Jon Ryfetten

The Next Generation Index Structures - Learned Indexes

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

Jon Ryfetten

The Next Generation Index Structures - Learned Indexes

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Index structures are today the answer when it comes to efficient data access. Recently, a new type of indexes called learned indexes surfaced, employing machine learning at the core. With very recent innovations, the indexes can have the potential to become the "de facto" standard and then be the answer to efficient data access. In 2017, the first learned index paper surfaced, which marked the beginning of the new research field within databases. For decades, we have seen indexes been optimized, tweaked, and updated for new hardware. Improvements such as LSM, bloom filters, and other new indexes have been developed over the years to improve indexes and optimize performance. All of the improvements and optimizations had one thing in common; They assumed nothing about the stored data and only slightly improved the performance over their competitors. Learned indexes are now about to change the situation completely. It makes a radical paradigm change in the way indexes are made by focusing on the data being stored instead of assuming a general distribution of data. It also makes extensive performance claims.

Recent publications of learned indexes, such as the ALEX and PGM index, improve the space occupancy by orders of magnitude (from gigabyte to megabyte) [1][2]. It can also improve query and update time by up to 71% [2] over a B-tree. Google recently measured over 50% increase in throughput and a significant increase in lookup time using a learned index in Bigtable [3]. In this thesis, we will take deep dive into the current landscape of learned indexes. Can learned indexes be the new "de facto" standard? We will look at the different approaches, the performance of learned indexes versus traditional indexes, and peek at the future direction for learned indexes.

Preface

The motivation for this research stemmed from a passion for AI and a passion for understanding how it is possible to store and access data efficiently. It is interesting to see the opportunities that are in AI and how humans can apply it. When I first heard about learned indexes, I was instantaneously fascinated. Sadly, when trying to find information about learned indexes, it was hard, and no literature that extracted information from the field existed. This master thesis is here to make it easier to understand the field of learned indexes.

I want to thank my supervisor Svein Erik Bratsberg for giving me the freedom to pursue the topic of AI optimized indexes and as an incredible resource along the way.

I hope you enjoy your reading.

Jon Ryfetten

Trondheim, June 9, 2021

Contents

Abstract . . . . **iii**
Preface . . . . **iv**
Contents . . . . **v**
Figures . . . . **viii**
Tables . . . . **xi**
1 Introduction **1**
1.1 Databases meets AI 1
1.2 Purpose 2
1.3 Goals and Research Question 2
1.4 Content of thesis 3
2 Background and related work **4**
2.1 Definitions 4
2.1.1 Structured and unstructured data 4
2.1.2 Clustered and unclustered index 4
2.1.3 B+-tree 4
2.1.4 Big data 5
2.1.5 Linear models 6
2.1.6 Piecewise linear approximation 7
2.1.7 Neural Network 8
2.1.8 Cumulative Distribution Function (CDF) 8
2.1.9 Memory fence 9
2.1.10 Cold and warm cache 9
2.1.11 Data skew 9
2.1.12 Database transaction 10
2.1.13 Write and read locks 10
2.1.14 Non-volatile memory (NVM) 10
2.2 Related work 10

2.2.1 LSM-tree . . . 10
2.2.2 ART - The Adaptive Radix Tree . . . 11
3 Learned Indexes . . . 12
3.1 How big data affects data requirements? . . . 12
3.2 How learned indexes work . . . 13
3.2.1 Cumulative Distribution Function . . . 14
3.3 Why so fast? . . . 15
3.4 Is learned indexes provably better than classic indexes? . . . 15
4 Approaches . . . 17
4.1 The RMI . . . 17
4.2 FITing-Tree . . . 19
4.3 LISA . . . 22
4.4 PGM . . . 23
4.5 ALEX . . . 26
4.6 Bourbon . . . 29
4.7 RadixSpline . . . 30
4.8 Tsunami . . . 32
4.9 Comparison . . . 34
4.9.1 Properties . . . 35
4.10 Complex models versus linear models . . . 37
4.11 Bottom-down versus top-down . . . 38
4.12 Secondary indexes . . . 38
5 Performance of learned indexes . . . 39
5.1 Practical example - Google Bigtable . . . 39
5.1.1 Evaluation . . . 40
5.2 Practical example - Learned indexes in DNA sequence analysis . . . 40
5.3 Difficulties when performance testing indexes . . . 41
5.3.1 Tradeoffs . . . 41
5.3.2 Optimizations . . . 42
5.3.3 Hardware specific optimizations . . . 42
5.3.4 Purpose when performance comparing indexes . . . 43
5.4 Performance comparison between indexes . . . 43
5.4.1 Search On Sorted Data Benchmark (SOSD) . . . 44
5.4.2 Extension of SOSD . . . 46
5.4.3 Why learned index structure perform well . . . 47

