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`

January 11, 2010

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 efficiently find records. B-tree and KD-tree are the two of the 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 for 2-dimensional data [LLZ+20]. Afterwards we conduct several experiments to evaluate the performance of learned indexes compared with traditional B-Tree and KD-Tree. In addition to the implementation and evaluation, we then theoretically analyse the properties that the learned indexes hold and should hold.

As extension to the existing learned indexes, we also explore the possibilities of using convolution operation and convolutional neural network to improve the performance of 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 important roles in databases and have been widely used in modern data management systems (DBMS) such as PostgreSQL. Despite their huge success, one shortcoming of these hand-engineered data structure is that they do not consider the distribution of the database entries, which might be helpful in designing faster indexes.

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 extremely 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 report, we explore the development of database indexes, from hand-engineered indexes to 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. **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	
$\mathbf{X}$	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

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.

1.3 Motivation

In traditional database indexes, the complexity for locating an item is usually bounded by some function related to the total number of elements. For example, with a B-Tree, an item can be found within $\mathcal{O}(\log n)$ time complexity. In the meantime, saving a B-Tree as index takes n space complexity. With the rapid growing of the volume of data, n becomes much larger than ever before. Hence, the big data era is calling for a database index that have constant complexity in both time and space.

To achieve such a goal, the distribution of the data is important. For example, assume that the data is fixed-length records over a set of continuous integers from 1 to 100 million, the conventional B-Tree index can be replaced by the keys themselves, making the query time complexity an $\mathcal{O}(1)$ rather than $\mathcal{O}(\log n)$. Similarly, the space complexity would be reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. This example shows that with the knowledge of the distribution of the data, it is possible to locate the item in database in constant time.

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**.

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.

Even though there might be potential advantages, the learned index model has several assumptions, as listed below.

1. All data records are stored in memory.

2. All data records are sorted by X.

3. All data records are stored statically in database, hence we do not take insertion and deletion into consideration.

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. For example, the PostgreSQL uses B-Tree as its index. B-Trees can be considered as a natural generalisation of binary search tree. In binary search tree, there is only one key and two possible children in the internal node. However, an internal node of B-Tree 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. The image below illustrates a simple B-Tree.

In this section, we introduce the construction and query processes of B-Trees and then analyse their properties.

Motivation

In computers, the memories are organised in an hierarchical way. For example, a classical computer system consists three layers of memory: the CPU cache, main memory and the hard disk. In such a system, the CPU cache is the fastest but the most expensive while and hard disk is the cheapest but also the slowest. When querying for an item, the CPU will first try to fetch it from the CPU cache. If not there the CPU will then try to fetch it from the main memory, and then the hard disk.

At the same time, the traditional hard disk drive (HDD) is made by a moving mechanical structure.

In summary, there are two properties in classical computer systems that we need to take into account:

1. The memory is not flat, meaning that memory references are not equally expensive.

2.

Definition and Terms

Before we formally define B-Trees, we assume the following terms:

- **Keys**: The key in a database is a special attribute that could identify a row in the database. In our work, each key corresponds to a **value** and forms a key-value pair.

- **Internal Node**: An internal node is any node of the tree that has child nodes.
- **Leaf Node**: A leaf node is any node that does not have child nodes.

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.

Insertion of B-Tree

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

Algorithm 1: Grid Cell Generation Algorithm for Lisa Method

```
Input: $m$:order_of_tree ,$(k,v)$:(key, value), $N$:Node;
Output: Btree is constructed
if $N$ is not yet full then
    insert $(k,v)$ into $N$
else
    create new Node N′
end
if $N$ is is a leaf then
    Temp:=$N$+$(k,v)$, $k'$:=$ceil(m/2-1)$
    k’ is the median index of Temp move entries greater than median to k
else
    Temp:=N+(k,p), k′:=ceil(m/2-1)
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
        Store the 2nd dimensional coordinates of first and last key for each cell.
    end
end
```

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

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

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 2.1.

