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)

B-tree and KD-tree are the two of the indexes used in 1-dimensional and 2-dimensional data. In this master project we implement them from scratch and analyse their complexity. Afterwards we introduce the learned indexes that replace these tree structures with a machine learning approach.

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 hand-engineered data structures and algorithms, such as B-Tree, KD-Tree, Hash Table, etc. These indexes have played an important role in databases and have been widely used in modern data management systems (DBMS). Despite their success, 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, the key can be used directly as an offset. With the key used as an offset, the values with the key can be retrieved in $\mathcal{O}(1)$ time complexity. Compared with B-Tree, which always takes $\mathcal{O}(\log n)$ time complexity for the same query. At the same time, by using the key as an offset directly, we do not need any extra overhead regarding memory space, where the B-Tree needs extra $\mathcal{O}(n)$ space complexity to save the tree.

From the above example, we found there are two promising advantages of learned indexes over hand-engineered indexes:

1. Learned indexes may be faster when performing queries, especially when the number of entries in the database are extremely huge.

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

We will explore and analyse these two advantages qualitatively in the *chapter 5*.

Nowadays, to leverage these two advantages, 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. This approach has been shown to be powerful and competitive compared with hand-engineered indexes, such as B-Tree.

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
$\mathcal{M}$	Polynomial Function

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 children possible in it's 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 an 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 will 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 parts.

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 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 nondecreasing 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

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.

Polynomial Regression

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

$$\begin{aligned} \mathrm{S}(\boldsymbol{\beta}) = ||\boldsymbol{y} - \boldsymbol{X\beta}|| &= (\boldsymbol{y} - \boldsymbol{X\beta})^T (\boldsymbol{y} - \boldsymbol{X\beta}) \\ &= \boldsymbol{y}^T \boldsymbol{y} - \boldsymbol{\beta}^T \boldsymbol{X}^T \boldsymbol{y} - \boldsymbol{y}^T \boldsymbol{X\beta} + \boldsymbol{\beta}^T \boldsymbol{X}^T \boldsymbol{X\beta} \end{aligned} \tag{2.1}$$

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

$$\mathrm{S}(\boldsymbol{\beta}) = \boldsymbol{y}^T \boldsymbol{y} - 2\boldsymbol{\beta}^T \boldsymbol{X}^T \boldsymbol{y} + \boldsymbol{\beta}^T \boldsymbol{X}^T \boldsymbol{X\beta} \tag{2.2}$$

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

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

By let it to be zero, we end up with

$$\begin{aligned} & -\boldsymbol{X}^T \boldsymbol{y} + (\boldsymbol{X}^T \boldsymbol{X})\boldsymbol{\beta} = 0 \\ \implies & \boldsymbol{\beta} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y} \end{aligned} \tag{2.4}$$

Fully Connected Neural Network

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.

Definitions

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 1: 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 ← 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 2: 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
```

2.2 Two Dimensional Data

2.2.1 KD-Tree

KD-Tree

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 use machine learning models through several steps and generate a learned index for spatial data to reduce the storage consumption and IO cost compared to existing indexes such as R-Tree.

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 cumulative distribution function (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.

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 partially monotonic function according to the borders of cells to map the data from $\mathbb{R}^d$ into $\mathbb{R}$.

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 equal number of cells. If a point (x,y)'s mapped value is larger than those of the keys stored in the first i cells, i.e. $M(x, y) > \sup \bigcup_{j=0}^{i-1} M(C_j)$, we store (x,y) in cell i. Subsequently, for a query rectangle qr = $[l_0, u_0) \times [l_1, u_1)$, we only need to predict i_1 and i_2, the indices of (l_0, l_1) and (u_0, u_1), respectively, scan the keys in i_2 - i_1 + 1 cells, and those keys that fall in the query rectangle qr .

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 M is a partially monotonic 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 $x_1 \leq y_1$
4. **Shard**.shard S is the preimage of an interval $[a, b) \subseteq [0, +1)$ under the mapping function M, i.e., $S = M^{-1}([a.b))$.
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 .

Training

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

1. The training dataset, notated as (X, Y) with entries notated as (x, y). X represents the two dimensional key values, and Y represents the corresponding data item value.
2. This number represents the number of cells(pages) into which the key space mapped values will be divided. It is an integer variable. Pages or cells will be used interchangeably in the next section to represents the subset of keys in a particular cell.

During training, sort all keys according to their mapped values, divide the keys into equal sized cells(pages), and store the mapped values of first and last key for each page into an array. For prediction, find the page corresponding to mapped value of query point, and scan this page sequentially.

Algorithm 3: 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
1  for $i \leftarrow 0$ to $len(x)$ do
2      x[i].mapped_value = x[i][0]+x[i][1]
3  end
4  Sort x based on x.mapped_value
5  Divide x into equal size pages according to num_of_cells
6  Store mapped value of first and last key for each page
7  for $i \leftarrow 0$ to num_of_cells do
8      denseArray[i].lower = first key in page i
9      denseArray[i].upper = last key in page i
10 end
```