5.4.4 Multi-threading 48
5.4.5 Build times 48
5.5 Range search 49
5.6 Summary 49
6 Limitations and future work of learned indexes 50
6.1 Criticisms and skepticism of learned indexes 50
6.1.1 Learned index can not outperform tuned traditional data structures 51
6.1.2 Learned indexes comes from implicit assumptions 51
6.2 Limitations and future work 52
6.2.1 Author thoughts 53
6.2.2 Future ideas for research 55
6.3 New direction in the field of databases 55
7 Conclusion 57
Bibliography 59

Figures

2.1 An example of a B-tree with an order of three. Often 64, 128, 256 or more is used. Typically stores millions, or billions of items. . . . 5
2.2 An example of a linear model. Illustration from [6] 7
2.3 An example of a PLA function. The red line illustrate the generated PLA function of the function highlighted in blue. Illustration from [7] 7
2.4 An example of a CDF, where the distribution is in the range [0, 60] and uniform. Distributions in indexes has a different is very likely to not be uniform. Image from [8] 9

3.1 Illustration of a learned index. The goal is to predict the memory location given a key. The model predict the position with a given error. 14
3.2 The CDF from a series of different 64-bit datasets. Amzn is book sales, face is user ids, logn and norm are lognormal (0,2) and normal distribution respectively), osmc (uniformly sampled) is sampled locations and wiki is edit times at Wikipedia. Uden is dense interger and uspr is uniformly distributed spare integers. Image collected from "SOSD: A Benchmark for Learned Indexes"[16] 15

4.1 An example of the RMI index could be used with use of multiple stages. Model 1.x predicts the correct model 2.x and model 2.x predicts the correct 3.x. Model 3.x predicts the position. Illustration is extracted from [4] 18

4.2 The design of the FITing-index. In the example illustration, the index is clustered. Support for non-clustered index, is added by including an new layer, which is an additional "indirection layer". Essentially, an array of pointers. Illustration is extracted from [18] . 19
4.3 The process of creating the index. Illustration is extracted from [20] 23
4.4 An example of how the PGM index could look like. At each level, we search within a range of $[p-\varepsilon, p+\varepsilon]$, where p is the predicted position from the last model. Illustration is extracted from [2] . . . 25
4.5 Design of the ALEX index. The structure of the tree can dynamically adapt to the dataset. Illustration extracted from [1]. 26
4.6 Performance of ALEX compared to RMI [4], B+-trees and ART [10]. Illustration extracted from [1] 28
4.7 Example of the radix spline index. The most significant bits is highlighted, which is 101 (5). Then a search is performed in the spline point between the pointer and pointer + 1, which is highlighted in the illustration. Illustration is extracted from [26]. 30
4.8 Performance results for lookup-optimized index configurations. RS is short for RadixSpline. Illustration extracted from [26] 32
4.9 The illustration visualize the number of points that are required to scan using the different indexes. K-d tree is not optimized for the workload, and Flood is not optimized for the query skew and correlation. Tsunami has equally sized regions for the workload and adaptive to the query skew. The number of points to scan is then significantly less. Illustration extracted from [28] 33

