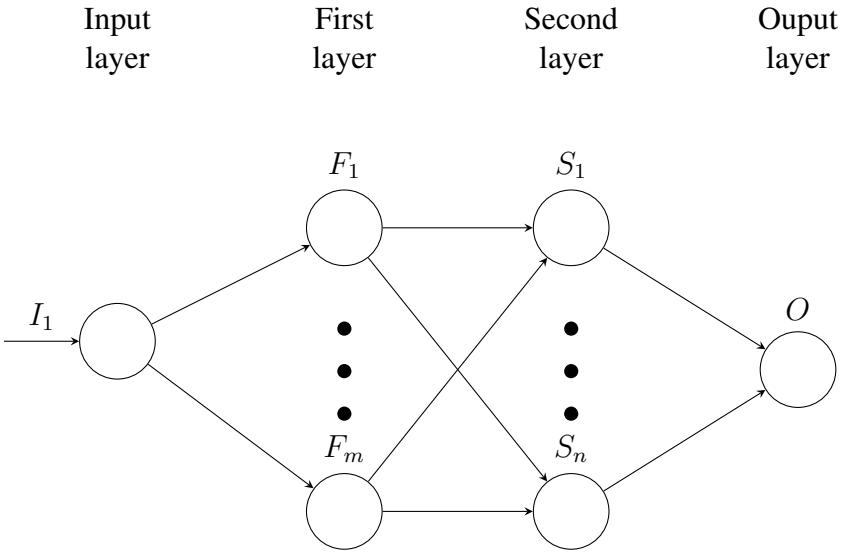


Figure 2.3.: The architecture of the fully connected neural network used as baseline learned index. In this neural network, we use only 2 fully connected layers. The input of this neural network is only one neuron such that it represents the given query key. The output of this neural network is limited to 1 neuron such that it represents the predicted proportional position of the key-value pair.

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 $\textit{ReLU}(F_i)$ 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

In our baseline models, it is not very difficult to reduce the mean square error from millions to thousands. However, it is much harder to reduce it from thousands to tens. This is the so called last-mile problem.

In order to solve this problem, recursive model index was proposed [KBC+18]. The idea is to split the whole set of data into smaller pieces and assign each piece an index model. By doing so, each model is only responsible for a small range of keys. 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 traditional B-Tree model.

As shown in Fig. 2.4. A recursive model can be regarded as 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. ♣

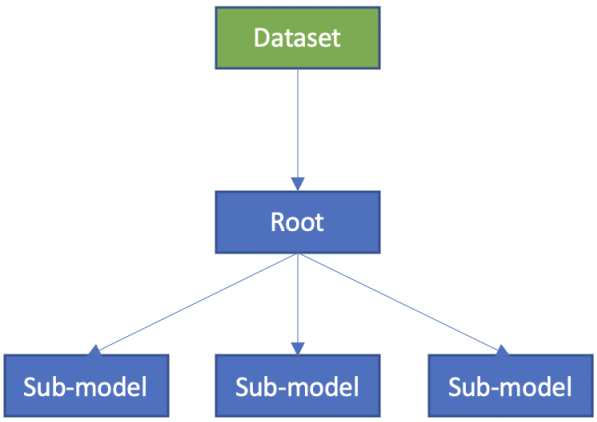


Figure 2.4.: An example recursive model index with one root model and three leaf model.

Properties

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

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 $f : \mathbb{R} \to \mathbb{R}, x \to y$ 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.

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

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, 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. $N_M^{(i+1)}$ 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 id in next stage by `next` $= y_0 * N_M^{(i+1)}/$`max(Y)`.

Algorithm 2: Training of Recursive Model Index

```
   input: num_of_stages; num_of_models; types_of_models; x; y
 1 trainset=[[(x,y)]]
 2 stage← 0
 3 while stage <num_of_stages do
 4 |   while model <num_of_models[stage] do
 5 |   |   model.train(trainset[stage][model])
 6 |   |   models[stage].append(model)
 7 |   end
 8 |   if not last stage then
 9 |   |   for i ← 0 to len(x) do
10 |   |   |   model=models[output from previous stage]
11 |   |   |   output=model.predict(x[i])
12 |   |   |   next=output * num_of_models[stage+1]/max_y
13 |   |   |   trainset[stage+1][next].add((x[i],y[i]))
14 |   |   end
15 end
```

Polynomial Internal Models

In the recursive model index, we use internal models to learn the CDF of a part of the full training data. In order to learn the CDF, we need to know or assume the distribution of a specific part of the data. In this report, we support the following distributions.

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

Here we describe how we 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 $Y = \boldsymbol{X}\boldsymbol{\beta}$.

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

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

Here we know that $(\boldsymbol{\beta}^T\boldsymbol{X}^T\boldsymbol{y})^T = \boldsymbol{y}^T\boldsymbol{X}\boldsymbol{\beta}$ 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

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

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

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

■

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. We have fixed our dimensions of data points to 2-dimensions and their values are stored as a scalar. In our implementation, KD-Tree is a binary tree with every node having data points partitioned in 2-dimensional space. A KD-Tree is a binary tree since each node has two children and when the keys are multidimensional each node is partitioned at split axis based on the depth of the tree and the dimension of the key which in turn determines the number of split axis the tree will have. For example for a 3-dimensional space the first split axis is x-coordinate and then at the next level it will be y-coordinate and lastly z-coordinate.

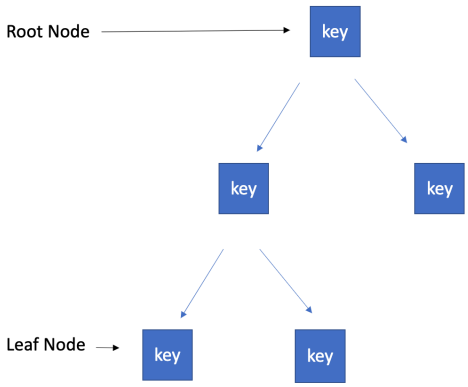


Figure 2.5.: KD-Tree

Properties

1. **Node**: Node of a KD-Tree is essentially a 2-dimensional data point. It can be k-dimensional for KD-Tree. Node is called a Root Node when it is at the top level as can be seen in Fig. 2.5 and does not have a parent. A node is called a Leaf Node when it has no children and is at the bottom level.

2. **Levels**: Nodes are at the same level when they have the same distance from root. Hence, a level increases as the distance of the nodes increase from root. In the 2.5 we can see the key at root is at level 1 and then we have level 2 and level 3.

3. **Dimensions**: Every KD-Tree can be structured in such a way that the data points are divided into k-dimensions. Each node is recursively cut into as many dimensions as is mentioned in the dimensions. In our case the data points are 2-dimensional and hence the space is divided into 2-dimensions alternatively until a leaf is reached.

Algorithm 3: Construction Algorithm for KD-Tree

Input : pointList; Keys=[$(x, y); x \in \mathbb{R}; y \in \mathbb{R}$], dimension = 2, split_axis; 0 or 1
Output: KD-Tree

```
for i ← 0 to len(pointList) do
    int split_axis := split_axis mod dimension // Select the split_axis
     based on depth
    Sort point list and choose median
    Create node at median
    node.leftChild := kdtree(points in pointList before
     median, split_axis+1); //SubTree Creation
    node.rightChild := kdtree(points in pointList after
     median, split_axis+1);
    return node
end
```

For constructing the KD-Tree we have the following constraints:

1. As one moves down the tree, one cycles through the axes used to select the splitting planes. For example, in a 2-dimensional plane we split the data on x-axis at the root and then split it on y-axis for it's children. We then split the grandchildren on x-axis again and so on.

2. Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. This ensures that the tree is balanced. A balanced tree is the one in which leaf node is approximately the same distance from the root.

In the algorithm above, we first sort all the values obtained in the Point list. We initialise the first split axis to 0 and then toggle between 0 and 1 as we increase the depth. Once the data points are sorted, their median is chosen.

Example 2.1 For example if we have points sorted as
$((1,2),(3,4),(5,6),(7,8),(9,10)$
we will take the median as $(5,6)$ and assign $(3,4)$ to the left and $(7,8)$ to the right. The reason node.leftChild = $(3,4)$ is because $3 < 5$ and node.rightChild = $(7,8)$ is because $7 > 5$ i.e., we compare the x-axis of the points to the root when we create the subsequent nodes. Now if we want to add point $[1,2]$ to the above tree it will check the x-coordinate at the root and since $1 < 5$ it will go the left. In the left since we already have a node $(3,4)$ it will check the y-coordinate of the points i.e., $4 > 2$ and so it will go and check in the left of the node if this point can be added there and since it's empty it is added in the left. Further points are added in the same way by checking the x-coordinate and y-coordinate alternatively. This process is then carried out recursively and subtrees are created on the left and right until all the points are added to the tree. Similarly point $(9,10)$ is added in the right subtree as the rightChild of node $(7,8)$. $(None)$ is added when the rightChild or leftChild of the node is empty. The tree will now look like

$(((1,2),(None),(3,4)),((None),(9,10),(7,8)),(5,6))$

Points are added to left and right subtree as follows:
left subtree points: $((1,2)(3,4))$,
right subtree points: $(7,8),(9,10)$
centre being the root: $(5,6)$.
([Left Subtree, Right subtree, Root Point]) ♣

Construction time and space complexity of 2-dimensional KD-Tree:

The most expensive part of the construction of KD-Tree is sorting the points on both the axis. Instead of working with more complex algorithm to find the median we first presorted the values on x-axis and then on y-axis.
$T(n) = O(n)$, if $n = 1$
$O(n) + 2T(n/2)$, if $n > 1$

The above equations then solve to $O(nlogn)$.