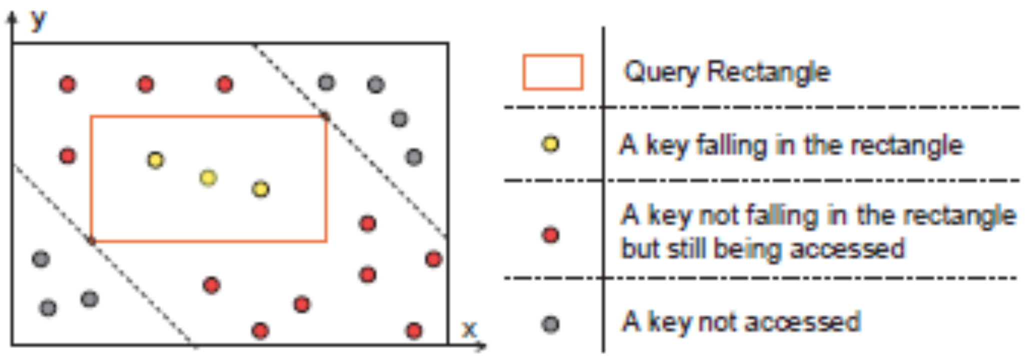
y
x
Query Rectangle
A key falling in the rectangle
A key not falling in the rectangle but still being accessed
A key not accessed

Figure 2.1: Baseline Method Limitation

Prediction

Algorithm 4: Prediction Algorithm for Lisa Baseline Model

```
input: x_test: Key; d: the denseArray
1 x_test.mapped_value = x_test[0]+x_test[1]
2 for $i \leftarrow 0$ to $len(denseArray)$ do
3     if $\mathcal{M}(x\_test) \in [d[i].lower, d[i].upper]$ then
4         Key is in Page i
5         break
6     end
7 end
8 Sequentially search for x_test in page j
```

Limitation

Prediction cost in baseline method consists of following two parts.

1. Search cost for the page which contains the key. If page size is small, this cost will be high as keys will be divided into larger number of pages
2. Scan cost of all the keys in a particular page to match the query point. If page size is large, number of pages will be smaller, number of keys per page will be higher, resulting in higher cost of sequential scan with in the page.

Consider the example in figure 2.1. 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 14 keys in the triangle.

Lisa Overview

Given a spatial dataset, we generate the mapping function M, the shard prediction function SP and a series of local models. Based on them, we build our index structure, LISA, to process

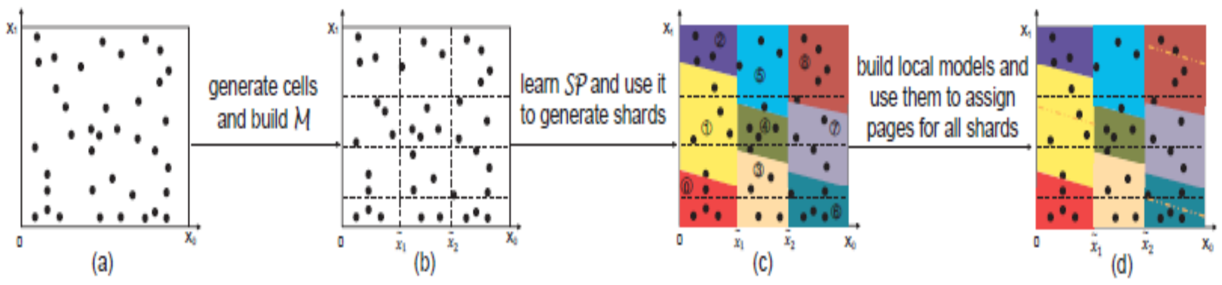
generate cells and build $\mathcal{M}$
learn $\mathcal{SP}$ and use it to generate shards
build local models and use them to assign pages for all shards
(a)
(b)
(c)
(d)

Figure 2.2: Lisa Framework

range query,KNN query and data updates. LISA consists of four parts: the representation of grid cells, the mapping function M, the shard prediction function SP, and the local models for all shards. As illustrated in the figure. the procedure of building LISA is composed of four parts.

As seen in the overview of LISA, there are four stages in building the LISA.

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.

Grid Cells Generation 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 and numbering the cells also along these axes. 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, i.e. mapped value of all keys in cell i will be less than mapped values of keys in cell j, if i <j. Consider the example shown in the figure 2.3 total number of keys are 18, and we decided to partition the key space into 9 cells, resulting in 2 keys per cell. To partition the key space, we first sort the keys values according to 1^{st} dimension, divide the keys into 3 cells each containing 6 keys. Then for each cell, we sort the keys again according to 2^{nd} dimension, and divide the keys in each cell into 3 new cells.