5.1 Performance comparison between learned index (RMI) and the regular two-level index in Bigtable. Illustration extracted from [33] . . 40
5.2 Lookup results in nanoseconds. Table extracted from [29] 45
5.3 Performance counters. Illustration extracted from [29] 45
5.4 Pareto analysis of the indexes. Lookup time compared to space usage for learned indexes and traditional indexes. Illustration extracted from [22] 47
5.5 Multi-threading performance on the amzn dataset. With and without memory fence. Illustration extracted from [22]. 48

5.6 Build times of the different indexes, using the amazon dataset with different amount of keys. Log scale on the y-axis. Illustration extracted from [22]. 49

Tables

4.1 Properties of the learned indexes 36
4.2 Time complexity of the learned indexes based on details from papers. 37

Chapter 1

Introduction

The term learned indexes was coined in 2017 when the paper "The Case for Learned Index Structures" [4] surfaced. It is written by Tim Kraska (MIT) in collaboration with Google as an exploratory research paper. The paper opened a new field within databases, challenging a decade-old field of traditional indexes. By considering indexes as models, they were able to replace traditional indexes with machine learning models. These models learn the patterns in the data and enable the automatic synthesis of specialized index structures with low engineering cost, which creates the term learned index.

1.1 Databases meets AI

The paper showed how learned indexes could enhance, improve, and even replace traditional indexes such as the B-tree. They describe how indexes are just models to map keys to positions of records in the memory. Using a model with a combination of neural networks and linear models, they create an index that uses a fraction of memory compared to a B+-tree. The index also improves the lookup time by over three times in some cases. In this preliminary study, the index only supports read operations.

One of the simplest examples of why a learned index could outperform a B+-tree is when we have a fixed number of continuous integer keys that have to be stored. If the first key, 1, points to the first record. Then key, 2, points to the second record, and so on. A learned index could then, in theory, learn the mapping between and learns the function $f(x) = x$. The function could then be used to

directly calculate the memory location of the data, given a key. Therefore achieve a time complexity of $O(1)$. While a B+-tree would assume a general distribution and, therefore, always has the time complexity of $O(\log(n))$ for insert, delete, and update. The space complexity of the structure for a learned index is also $O(1)$, as it is just storing a learned function. It means we can dramatically reduce both the memory footprint of the index and increase the performance dramatically.

1.2 Purpose

Learned Indexes is one of the most promising index structures to replace the traditional B-tree. At the time of the invention of the B-tree, data were stored in magnetic tapes. A storage medium made in the late 1920s with pure sequential access. Reading an entire block of data at a time was, therefore, critical. Since then, we have seen an emergence of new hardware and a massive increase in data volume. In the original B-tree paper from 1971, they expect the B-tree to theoretically handle an index size of 1 500 000 items with the available hardware at the time [5]. The index would then be able to handle two operations per second. With today's hardware and optimizations to the B-tree, the tree could process over 50 000 000 operations in a second, and the B-trees can store billions of items. However, a small effort has been used to make the B-tree take advantage of the multi-core chips, Non-volatile Memory Express (NVMe) and flash disks, and Non-volatile Memory (NVM). With the rise of digitization, storing large quanta of data has never been more critical. Today, B-trees serve as a crucial component in systems such as filesystems and databases. Performance is crucial for these systems, yet, B-tree still is one of the best options.

Learned indexes serve as a fresh breeze as the next generation of index structures to replace B+-trees and other traditional indexes. Currently, there is a lack of a paper that compares the different learned indexes and describes the field in detail. The purpose of this thesis is to take a deep dive into the current landscape of learned indexes and synthesize a comprehensive article about learned indexes.