Since KD-Tree is A binary tree, and every leaf and internal node uses $O(1)$ storage, hence the total storage is $O(n)$.

2.2.2. LISA: Learned Index for Spatial Data

Spatial data and query processing have become ubiquitous due to proliferation of location-based services such as digital mapping, location-based social networking, and geo-targeted advertising. Motivated by the performance benefits of learned indices for one-dimensional data, this section explores the application of learned index for spatial data. The main motivation is to map spatial data into one-dimensional data through several steps and apply machine learning techniques to generate a learned index for the one-dimensional data.

Motivation

In the last section, we described a recursive model index (RMI) that consists of a number of machine learning models staged into a hierarchy to enable synthesis of specialised index structures, termed learned indexes. Provided with a search key x, RMI predicts the position of x's data with some error bound, by learning the CDF over the key search space. However, the idea of RMI is not applicable in the context of spatial data as spatial data invalidates the assumption required by RMI that the data is sorted by key and that any imprecision can be easily corrected by a localised search. Although it is possible to learn multi-dimensional CDFs, such CDFs will result in searching local regions qualified on one dimension but not all dimensions.

Example 2.2 For example, consider the joint cumulative function of two random variables X and Y defined as $F_{XY}(x, y) = P(X \leq x, Y \leq y)$. The joint CDF satisfies the following properties:

- $F_X(x) = F_{XY}(x, \infty)$, for any x (marginal CDF of X)
- $F_Y(y) = F_{XY}(\infty, y)$, for any y (marginal CDF of Y)
- if X and Y are independent, then $F_{XY}(x, y) = F_X(x)F_Y(y)$

♣

Need to find a solid argument to explain why learning multi dimensional CDFs will result in searching local regions qualified on one dimension.

LISA solves this problem by partitioning search space into a series of grid cells based on the data distribution and building a function map the data from $\mathbb{R}^d$ into $\mathbb{R}$, in our case, we have $d = 2$. We call this function as *Mapping Function*.

Definitions

This section presents the definition

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 $(l_0, l_1) and (u_0, u_1)$, i.e., cell = $(l_0, u_0) \times [l_1, u_1)$
3. **Mapping Function**. A mapping function $\mathcal{M}$ is a function on the domain $\mathbb{R}^2$ to the non-negative range, i.e $M : [0, X_0] \times [0, X_1] \rightarrow [0, +\infty)$ such that $M(x_0, x_1) \leq M(y_0, y_1)$ when $x_0 \leq y_0$ and $x_1 \leq y_1$.

2.2.3. Baseline Method

We can extend the learned index method for range queries on spatial data by using a mapping function. This baseline method works as follows. We first sort all keys according to their mapped values and divide the mapped values into some cells such that each cell contains the same number of keys (except the last one). If a point (x, y)'s mapped value is larger than those of the keys stored in the first i cells, i.e. $\mathcal{M}(x, y) > \sup \bigcup_{j=0}^{i-1} M(C_j)$, we store (x, y) in the $(i + 1)$th cell.

For a range query, we have a 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 2.3 As shown in Fig. 2.6, the key space is divided into 3 cells using the mapping function $\mathcal{M}((x, y)) = x + y$. The query rectangle consisting of only 1 key, falls inside the second part. During prediction, we need to find out the cells to which our query rectangle belongs (the 2^{nd} cell in our example). Once the cell is found, we need to compare the key of the query point, against all the possible keys in that cell until a match is found. This results in 8 irrelevant points accessed for the range query that only contains one relevant key. ♣

Training

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.

In order to construct the baseline model, we need to have several parameters listed below:

1. N, which represents the number of cells into which the key's mapped value space will be divided.

As described in Algorithm 6, during training, we perform the following operations:

1. Sort all keys according to their mapped values.

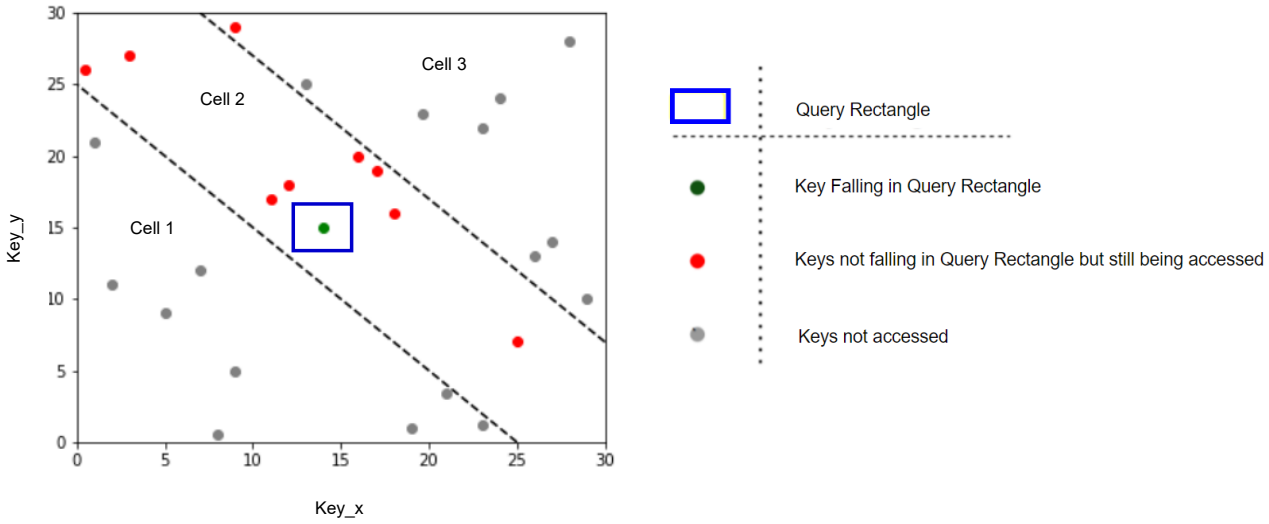


Figure 2.6.: LISA Baseline Method
a) key space is divided into 3 cells with equal nu of keys
b) To search for a query, we first need to find out the cell which contain the query point
c) Once the query point is found, we need to compare the query point key values with all the keys in the cell until a match is found

2. Divide the keys into equal sized cells

3. Store the mapped values of first and last key for each cell into an array

Algorithm 4: Training Algorithm for Lisa Baseline Method

input : num_of_cells; trainset=$[(x, y); x \in \mathbb{R}^2; y \in \mathbb{R}]$
Output: M:Mapped Function

for $i \leftarrow 0$ **to** $len(x)$ **do**
    `x[i].mapped_value = x[i][0]+x[i][1]`
end
`Sort x based on x.mapped_value`
`Divide x into equal size pages according to num_of_cells`
`Store mapped value of first and last key for each page`
for $i \leftarrow 0$ **to** *num_of_cells* **do**
    `denseArray[i].lower = first key in page i`
    `denseArray[i].upper = last key in page i`
end

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.

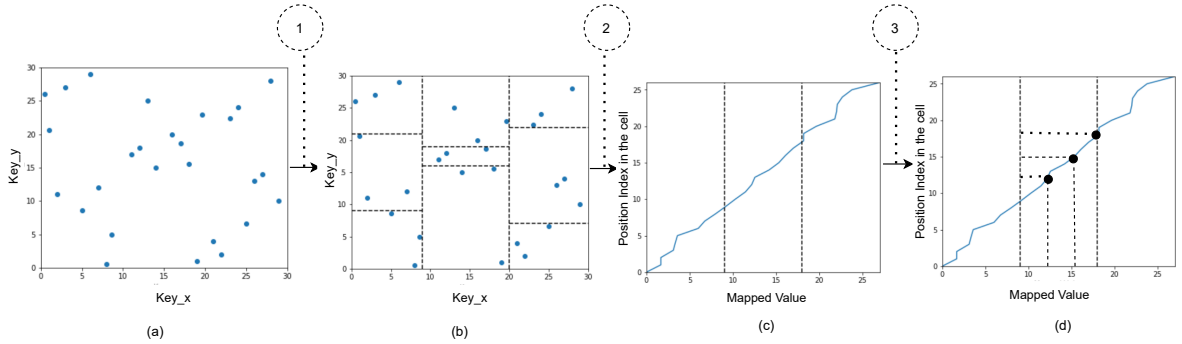


Figure 2.7.: Lisa Framework
1) Generate grid cells, and apply Lebesgue Measure to each cell to map two dimensional key value to a scalar.
2) Sort mapped values and divide them across equal length intervals termed as mapped intervals(3 in the figure)
3) For each mapped interval,divide the mapped value range in shards(3 in the figure) and learn a linear regression function per shard

Prediction

Algorithm 5: Prediction Algorithm for Lisa Baseline Model

input: `x_test`: Key; `d`: the denseArray

```
x_test.mapped_value = x_test[0]+x_test[1]
for i ← 0 to len(denseArray) do
    if M(x_test) ∈ [d[i].lower, d[i].upper] then
        Key is in Page i
        break
    end
end
Sequentially search for x_test in page j
```

2.2.4. Lisa Overview

Given a spatial dataset, we generate the mapping function $\mathcal{M}$, the shard prediction function $\mathcal{SP}$. Based on them, we build our index structure, LISA, to process range query and KNN query. LISA 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 2.7. the procedure of building LISA is composed of four parts.