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.

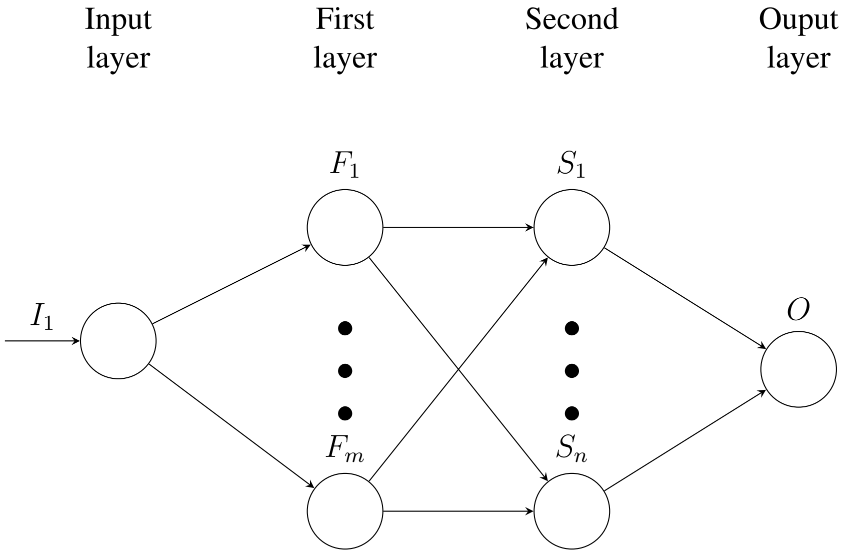

Figure 2.1: The architecture of the fully connected neural network used as baseline learned index. In this neural network, we use only 2 fully connected layers. The input of this neural network is only one neuron such that it represents the given query key. The output of this neural network is limited to 1 neuron such that it represents the predicted proportional position of the key-value pair.

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 $\boldsymbol{z_1} = \text{max}(\boldsymbol{y_1}, 0)$.

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 $\boldsymbol{z_2} = \text{max}(\boldsymbol{y_2}, 0)$.

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

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

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. 2.2. 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.

> For example, in the Fig. 2.2, 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.

Definitions

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

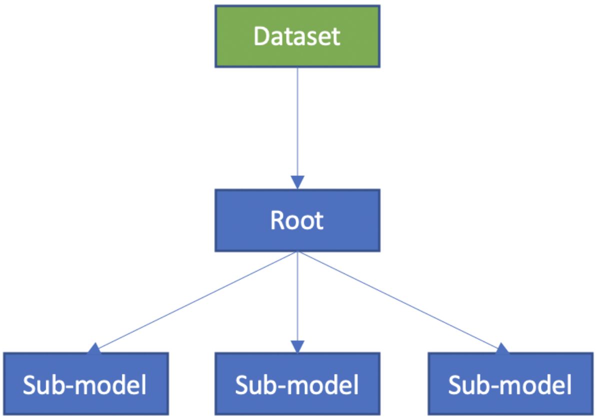

Figure 2.2: An example recursive model index with one root model and three leaf model.

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 $\texttt{next} = y_0 * N_M^{(i+1)} / \texttt{max(Y)}$.

Algorithm 2: Training of Recursive Model Index

```
input: num_of_stages; num_of_models; types_of_models; x; y
trainset=[[(x,y)]]
stage← $0$
while stage <num_of_stages do
    while model <num_of_models[stage] do
        model.train(trainset[stage][model])
        models[stage].append(model)
    end
    if not last stage then
        for $i \leftarrow 0$ to $len(x)$ do
            model=models[output from previous stage]
            output=model.predict(x[i])
            next=output * num_of_models[stage+1]/max_y
            trainset[stage+1][next].add((x[i],y[i]))
        end
end
```