1.3 Goals and Research Question

Goal *Give a comprehensive picture of the current landscape of learned indexes*

Research question 1 (RQ1) *What are the different approaches today at achieving efficiently learned indexes?*

Research question 2 (RQ2) *How do learned indexes perform compared to traditional indexes?*
Research question 3 (RQ3) *What is the future direction of learned indexes?*

1.4 Content of thesis

In Chapter 2, we take a look at the current traditional index solutions and various backgrounds and definitions needed later. Chapter 3 introduces the fundamentals behind learned indexes. In Chapter 4 we explore how the different learned indexes is created, the approaches they use, properties for the indexes, and comments on drawbacks and performance. There is also a theoretical discussion on complex models versus simple models as learning components. In Chapter 5, we look at how learned indexes perform in real-world situations and benchmarks. Then in Chapter 6, we discuss the limits and skepticism against learned indexes. We also take a look at the direction of the indexes and what to expect in the future.

Chapter 2

Background and related work

2.1 Definitions

2.1.1 Structured and unstructured data

Structured data is data well defined, often in a table. It is usually highly organized and indexable. There exist a data model which describes the relationship between the entities. Unstructured data does not have any particular format, no data model, and stored in a native format. Examples of unstructured data could be surveillance pictures and newspaper text.

2.1.2 Clustered and unclustered index

Clustered versus unclustered indexes describes the relationship between keys and the position of data. Clustered indexes means that the data is stored in the index, such as MySQL InnoDB and Microsft SQL B+-trees. Unclustered index means that the data is stored a different place, such as on an external disk.

2.1.3 B+-tree

A B-tree [5] is a data structure that enables search, insertion, deletions, and sequential access in logarithmic time. Nowadays, it is one of the most used data structures in databases. We also find it in file systems and operating systems. The invention completely changed the way of storing data and enormously reduced the input/output (IO) for databases. It was invented by Bayer and McCreight nearly 50 years ago as an extension to the binary search tree. The difference between a binary search tree and a B-tree is that a binary search tree contains

two elements per page, while a B-tree could hold multiple elements per page. Having multiple elements per page improves performance as more elements are kept in the memory's cache. Reading an element from disk compared to RAM could take over four orders of magnitude slower than RAM access. B-trees are balanced search trees, which makes them high-performance. They have a time complexity of $\mathcal{O}(\log n)$ for both search, delete and insert.

An example of a B-tree can be seen in Figure 2.1. A B-tree contains nodes and leaves. Each node can have m children, where m is the order of the tree. Nodes have at least $m/2$ children. The nodes only contain keys and references to other nodes or leaves. A leaf contains keys that are connected to values. We use the node's keys to find the leaf we are searching after. When inserting or deleting the tree, we always keep the tree balanced, making the search time logarithmic. The nodes point to other nodes or leaves, while leaves have keys to values.

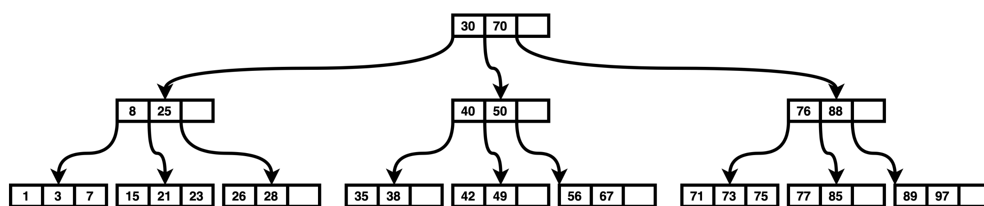


Figure 2.1: An example of a B-tree with an order of three. Often 64, 128, 256 or more is used. Typically stores millions, or billions of items.

There exist multiple B-tree types. The most important are the vanilla B-tree and the B+-tree. The B+-tree does not store any values in a node, and nodes neither contain values. It also includes links between leaves to speed up the traversal of the tree.

2.1.4 Big data

It refers to large or complex data on a scale that makes it impractical and impossible to deal with within traditional data-processing applications (e.g., DBMS). The speed at which the generated data is often also considered very high, but it is not a requirement to be categorized as big data. The concept itself of Big Data gained attraction after Doug Lanyey coined the definition of big data as the three V's. Volume, Velocity, and Variety.

Volume: Big data is enormous. Data is often collected from multiple sources, including devices such as the Internet of Things (IoT) that rapidly generate new data.