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

4. Build local models.

Definitions

This section presents the additional definition specific to Lisa implementation.

4. **Shard**. The shard S is the pre-image of an interval $[a, b) \subseteq [0, +1)$ under the mapping function $\mathcal{M}$, i.e., $S = M^{-1}([a.b))$.
   Given an initial data set, we divide the key space into cell grids based on the data distribution, map keys values to an one dimensional space using mapping function, followed by learning several monotonic shard prediction functions. After sorted, the one dimensional mapped value space is then divided into equal-length intervals, and one shard prediction function is learned for each interval, to partition the keys belonging to a particular interval, into different shards. As keys are sorted by mapped values before partitioning them into equal sized intervals, and all shards exhibit a total order with respect to their corresponding intervals in the mapped range(Shard Prediction function for each interval is monotonically increasing), following relationship holds
   $inf(M(S_i)) > sup(M(S_j))$ when $i > j$.

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 .

2.2.5. Design and Implementation Details

Grid Cells Generation

The first task in Lisa implementation is to partition the 2 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 principal idea behind this partition strategy is to divide the key space into cell boundaries and apply a mapping function to create monotonically increasing mapping values at the cell boundaries.

$M(x_i \in V) < M(x_j \in V)$ when $i < j$, where $x_i \in C_i$ and $x_j \in C_j$

i.e. mapped value of a key in cell i will always be less than mapped values of a key in cell j, if $i < j$.

Example 2.4 Consider the example shown in the figure 2.8: 27 keys are partitioned into 9 cell, resulting in 3 keys per cell. To partition the key space, we first sort the keys values according to 1^{st} dimension and divide the keys into 3 vertical columns each containing 9 keys. Then for each vertical column of 9 keys, we sort the keys again according to 2^{nd} dimension, and divide the keys in each column into 3 new cells. The number of cells N into which the keys space is divided, 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 don't contain overlapping keys.

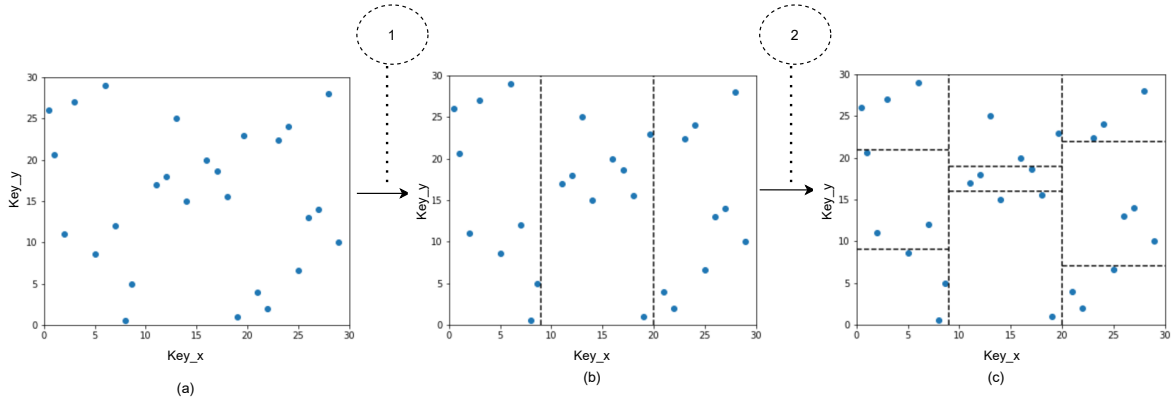


Figure 2.8.: Cell Partition Strategy:
1) : Sort Keys on x dimension and divide into 3 vertical columns each containing 9 keys
2) : Sort each vertical column keys on y dimension and divide into 3 horizontal columns each containing 3 keys

Algorithm 6: Grid Cell Generation Algorithm for Lisa Method

input: `num_of_cells;x; y`

`trainset=[`$(x, y); x \in \mathbb{R}^2; y \in \mathbb{R}$`]`

$keysPerPage = len(x)/num\_of\_cells$

`Sort x based on first dimension x[:][0]]`

`In first for loop, divide the keys into equal size subsets based on first dimension`

for $i \leftarrow 0$ **to** $\sqrt{(num\_of\_cells)}$ **do**

  `Store the 1st dimensional coordinates of first and last key for each cell. Each such cell will contain` $keysPerPage * sqrt(num\_of\_cells)$ `keys`

end

`Sort keys in each cell based on 2nd dimension,x[:][1]`

for $i \leftarrow 0$ **to** $\sqrt{num\_of\_cells}$ **do**

  for $j \leftarrow 0$ **to** $\sqrt{(num\_of\_cells)}$ **do**

    |

  end

  `Store the 2nd dimensional coordinates of first and last key for each cell.`

end

Mapping Function

A mapping function $\mathcal{M}$ is a function on the domain $\mathbb{R}^2$ to the non-negative range, i.e $M : [0, X_0] \times [0, X_1] \rightarrow [0, +\infty)$ such that $M(x_i \in V) < M(x_j \in V)$ 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 $i < j$.

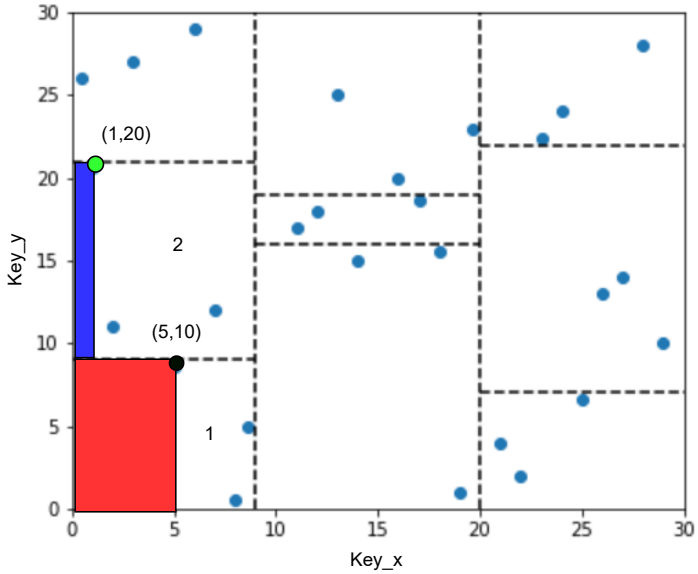


Figure 2.9.: Lebesgue Measure Representation for 2 dimensional data
1) Lebesgue Measure for the black point in first cell will be ratio of area of red rectangle divided by the total area of 1^{st} cell = $50/100 = 0.5$
1) Lebesgue Measure for the green point in second cell will be ratio of area of blue rectangle divided by the total area of 2^{nd} cell = $20/100 = 0.2$

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

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

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

As shown in figure 2.9, 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 2.10 shows the mapping of 2 dimensional key space 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 Ψ is the average number of keys falling in a shard. With these definitions, we know that

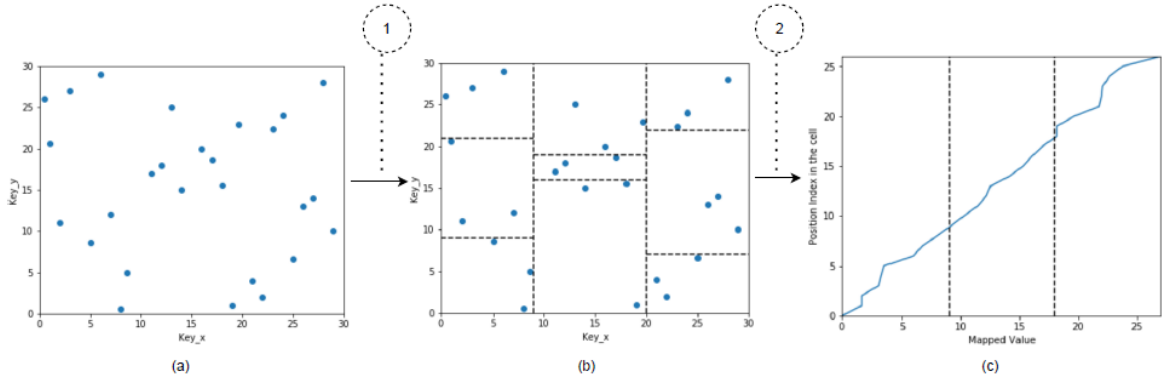


Figure 2.10.: Mapping 2 dimensional key Values to one dimensional cdf
1) Generate grid cells, and apply Lebesgue Measure to each cell.
2) Sort key in each cell according to mapped value. Mapped values in consecutive cells are already sorted by mapping function definition. Plot the cdf of mapped values.

each $\mathcal{F}_i$ generates $D = \lceil \frac{V+1}{\Psi} \rceil$ shards.