Prediction

Algorithm 3: 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
```

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 $Y = \boldsymbol{X\beta}$.

Our goal is to find β such that the sum of squared error, i.e. $\mathrm{S}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (\hat{y} - y)^2$ is minimal. This optimisation problem can be resolved by ordinary least square estimation as shown below.

First we have the error as

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

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

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

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

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

By let it to be zero, we end up with

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

2.2 Two Dimensional Data

2.2.1 KD-Tree

KD-Tree

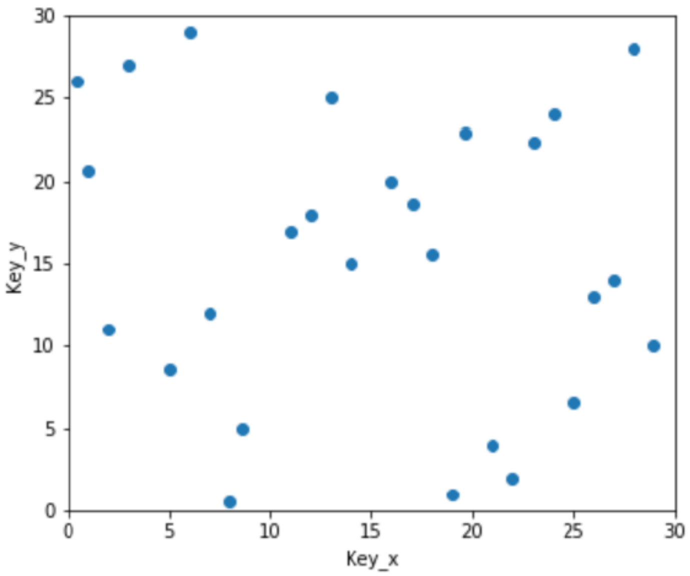

Figure 2.3: Key Distribution in 2 dimensional case:Idea of learned indexes is not applicable in the context of spatial data as data is not sorted by key. Learning multidimensional CDFs will result in searching local regions qualified on one dimension but not all dimensions.

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, as shown in Fig. 2.3, 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.

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 $M : [0, X_0] \times [0, X_1] \to [0, +\infty)$ 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.

As shown in Fig. 4.1, 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.
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.

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 $i \leftarrow 0$ to $len(denseArray)$ do
    if $\mathcal{M}(x\_test) \in [d[i].lower, d[i].upper]$ then
        Key is in Page i
        break
    end
end
Sequentially search for x_test in page j
```

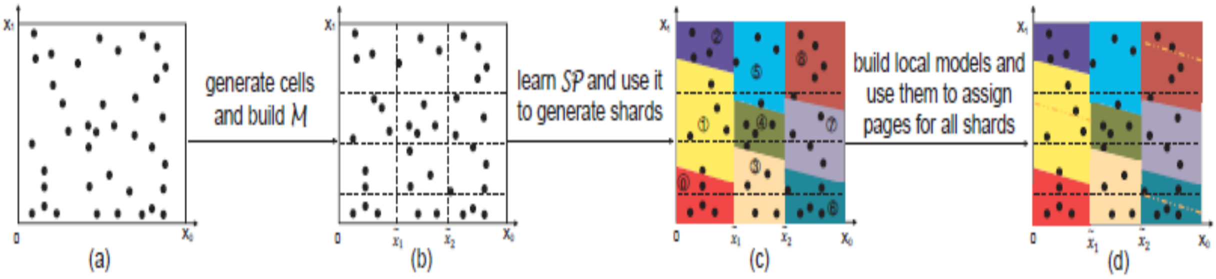
x1
x0
0
(a)
generate cells and build M
(b)
learn SP and use it to generate shards
(c)
build local models and use them to assign pages for all shards
(d)

Figure 2.4: Lisa Framework

2.2.4 Lisa Overview

Given a spatial dataset, we generate the mapping function 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 M, the shard prediction function $\mathcal{SP}$, and the local models for all shards. As illustrated in the Fig 2.4. the procedure of building LISA is composed of four parts.

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

Definitions

This section presents the additional definition specific to Lisa implementation.

4. **Shard**. 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 .