Velocity: The rate of inserts and updates is high. The high velocity creates new requirements, such as the ability to process the operations quickly. Sensors and IoT devices are examples of sources of data that quickly generate data and is driving the need to deal with the data in near real-time.

Variety: Data comes in many different types of formats, both structured and unstructured data. From voice, video, and images to numbers and text.

2.1.5 Linear models

Linear models are one of the simplest examples of a machine learning model. The goal of a linear model is to fit a straight line to a set of data and then attempt to model the relationship between two variables. An example could be age and average height. We can then predict the height by using the age. Since average height, given age, is not linearly, we would get an error. When creating the model, we can have different metrics on how we want to fit the line. Sometimes, we want to reduce the max distances. Other times, we want to reduce the average error. Once the line is fitted, we end up with two constants, a and b, which is used in the equation $f(x) = a + bx$. An example of a linear model is given in Figure 2.2

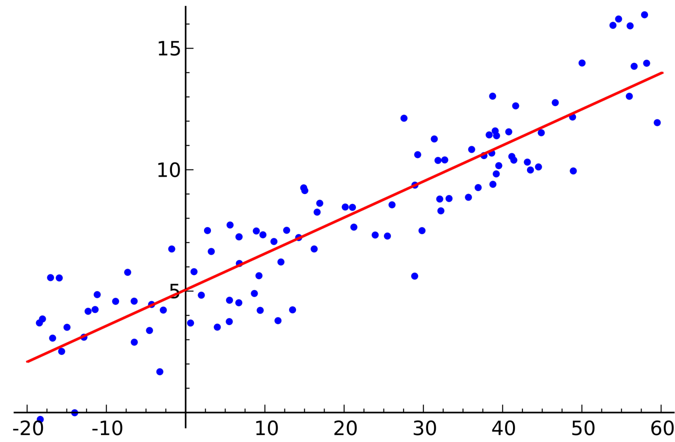


Figure 2.2: An example of a linear model. Illustration from [6]

2.1.6 Piecewise linear approximation

It is also called piecewise linear function and piecewise linear model. Formally, a function is defined as a collection of intervals, where each interval contains an affine function. More simply, it is a linear approximation to a function. The approximation is divided into multiple segments of linear models. An example is displayed in Figure 2.3

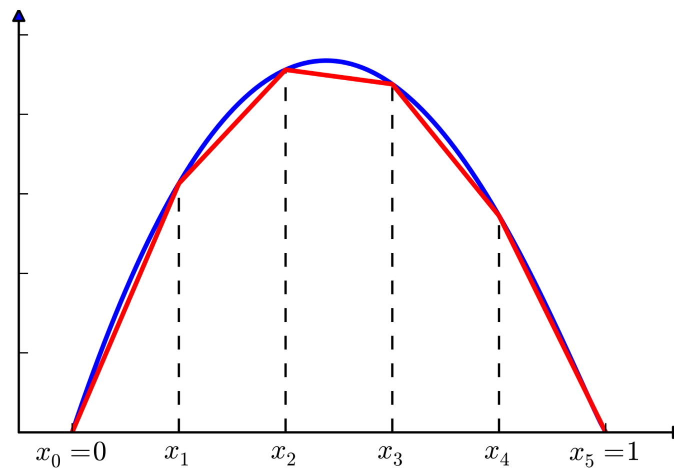


Figure 2.3: An example of a PLA function. The red line illustrate the generated PLA function of the function highlighted in blue. Illustration from [7]

2.1.7 Neural Network

A neural network is a biologically inspired machine learning method. A network is a set of connected nodes, where each node is called a neuron. A neuron process a signal and output a new signal. Such a signal is a decimal number (could be integers in some situations). The neuron itself is some non-linear function, and the output is the sum of all the input processed in this function. Typically, a neuron has a weight associated, which adjusts the importance of the signal. Often, the network is organized into several layers, where each layer performs the same transformations. The training of a neural network is done by initializing the neurons to specific values (could be random to a certain degree). Then we feed the network with data and compare the predicted output to the actual values from the training data. The error between the prediction and actual data is then back-propagated through the network (backward, from the output layer to the input layer). In the last decade, we saw an explosion in the use of neural networks. Primarily due to advances in hardware (GPU particularly), available training data, and new research within deep learning. Although, many of the building blocks for neural networks have been available for a long time.