Example 2.5 For example, assume we have a dense array of mapped values as

$$[1, 1.2, 2.2, 3, 3.4, 4]$$

We want to partition it into 2 parts, so we have $\boldsymbol{M}_p = [3]$ and $V + 1 = 3$. In this case we will train 2 linear regression functions. Suppose that the average number of keys in a shard is $\Psi = 2$, then each $\mathcal{F}_i$ generates $D = \lceil \frac{V+1}{\Psi} \rceil = \lceil \frac{3}{2} \rceil = 2$ shards. ♣

Then with a given x, the predicted shard is given by $\mathcal{SP}(x) = \mathcal{F}_i(x) + i \times D$, where $i = \text{binary-search}(\boldsymbol{M}_p, x)$. 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.

Example 2.6 In the above example, given a key $x = 2.2$, 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 $D = \lceil \frac{V+1}{\Psi} \rceil = 2$ shards, the shards that the first linear regression function generates will be from 0 to 1 and the shards that the second linear regression function generates will be from 2 to 3. Hence, the predicted shard id 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 id 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 2.7 In our previous example, in the interval $[0, 3)$, we have $\boldsymbol{x} = (1, 1.2, 2.2)$ and $\boldsymbol{y} = (0, 1, 2)$. Then for a point with the mapped value $m = 1.2$, the expected output will be $f_i(m) =$

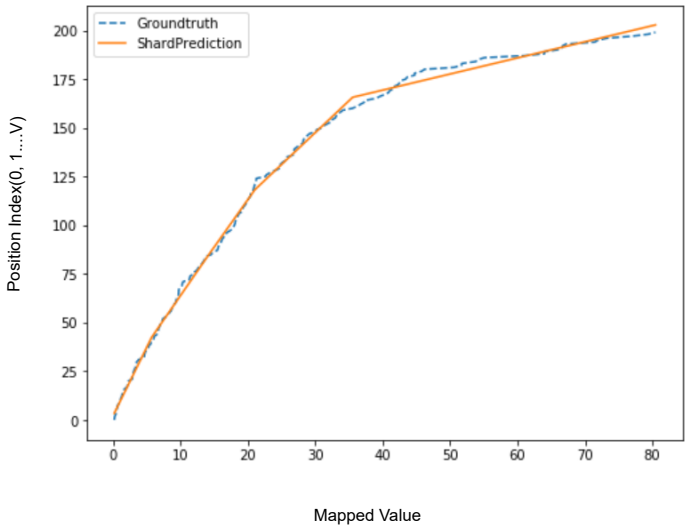


Figure 2.11.: Piecewise linear regression functions learnt by ShardTrainingAlgorithm, V is the number of keys per mapped interval

1 and the shard id is given by $\lceil \frac{1}{2} \rceil + 0 \times 2 = 1$. Hence, the point with mapped value $m = 1.2$ will be allocated to the first shard. 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 2.11. ♣

Formally, a piecewise linear function can be described as

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

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

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

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

$$\sum_{i=0}^{\eta} \alpha_i \geq 0, \forall 0 \leq \eta \leq \sigma$$

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

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

where $1_{x_0 \geq \hat{\beta}_1}$ equals to 1 if $x_0 \geq \hat{\beta}_1$, 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

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

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

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

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

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

Show how these are calculated

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.

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

Show how these are calculated 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 7, we perform following operations during shard training, :

1. Divide the sorted mapped values into equal sized U intervals.
2. Suppose V +1 is the number of mapped values in each interval and ψ is the estimated average number of keys falling in a shard.
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 = $\lceil \frac{V+1}{\Psi} \rceil$ number of shards
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 7.

Algorithm 7: Shard Training Algorithm

input: M_p, Ψ, U

Partition M_p into equal length U intervals $\boldsymbol{M}_p = [m_1, \cdots, m_U]$

for $i \leftarrow 0$ **to** U **do**

  $x = [x_0, \cdots, x_V]$ be the keys' mapped values in interval i

  $y = [0, \cdots, V]$

  Initialize $\beta^{(0)}$ as $\beta^{(0)} = x_0$ and $\beta_i^{(0)} = x_{\lfloor i \times \frac{V}{\Psi} \rfloor}, \forall i \in [1, \sigma]$

  while $k <$ *1000* **do**

    Initialize $A^{(k)}$ according to (2.7)

    $\alpha^{(k)} = ((A^{(k)})^T A^{(k)})^{-1} (A^{(k)})^T y$

    Calculate $g^{(k)}, Y^{(k)}$

    $s^k = -(Y^{(k)})^{-1} g^{(k)}$,

    Find update step $lr^{(k)}$ such that

     $\mathrm{L}(\alpha^\star(\beta^k + lr^{(k)} s^k), \beta^k + lr^{(k)} s^k) = \min\{L(\alpha^\star(\beta^k + lr^{(k)} s^k), \beta^k + lr^{(k)} s^k)\}$

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

  end

end

Local Models for Shards

Local models are not relevant to our implementation as all the data is contained in the 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. In addition, 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 2.1 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

Point query in a B-Tree is basically the comparison of the value of the key that needs to be searched with the keys in the node. It first linearly checks the value in the root key and looks for a key value which is greater than the searched key. As soon as it finds a key greater than the searched key it will search in the child of the key before it. For example if we were to

search of key [41] in the example of the B-Tree used above, we would first check the value [11] and since it smaller than [31] and there are no keys greater than this in the root node it will search in the right child of the root node. It will then linearly search the node and locate it's associated value and return it.

Algorithm 8: Algorithm for B-Tree Search

```
Input: k ; key, root ; Root of the B-Tree
Output: Value associated with key
for i ← 0 to len(RootNode) do
    if keys in root greater than k then
        SEARCH_CHILD(key) //Search linearly the child associated with the key
         location one before
        if Child is a leaf then
            Linearly search until the key is reached
        else
            SEARCH_CHILD(key)
        end
    end
end
```

Point Query with Baseline Index Model

Point Query with Recursive Model Index

Algorithm 9: Training of Recursive Model Index

```
input: x; models; num_of_stages; max_y
stage← 0
next_model← 0
while stage <num_of_stages do
    output = model.predict(x)
    next_model=output*len(models[stage+1])/max_y
    if last stage then
        y = next
end
```

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 2.2 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:

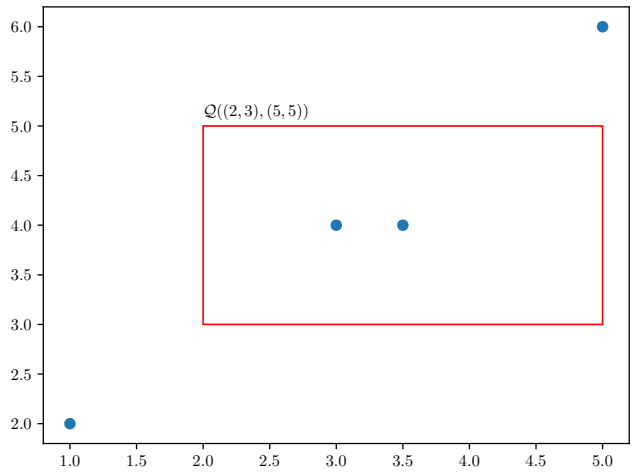


Figure 2.12.: A Range Query Example where $\mathcal{Q}(\boldsymbol{l},\boldsymbol{u}) = \mathcal{Q}((2,3),(5,5))$
In this example, the range query should return the points that lies inside the red rectangle, i.e. $[(3,4),(3.5,4)]$.

Range Query with KD-Tree

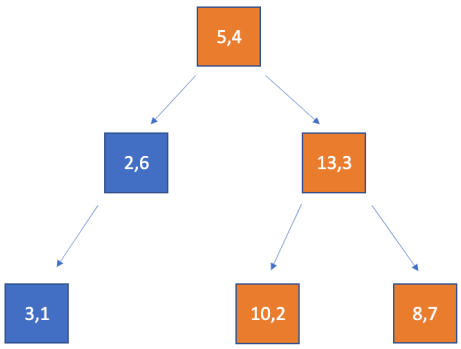


Figure 2.13.: KD-Tree for Range Query

We check the validity of a point lying withing the rectangle by checking the range of x coordinate and y coordinate of a point.