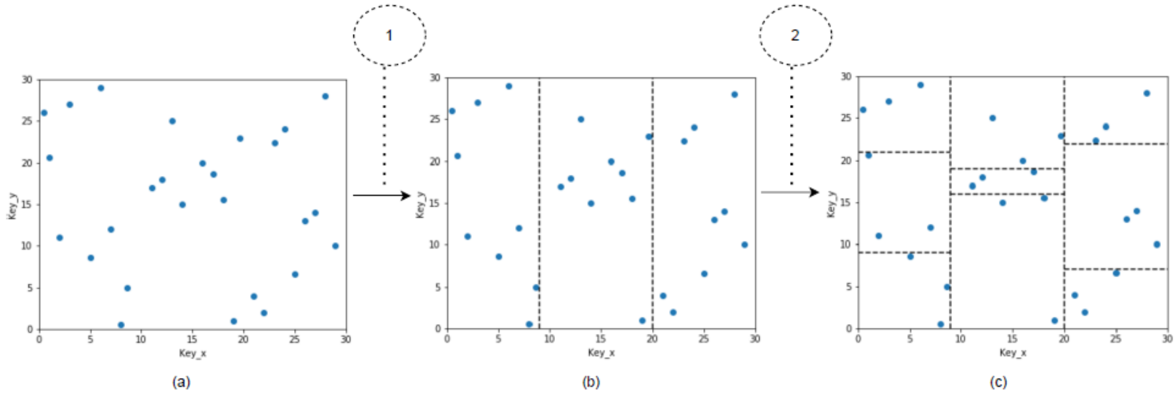

Figure 2.5: Cell Partition Strategy:
1) : Sort Keys on x dimension and divide into 3 vertical columns each containing 9 keys
2) : Sort each vertical column keys on y dimension and divide into 3 horizontal columns each containing 3 keys

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 $i < j$.

Consider the example shown in the figure 2.5: 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.

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 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 $i < j$.

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

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

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

As shown in figure 2.6, 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 2.7 shows the mapping of 2 dimensional key space to one dimensional CDF.

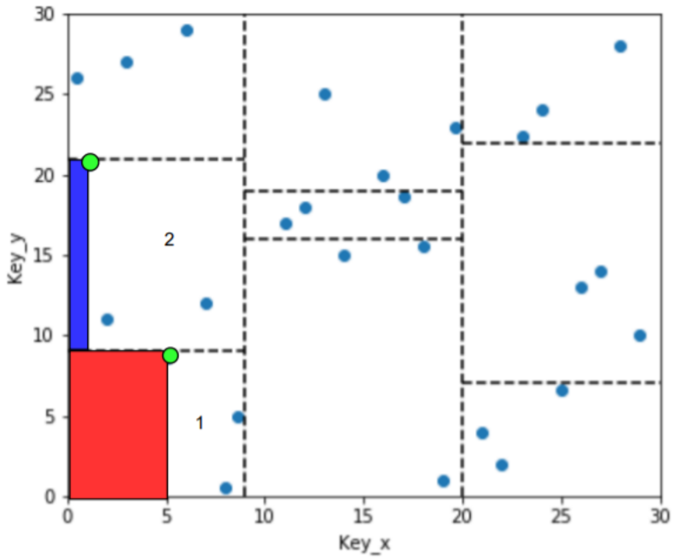

Figure 2.6: Lebesgue Measure Representation for 2 dimensional data
1) Lebesgue Measure for the green point in first cell will be ratio of area of red rectangle divided by the total area of 1^{st} cell
1) Lebesgue Measure for the green point in second cell will be ratio of area of blue rectangle divided by the total area of 2^{nd} cell

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

For example, assume we have a dense array of mapped values as $[1, 1.2, 2.2, 3, 3.4, 4]$, and we want to partition it into 2 parts, then 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.