2.1.8 Cumulative Distribution Function (CDF)

Used to specify distribution of multivariate random variables.

$$F_X(x) = P(X \leq x)$$

Where X is the real-value variable. F takes a value between 0 and 1, $\mathbb{R} \rightarrow [0, 1]$. The function describes the probability that the variable takes a value less than or equal to the given x-value. An example is displayed in Figure 2.4.

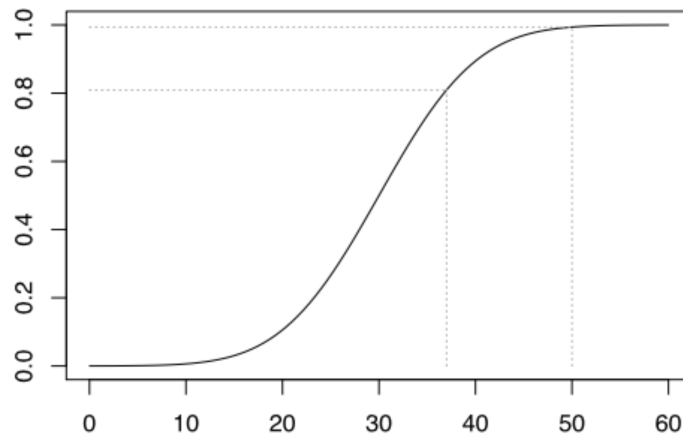


Figure 2.4: An example of a CDF, where the distribution is in the range [0, 60] and uniform. Distributions in indexes has a different is very likely to not be uniform. Image from [8]

2.1.9 Memory fence

Also known as a memory barrier. On modern CPUs and compilers, instructions may be reordered after the program is written. It is often done to improve pipelining, overall computation and memory access. Enabling the memory barrier prevents the compiler and/or CPU from performing operations issued before the barrier, and operations after the barrier will be performed after the barrier, often used in concurrent programs to prevent out-of-order execution.

2.1.10 Cold and warm cache

Originally an analogy to a cold and warm engine. A cold cache does not have data and does not provide a speedup (data must be moved to cache). A warm cache contains data, and therefore could potentially provide a speedup.

2.1.11 Data skew

Data skew means that the distribution of data is uneven or asymmetric. In the context of indexes, data skew could also be used about the keys. Bits distribution in the keys could be uneven or asymmetric. The workload could also have a skew, such as that some queries is executed more often.

2.1.12 Database transaction

Transactions in databases represent a change to the database. They make a set of operations independent of other operations ongoing in the database. Transactions provide reliability and isolation. They are atomic; they either fail or complete entirely. They also keep the database in a valid state upon completion (consistent), do not interfere with other transactions (isolated), and when a transaction is completed, it remains complete, even if a power loss or crash (durable). These four properties are often referred to with the acronym ACID.

2.1.13 Write and read locks

They are used for concurrency, where a file or data is "locked". When a process tries to access a file or data that is locked, it needs to wait until the lock has been unlocked before it gets access to read or modify the data. Locks are often used for transactions in database systems. Locks protocols guarantee that transactions produce the same output, even when they are executed in an overlapping time-space.

2.1.14 Non-volatile memory (NVM)

Memory that is stored even after the power has been shut down. Many different types have been deployed, from magnetic tape to spinning disk and flash memory. In the last decade, we have seen the grown of NVMe, a faster type of memory. Most recently, a non-volatile dual in-line memory module (NVDIMM). A random-access memory (RAM) for computers. First products are expected to be on the market from 2021. The advantage of NVM RAM over volatile RAM is that it enables high performance and high recovery when storing the database entirely in memory.