Example 2.3 For example we have a tree with Point list as

$$((5,6),(4,2),(7,9),(3,1),(5,5),(10,7),(2,11))$$

and with lower bound = $(2,3)$ and upper bound = $(6,7)$, we will get a tree as is shown in 2.13. We can see the points along with the rectangle range plotted in 2.14. Points $(5,5)$ and

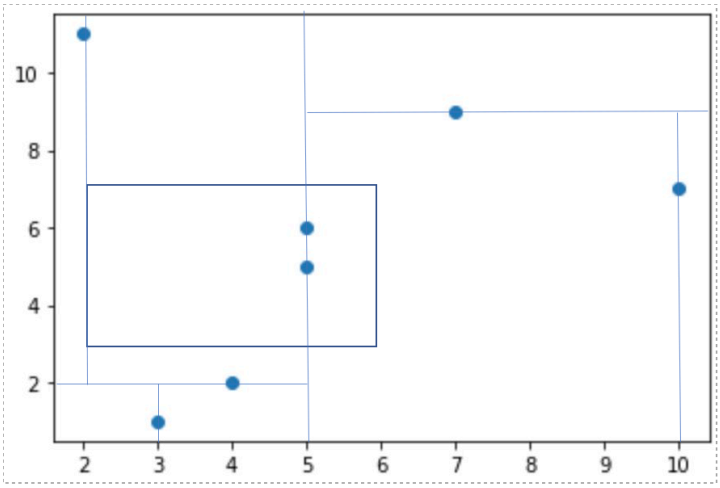


Figure 2.14.: KD-Tree Range Query Plot on 2-dimentional plane

$(5, 6)$ are returned in the query since they lie within the rectangle as seen in the plot. First the root point is checked and since the x-coordinate and y-coordinate both lie within the rectangle bounds i.e., $2 > 5 > 6$ and $3 > 6 > 7$. It then checks if the x-coordinate is lower than or greater than the lower bound x-coordinate. Since the value is larger than lower bound x-coordinate that is 2 it will then traverse to the left. In the left it has child node as $(4, 2)$ however, since the y-coordinate doesn't lie in the range of the upper bound this point is not selected. Therefore, it recursively traverses the tree and checks if the point lies within the bound until it reaches a leaf. ♣

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.

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, the KNN query is only performed on 2-dimensional data. In addition, we only use ℓ_2 norm as the distance metric. A KNN query will be formalised as $\mathcal{K}(\boldsymbol{X})$ where $\boldsymbol{X} \in \mathbb{R}^2$.

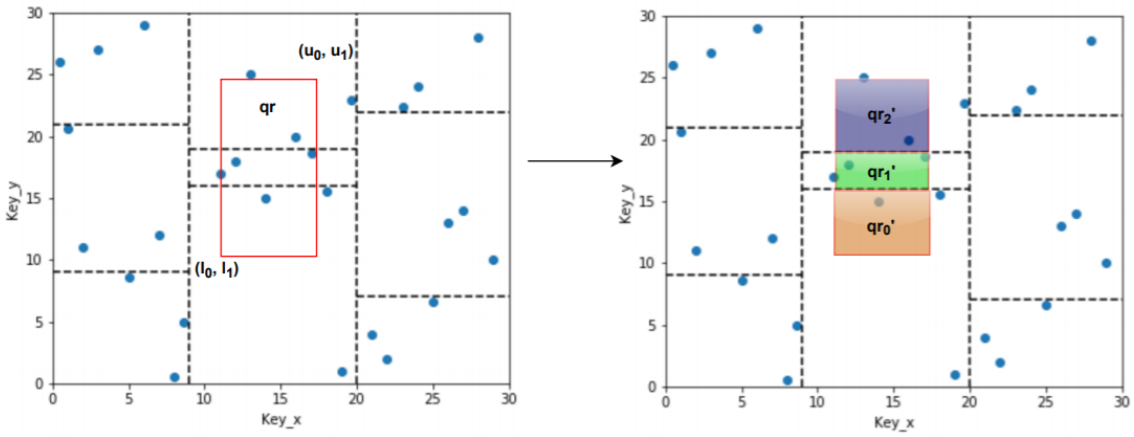


Figure 2.15.: Range Query Search in Lisa. 1) Find the cells that overlap with query rectangle qr.
2) Decompose qr into the unions of smaller query rectangles, each of which intersect one only one cell.
3) Find shards corresponding to lower and upper coordinates for each query rectangle, and perform a sequential search.

KNN query with KD-Tree

As a baseline, we first perform KNN query with KD-tree.

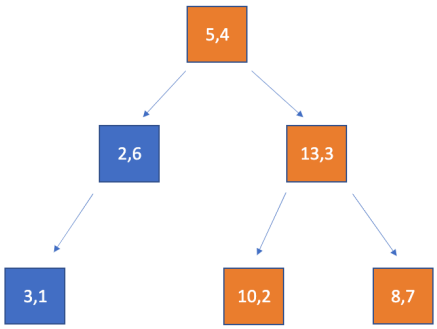


Figure 2.16.: KD-Tree for KNN Query

Example 2.4 For example, we have Point list as $((5,4),(2,6),(13,3),(8,7),(3,1),(10,2))]$ then we will have a tree structure as shown in 2.16 and it's plot on 2-dimensional plane is shown in 2.17. As we can see in 2.17 that even though point $(8,7)$ is the leaf we will reach when we traverse the tree to search for point nearest to test point $(9,4)$ it is not in fact the nearest point to the test data. In this case if we want to look for 4 nearest neighbour, we first check the square of euclidean distance of $(9,4)$ with the root of the tree i.e., $(5,4)$ which is 16. It then calculates the distance with $(2,6)$ and $(13,3)$ which calculates to 53 and 17 respectively. It then keeps traversing to the right node while calculating distance until it reaches a leaf. After reaching the leaf it then checks the distance with $(8,7)$ with a distance of 10 which is in fact

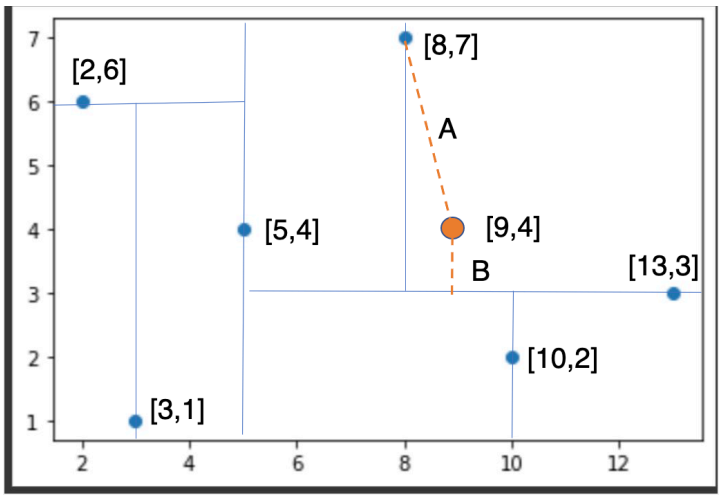


Figure 2.17.: KD-Tree KNN Plot on 2-dimentional plane

smaller than the previous shortest distance of 16 with the root. It adds these points to the list. It then makes a decision weather to go left from $(13, 3)$ based on the distance of the test point with leaf $(8, 7)$ i.e., A and the perpendicular distance with point $(13, 3)$ which is B. Since distance A > B as can be seen in the figure 2.17 there is a chance that there could a point in the subtree with a distance smaller than the previous points. In this case it will then check the distance with point $(10, 2)$ and the distance is the shortest(best distance) so far of 5. ♣

KNN Query with LISA

It is difficult to apply traditional KNN query pruning strategies applicable for KD-Trees, to LISA model as we don't maintain a tree like structure in LISA. Shard boundaries are learned per mapped interval and no data structure is maintained to refer to shards in adjacent mapped intervals.The key idea in the KNN query is to convert it into a range query by estimating an appropriate query range.The query range is augmented if less than K neighbors are found in a range query.

Consider 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. 2.18, 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 δ empirically. We try with different values of δ and choose the one for which we get the best results.

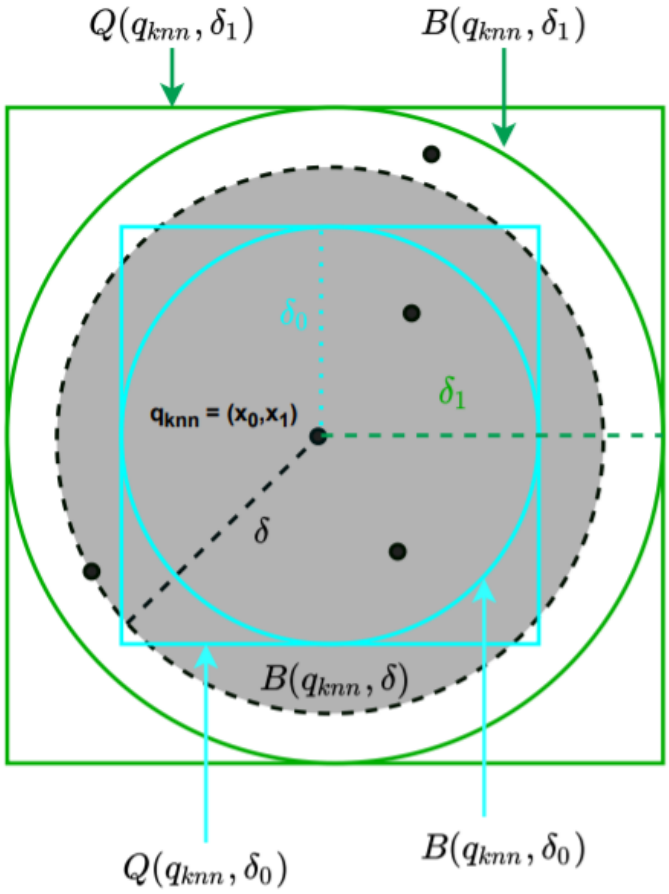


Figure 2.18.: KNN Query Implementation in Lisa(K=3)
1)q_{knn} represents the query point, $\mathcal{Q}(x, \delta) \triangleq [x_0 - \delta, x_0 + \delta) \times [x_1 - \delta, x_1 + \delta)$, represents query rectangle and $\mathcal{B}(x, \delta)$ represents the key space at distance δ containing K nearest keys.
2)KNN query can be solved by range query if we can estimate an appropriate distance bound δ for every query point

3. Evaluation

In order to evaluate the performance, we perform the evaluation on manually synthesised dataset.