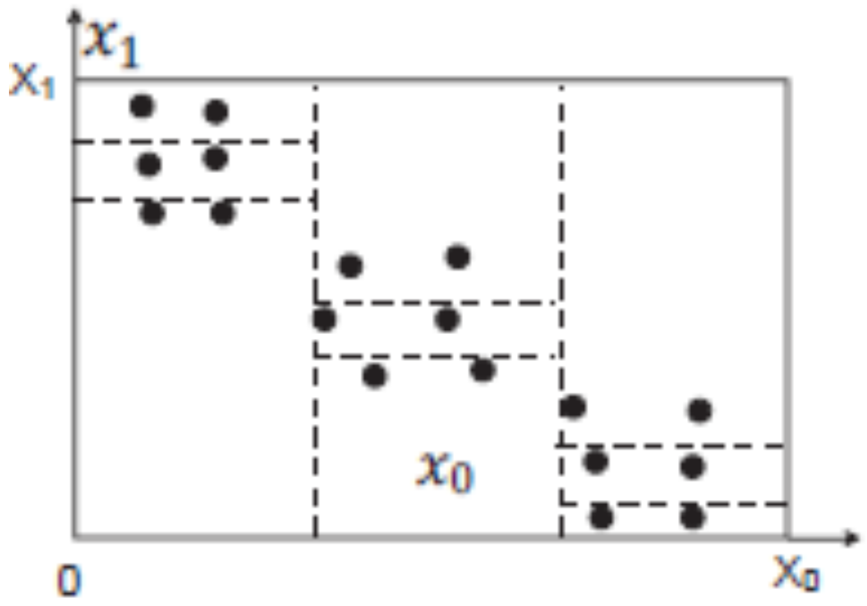

Figure 2.3: Cell Partition Strategy

Algorithm 5: Grid Cell Generation Algorithm for Lisa Method

```
   input: num_of_cells;x; y
1  trainset=[$(x, y); x \in \mathbb{R}^2; y \in \mathbb{R}$]
2  $keysPerPage = len(x)/num\_of\_cells$
3  Sort x based on first dimension x[:][0]]
4  In first for loop, divide the keys into equal size subsets
    based on first dimension
5  for $i \leftarrow 0$ to $\sqrt{(}num\_of\_cells)$ do
6      Store the 1st dimensional coordinates of first and last
        key for each cell.  Each such cell will contain
        $keysPerPage * sqrt(num\_of\_cells)$ keys
7  end
8  Sort keys in each cell based on 2nd dimension,x[:][1]
9  for $i \leftarrow 0$ to $\sqrt{num\_of\_cells}$ do
10     for $j \leftarrow 0$ to $\sqrt{(}num\_of\_cells)$ do
11         Store the 2nd dimensional coordinates of first and
            last key for each cell.
12     end
13 end
```

Mapping Function

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 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

$$f(x) = \begin{cases} b_0 + \alpha_0(x - \beta_0) & \beta_0 \leq x < \beta_1 \\ b_1 + \alpha_1(x - \beta_1) & \beta_1 \leq x < \beta_2 \\ \vdots & \\ b_\sigma + \alpha_\sigma(x - \beta_\sigma) & \beta_\sigma \leq x \end{cases} \tag{2.5}$$

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. (2.5) reduces to

$$f(x) = \begin{cases} \bar{\alpha} + \alpha_0(x - \beta_0) & \beta_0 \le x < \beta_1 \\ \bar{\alpha} + \alpha_0(x - \beta_0) + \alpha_1(x - \beta_1) & \beta_1 \le x < \beta_2 \\ \cdots \\ \bar{\alpha} + \alpha_0(x - \beta_0) + \alpha_1(x - \beta_1) + \cdots + \alpha_\sigma(x - \beta_\sigma) & \beta_\sigma \le x \end{cases} \tag{2.6}$$

Then to make Eq. (2.6) monotonically increasing, we only need to ensure that

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

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

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

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

Let $\boldsymbol{r} = \boldsymbol{A\alpha} - \boldsymbol{y}$, then we have

$$\begin{aligned} L(\boldsymbol{\alpha}, \boldsymbol{\beta}) = (\boldsymbol{y} - \boldsymbol{A\alpha})^T(\boldsymbol{y} - \boldsymbol{A\alpha}) &= \boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{A\alpha} + \boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{A\alpha} \\ &= \boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{y} + \boldsymbol{\alpha}^T\boldsymbol{A}^T\boldsymbol{A\alpha} \end{aligned} \tag{2.7}$$

and if we let

$$\begin{aligned} \frac{\partial L(\boldsymbol{\alpha}, \boldsymbol{\beta})}{\boldsymbol{\alpha}} &= 2\boldsymbol{A}^T\boldsymbol{A\beta} - 2\boldsymbol{A^T y} = 0 \\ &\implies \boldsymbol{\alpha} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{Ay} \end{aligned} \tag{2.8}$$

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

(2.9)

3 Evaluation

4 Insights and Findings

5 Conclusion

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.