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

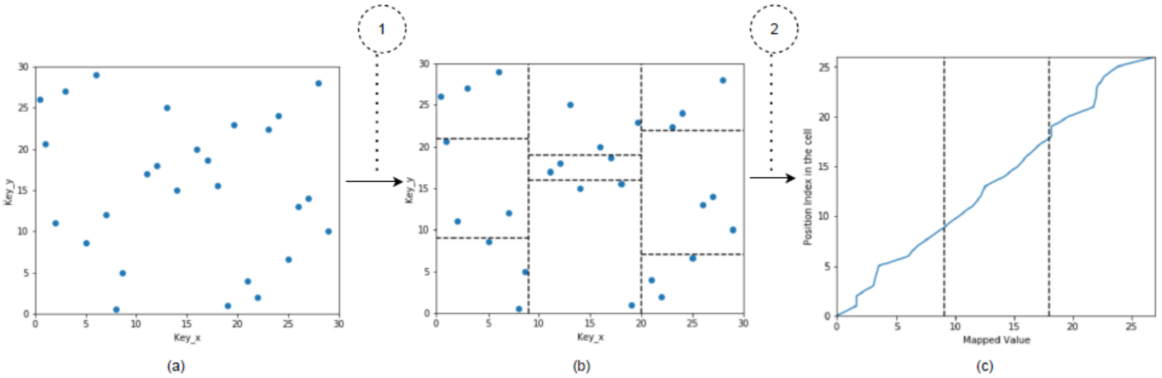

Figure 2.7: Mapping 2 dimensional key Values to one dimensional cdf
1) Generate grid cells, and apply Lebesgue Measure to each cell.
2) Sort key in each cell according to mapped value. Mapped values in consecutive cells are already sorted by mapping function definition. Plot the cdf of mapped values.

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. $\mathcal{F}_i(x) = \frac{f_i(m)}{\Psi}$.

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 $f_i(m) = 1$ 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.

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

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

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

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

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

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

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

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

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

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

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 perform the update along the direction of $s = -\boldsymbol{Y}^{-1}g$.

Show that Y is positive definite and explain why it matters

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

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

The iteration continues until L converges, i.e. the $|L^{(k)} - L^{(k-1)}| < \delta$ where δ is a pre-set hyperparameter.

3 Evaluation

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 $z^{(2)}(w^{(2)}(w^{(1)}x+b^{(1)})+b^{(2)}) = \max(w^{(2)}w^{(1)}x+w^{(2)}b^{(1)}+b^{(2)}, 0)$.

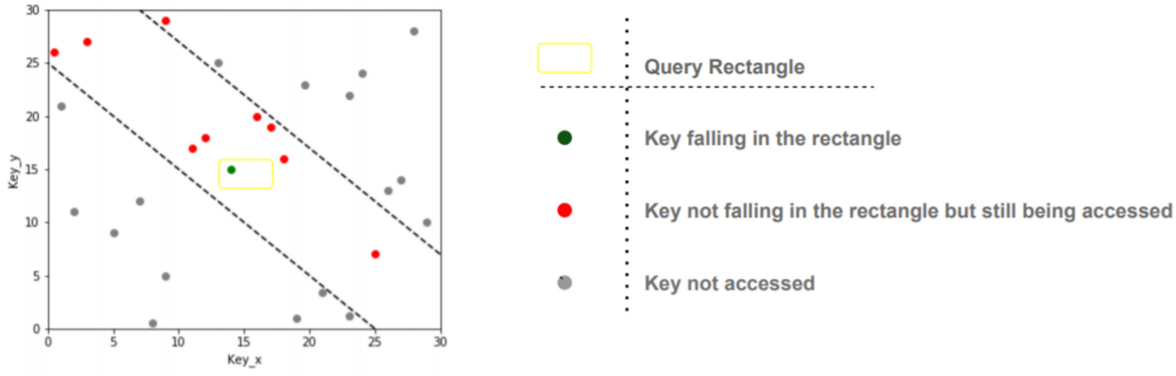

Figure 4.1: Baseline Method Limitation

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.

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.

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.