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 into 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.

Small Lognormal Distributed Data

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 lognormal distribution.

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

- All models are fully connected neural networks. 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.
- 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.

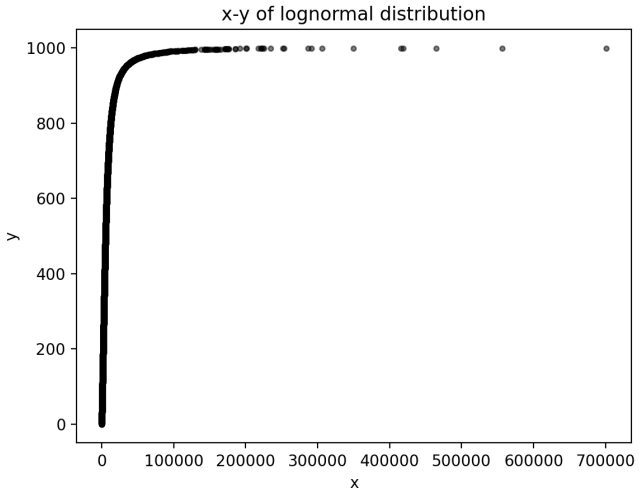


Figure 3.1.: The x-y graph where x is randomly sampled from a lognormal distribution

- 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 previous two group.

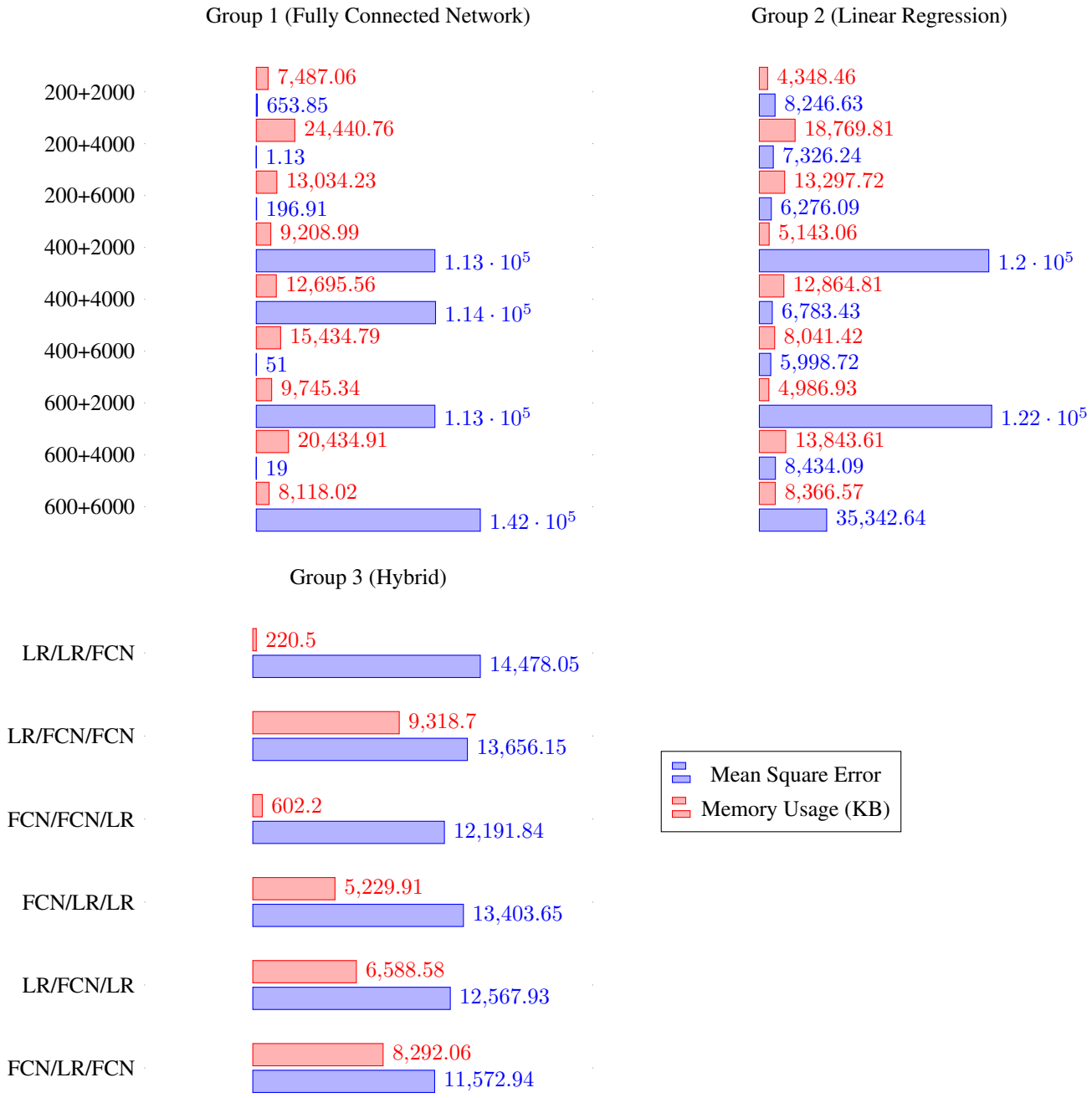
From the experiment results, we found that the second setting in group 1 (1 FCN model as root, 200 FCN models as second level models and 4000 FCN models as third level models) is the best regarding the mean square error. We also have the following findings in this searching process.

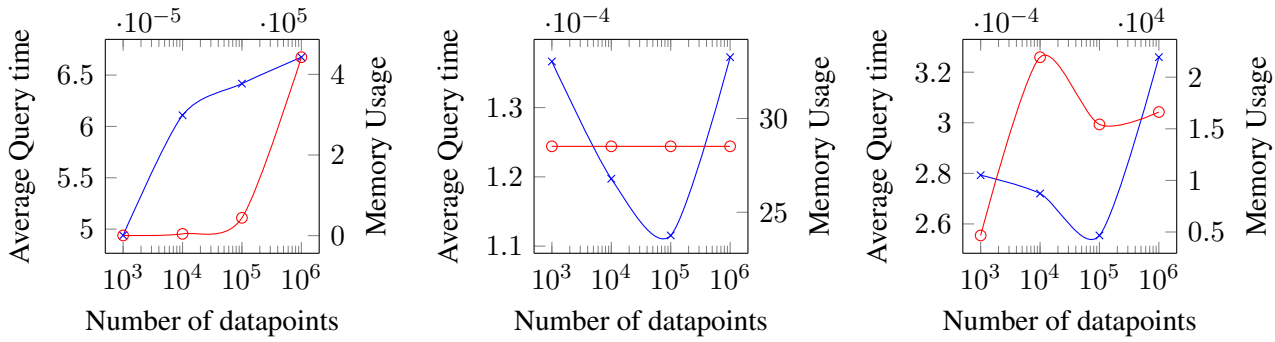
- Generally, the average error in group 1, where all models are fully connected neural networks is less than the error in group 2. Fully connected neural networks have a potential to be more accurate, i.e. it could achieve a small error if we tuned the models parameters properly.
- Tuning a model is 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.

Various Distributions and Sizes

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.
- For each distribution, we generate 1 thousand, 10 thousands, 100 thousands and 1 million data points. We then assign the generated data points into pages where $N_{page} = 10$.


Group 1 (Fully Connected Network)
Group 2 (Linear Regression)
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 · 10^5
12,695.56
1.14 · 10^5
15,434.79
51
9,745.34
1.13 · 10^5
20,434.91
19
8,118.02
1.42 · 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 · 10^5
12,864.81
6,783.43
8,041.42
5,998.72
4,986.93
1.22 · 10^5
13,843.61
8,434.09
8,366.57
35,342.64
Group 3 (Hybrid)
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

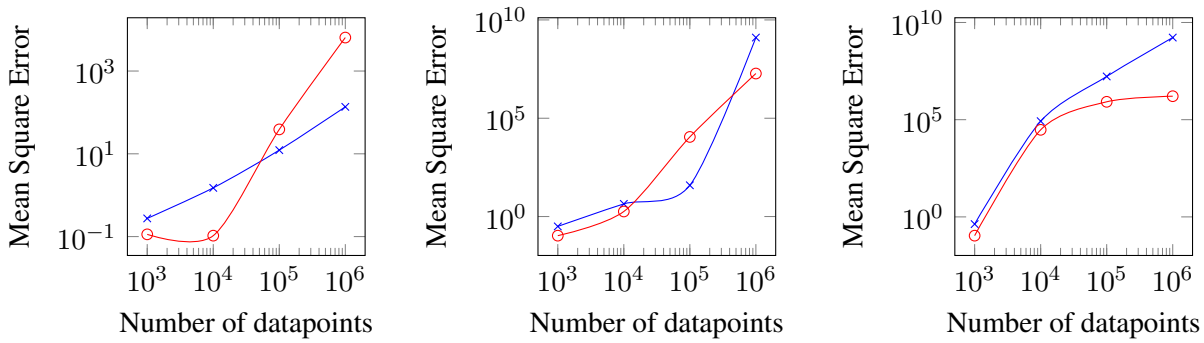
- We use a B-Tree with degree=20, a fully connected neural network with two layers and 32 nodes per layer, and a recursive model with 200 second-layer models and 4000 third-layer models.

