Department of Informatics, University of Zürich

MSc Thesis

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

Abstract

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. **Traditional Indexes used in Database**. In this chapter, we analyse one of the most important traditional index: B-Tree. We go through its motivation, algorithms, advantages and disadvantages. Besides, we explore some other traditional indexes such as B+ Tree briefly.

3. **Baseline Learned Indexes**. In this chapter, we build our first learned index with fully connected neural networks. We demonstrate how it works, explore its properties and analyse its advantages and disadvantages.

4. **Recursive Model Index**. Proposed by T.Kraska [KBC+18], Recursive Model Index (RMI) is one of the most popular model used as learned index. In this chapter, we will explain its mechanism, demonstrate its algorithm and qualitatively analyse its pros and cons.

5. **2D Learned Index**. Until so far, we only work with 1-dimensional data. 2-D indexes is also important in many applications. For example, in a location query, where users want to find out all the stores in a certain rectangle, all stores are indexed by 2-D keys, i.e. the coordinates. In this chapter, we will explore how 2-D keys are indexed and arranged using a learned fashion.

6. **Learned Indexes with Convolutional Neural Network**. In this chapter, we discover the possibilities of using convolutional neural network (CNN) as database index.

1.1 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, we first take a closer look at how database index works.

2 Traditional Indexes used in Database

2.1 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.

2.1.1 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.

2.1.2 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

2.1.3 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

3 Baseline Learned Indexes

3.1 Baseline Models

Our baseline model is a two layer fully connected neural network. Before constructing the neural network, we will try to theoretically analyze the requirements in such a neural network serving as database index.

3.1.1 Neural Network Settings

The input (key) to our neural network is a scalar, and the expected output of the neural network is the position of the given input. We assume the following parameters in the neural network:

- The weight in the ith layer $w^{(i)}$.
- The bias in the ith layer $b^{(i)}$.
- The activation function after the ith layer $z^{(i)}$.

For example the output of the first layer will be $z^{(1)}(w^{(1)}x + b^{(1)})$.

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

3.1.3 Monotonicity

4 Recursive Model Index (RMI)

4.1 Introduction

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.

4.2 Definitions

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

1. **Node**. 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**. 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.

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.