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 fully connected network and recursive models, across different distributions.
- The construction time among three index models.

Conclusion 3.1 From Fig 3.2, we analysed 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 are 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 very likely caused 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



(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

network. The memory usage of a recursive model is fluctuating, as the actually used number of 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 are only fixed number of computations.

□

Conclusion 3.2 From Fig 3.3, we analysed the errors of fully connected neural network and recursive model on different distributed dataset.

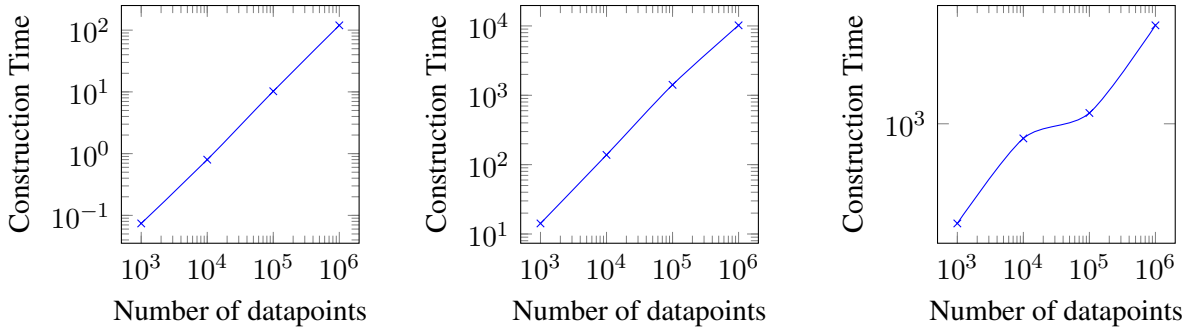
1. From Fig 3.3a, we found that both fully connected neural network and recursive model are capable of modelling uniformly distributed dataset with a rather low error. In the mean while, fully connected neural network could achieve less error, especially when there is large amount of data.

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

3. From Fig 3.3c, we found that the error from recursive model is significantly less than the error in 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 detail in the chapter *Insights and Findings*.

□

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



(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.

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 datapoints 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.

□

Large Lognormal Distributed Data 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 single process, it would take a tediously long time to train the recursive model.
2. It takes 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 which contains N pairs, 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 $\hat{y} = \frac{\tilde{y}}{R}$. An example is illustrated 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 exactly 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 from 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.

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

3.2.2. Task 1 : 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 implementation

Baseline model has one hyper-parameter: N (Number of cells specifying the number of equal length intervals into which mapped values are divided). As discussed in section, the 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. Optimum value of hyper-parameter N is to assign 1 key per page to mimimize the sequential search cost.

Conclusion As shown in table 3.1, following conclusions can be drawn:

1. Build time : Build time increases with increase in value of N, as metadata for additional cells needs to be calculated.
2. Average Query Time : Average Query Time decreases with increase in value of N as number of keys per cell decreases.
3. 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.

Training/Test Data Size	Model	N	Build Time (ms)	Avg Query Time (ms)	Memory Size (KB)
10,000	LISA Baseline	100	11.25	0.7189	315
10,000	LISA Baseline	10000	26.83	0.1985	547
100,000	LISA Baseline	1000	111.97	0.7271	3149
100,000	LISA Baseline	100000	272.93	0.2381	5469
1,000,000	LISA Baseline	1000	1104.65	4.4732	31274
1,000,000	LISA Baseline	1000000	2717.65	0.2436	54688

Table 3.1.: Hyper-parameters Search LISA Baseline Model
a) For any training size, optimum value of N will be equal to the number of keys in training data.
b) If N is assigned as numbers of keys in database, number of keys per cell will be 1, and query search cost will be reduced to $log_2 N$.

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 best performance if mapped interval boundaries are aligned to grid cell boundaries. That's why this parameter is initialized to N=$G \times G$

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

Conclusion Experiments results shown in tables 3.2, 3.3 and **??** are consistent across all 3 training sizes and have following interpretation.

1. For a particular value of G, average query time decreases and memory size increases with increase in value of S.

2. Average query time decreases and memory size increases with increase in values of G and S.

3. We need to choose S such that there are at least 35 keys per shard. We see mean square errors(mse) if number of keys per shard are less than 35 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

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 3.2.: 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 3.3.: Hyper-parameters Search LISA Model: Training Size:100,000 Points

b) For query points near the shard boundaries, there can be a mismatch in groundtruth shardId and predicted shardId.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.

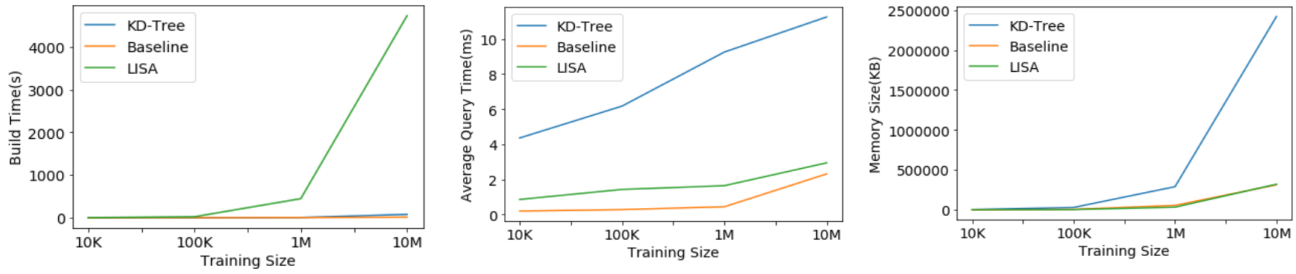
During test experiments we found that if shard size is less than 35 keys, then sometimes shard prediction error can be greater than 1 and point query search can fail resulting in mse errors.

3.2.3. Performance Comparison for KD-Tree, LISA-Baseline and LISA Models

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

Training/Test Data Size	Model	Build Time(s)	Avg Query Time(ms)	Memory Size(KB)	mse
10,000	KD-Tree	0.023	4.363	2890	0
10,000	Baseline	0.026	0.198	547	0
10,000	LISA	1.127	0.861	337	0
100,000	KD-Tree	0.340	6.176	28906	0
100,000	Baseline	0.324	0.241	5469	0
100,000	LISA	22.491	1.43	3169	0
1,000,000	KD-Tree	4.124	9.254	289062	0
1,000,000	Baseline	2.718	0.343	54688	0
1,000,000	LISA	445.324	1.445	32477	0

Table 3.4.: Point Query experimental results for KDTree, Baseline and LISA models



Plot-A (Build Time(s)) Plot-B (Average Query Time(ms)) Plot-C (Memory Size(KB))

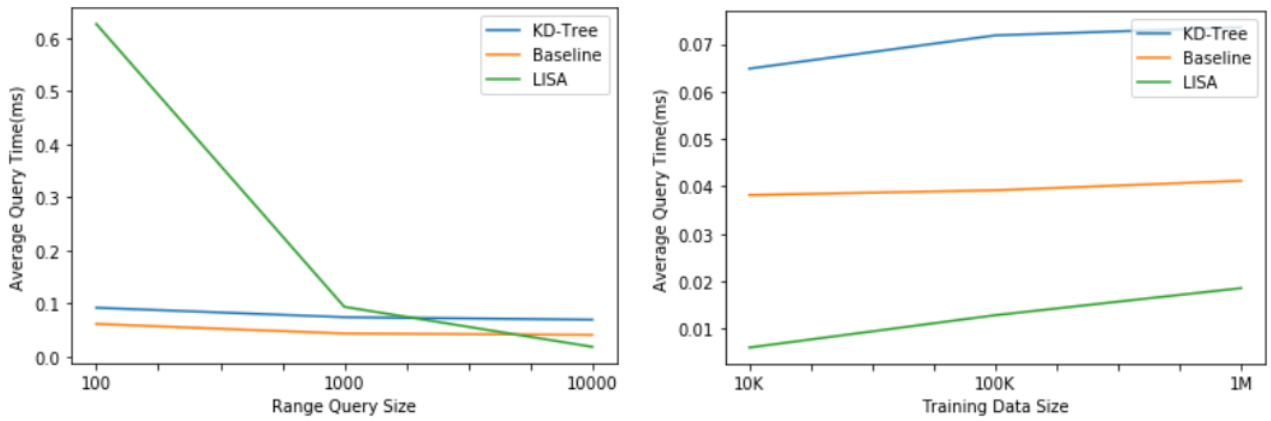
Figure 3.5.: Point Query experimental results for KD-Tree, Baseline and LISA models LISA outperforms KD-Tree in terms of average query time and memory requirements, however its build time is significantly higher than KD-Tree

Point Query Comparison for KD-Tree, LISA-Baseline and LISA Models

Table 3.4 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 dataset and take the average. As shown in the Fig. 3.5, LISA outperforms KD-tree in terms of average query time and memory requirements, however its build time is significantly higher than KD-Tree

Range Query Experiments

Table 3.5 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 range from sampled point to the range query size. Average query time for each range is further divided by the range size to compare the query time across various ranges. As shown in the Fig. 3.6, LISA outperforms KD-tree for range query size of 10000 for all training sizes,



Plot A: Average Query Time for various models for training size 10,000,000

Plot B : Average Query Time for various models for range query of size 10000

Figure 3.6.: Range Query experimental results for KD-Tree, Baseline and LISA models
a) Plot A shows average range query time for a fixed training size of 1M points. LISA outperforms KD-Tree for larger range queries.
b) 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

however its range query time for smaller range sizes is significantly higher than KD-Tree.

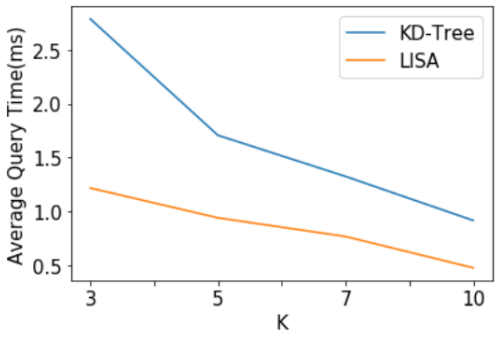
KNN Query Experiments

Table 3.6 shows evaluation results for LISA and KD-tree models for $\boldsymbol{K}$NN Queries for various value of K and training sizes. For a given $\boldsymbol{K}$ 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. Average query Time is further divided by $\boldsymbol{K}$ to compare the Query time across various values of $\boldsymbol{K}$.As shown in the Fig. 3.7, LISA outperforms KD-tree for different training sizes and K values.

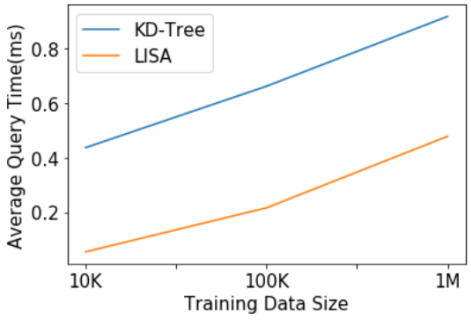
Training/Test Data Size	Range Query Size	Avg Query Time(ms)(KD-tree)	Avg Query Time(ms)(Baseline)	Avg Query Time(ms)(LISA)
10,000	10	0.1361	0.1113	0.8204
10,000	100	0.0533	0.0451	0.1201
10,000	1000	0.0438	0.0399	0.0294
10,000	10000	0.0648	0.0382	0.0061
100,000	10	0.1392	0.1298	2.8961
100,000	100	0.0539	0.0505	0.2792
100,000	1000	0.043	0.0428	0.0563
100,000	10000	0.0718	0.0392	0.0129
1,000,000	10	0.2238	0.2661	3.5181
1,000,000	100	0.0922	0.0617	0.6263
1,000,000	1000	0.0744	0.0437	0.0939
1,000,000	10000	0.0735	0.0412	0.0186

Table 3.5.: Range Query experimental results for KD-tree, Baseline and LISA models

Query.pdf



Plot A: Average KNN Query Time for training size of 1M Points for different values of K



Plot B : Average KNN Query Time for different training sizes for K = 10

Figure 3.7.: KNN Query experimental results for KD-Tree and LISA models
a) Plot A shows average KNN query time for a fixed training size of $1M$ points for different values of K. LISA outperforms KD-Tree for all values of K.
b) Plot B shows average KNN query time for various training sizes with K = 10. Lisa outperforms KD-Tree for all training data sizes.

Training/Test Data Size	K	Avg Query Time(ms)(KD-tree)	Avg Query Time(ms)(LISA)
10,000	3	1.4069	0.2020
10,000	5	0.8753	0.1181
10,000	7	0.6333	0.0811
10,000	10	0.4368	0.0549
100,000	3	2.0325	0.5867
100,000	5	1.2004	0.3549
100,000	7	0.8812	0.2779
100,000	10	0.6618	0.2161
1,000,000	3	2.7865	1.218
1,000,000	5	1.7072	0.9414
1,000,000	7	1.3255	0.7681
1,000,000	10	0.9172	0.5779

Table 3.6.: KNN Query experimental results for KD-tree and LISA model

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.

Requirements

For a learned index.

4.2. One Dimensional Learned Index

4.2.1. Baseline Learned Index

Activation Functions

- 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 $z^{(1)}(w^{(1)}x + b^{(1)}) = w^{(1)}x + b^{(1)}$. Then the output will be the input of the next layer, and hence the output of the second layer will be $z^{(2)}(w^{(2)}(w^{(1)}x + b^{(1)}) + b^{(2)}) = w^{(2)}w^{(1)}x + w^{(2)}b^{(1)} + b^{(2)}$. Similar induction can be obtained for multiple layers. Hence if we use identity activation, the trained neural network will fall back to a linear regression. The visualization below shows our lemma is correct.

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

  Proof: The output with ReLU activation function, will be $z^{(1)}(w^{(1)}x + b^{(1)}) = \max(w^{(1)}x + b^{(1)}, 0)$. Then the output will be the input of the next layer, and hence the output of the second layer will be $z^{(2)}(w^{(2)}(w^{(1)}x + b^{(1)}) + b^{(2)}) = \max(w^{(2)}w^{(1)}x + w^{(2)}b^{(1)} + b^{(2)}, 0)$.

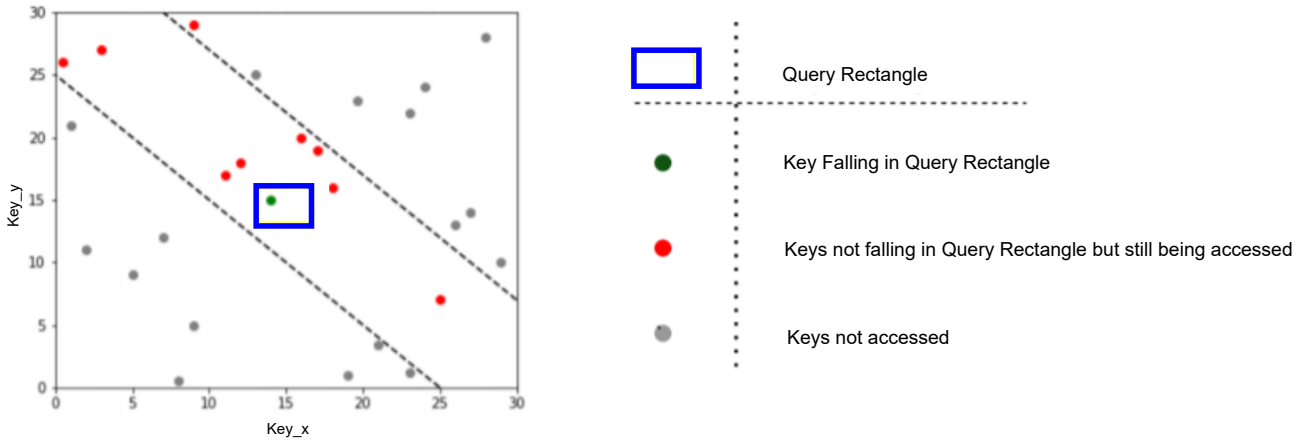


Figure 4.1.: Baseline Method Limitation

Similar induction can be obtained for multiple layers. Hence if we use identity activation, the trained neural network will fall back to a piecewise linear function. The visualization below shows our lemma is correct.

4.3. Two Dimensional Learned Index

Limitation

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 $N_2 \div 2$, 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 figure 4.1. 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.

Lisa Baseline model search optimization

In case of high dimensional key values, key with in a page 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. Once mapped value is found using binary search, we do a look up in its neighbourhood based on 2 dimensional key value. As shown in table **??**, we get significant savings in the query time with this approach.

Training/Test Data Size	Model	No. of cells	Build Time(ms)	Avg Query Time(ms)	Memory Size(KB)
10,000	Lisa Baseline	10	11.1208	0.284191	313.77
10,000	Lisa Baseline	100	12.0108	0.277918	315.85
10,000	Lisa Baseline	1000	12.7589	0.276572	336.97

Table 4.1.: Experimental results for baseline model with search optimization, Training Data Size : 10,000 points

Training/Test Data Size	Model	No. of cells	Build Time(ms)	Avg Query Time(ms)	Memory Size(KB)
100,000	Lisa Baseline	10	112.973	0.285525	313.77
100,000	Lisa Baseline	100	114.318	0.282053	315.85
100,000	Lisa Baseline	1000	116.699	0.280637	336.97

Table 4.2.: Experimental results for baseline model with search optimization, Training Data Size : 100,000 points

Training/Test Data Size	Model	No. of cells	Build Time(ms)	Avg Query Time(ms)	Memory Size(KB)
1,000,000	Lisa Baseline	10	1116.51	0.240508	313.77
1,000,000	Lisa Baseline	100	1118.85	0.235858	315.85
1,000,000	Lisa Baseline	1000	1134.88	0.234435	336.97

Table 4.3.: Experimental results for baseline model with search optimization, Training Data Size : 1,000,000 points

5. Convolution and CNN for Learned Indexes

6. Conclusion

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.

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 6.1.

Bibliography

[KBC+18] 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.

[LLZ+20] 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.