2.2 Related work

2.2.1 LSM-tree

LSM-tree was invented in 1996 by Patrick O'Neil, introduced to the world with the paper "The Log-Structured Merge-Tree (LSM-Tree) and is a competitor to the traditional B-tree" [9]. The LSM-tree makes itself attractive by providing a worst-case time complexity of $\mathcal{O}(1)$ for insertion operations but has a worst-case time complexity of $\mathcal{O}(n)$ for search and delete. It makes the LSM-tree very attractive

for write-intensive applications.

In the original paper, a two-level LSM tree was proposed. The tree divides the data into two different physical locations, in-memory and on disk. New records are stored in-memory. Once a set of entries exceeds a certain size, the set of entries is moved over to disk by merging it. It makes it possible to employ an index structure optimized for in-memory for the first subset of the data and use a different index optimized, for example, for disk for the next subset. In an LSM tree, each of the subsets is called levels. Sometimes we use more levels than two levels.

2.2.2 ART - The Adaptive Radix Tree

ART, presented in the paper "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases" [10], is a modern high-performance index for databases. As the title of the paper mentions, the index is an in-memory index for main-memory databases. Main memory databases have grown in popularity since most databases today fit into RAM. In main memory databases, a critical bottleneck is the index performance. ART is an adaptive radix tree for efficient indexing in main memory, and at the same time, very space-efficient. Usually, radix trees suffer from excessive worst-case space consumption. ART solves this by adaptively choosing compact and efficient data structures for internal nodes in the tree. It adapts the representation of every individual node locally. ART delivers very high performance on multi-threading and scales about 1.6x-1.7x using multiple threads.

Chapter 3

Learned Indexes

For decades, the CPU power has rapidly increased. From 1965 to 1975, the number of transistors doubled every year. Since 1975, it doubled every second year and was known as Moore's Law. Just until recently, in 2016, CPU performance was rapidly increasing. Today, Moore Law is dead, and we can not expect to see an ever-growing increase in computing performance. In contrast, in the last few decades, we have seen an explosion in the amounts of data generated, which not seems to stop. Big data has become a term commonly known. Every day, the amount of data required to process increases on a much larger scale than the available computer power [11]. With all this data, we try to crunch out every bit of useful information, which requires us to use sophisticated tools to analyze the data. We need to index data, which again increases the space requirements. Indexes in OLTP workloads can consume up to 55% [12] of the memory in state-of-the-art in-memory DBMS. One of the approaches to deal with this problem is learned indexes. It promises to dramatically improve the performance of data processing in terms of CPU cycles and, at the same time is also reduce space consumption with orders of magnitudes.

3.1 How big data affects data requirements?

While we have seen the death of Moore's law, at the same time, we have seen an explosion of new research and effort put into the field of AI and Machine Learning. Not surprisingly, we have seen a series of attempts to apply the inventions from Machine Learning in databases. One of them is learned indexes, but we have also seen query optimizations [13] [14] and system tuning [15] using AI. Learned in-

dexes made a controversial view on indexes by considering the data distribution as one of the key elements in developing an index. The results seen from learned indexes have surprised the database community and shown some solid results in a series of examples by significantly reducing the space consumption with magnitudes of orders while also improving performance.

In many industries today, data has become an essential resource used in decision-making. Some also call themself data-driven companies. These companies often process vast amounts of data to gain insight from the data. Devices such as the Internet of Things are one of the factors driving this data, finical data (stocks, time-series, tweets, etc.) and general data generated on the internet drive the need to process vast amounts of data. This is where learned indexes can be a game-changer, with the ability to reduce storage requirements with magnitudes of order and at the same time provide a significant performance improvements.

3.2 How learned indexes work

In the paper "The Case for Learned Index Structures" [4], they introduced the concept that traditional indexes (with sorting and range queries) are just models. These models assume nothing about the data distribution. They introduce the idea that a model can map a key to a certain location in a sorted array. We can replace the index with models such as neural networks by considering the index as a model that predicts the position with a given error. In reality, models such as neural networks have shown to be a bad choice [1] [4]. A rather successful choice is, for example, is a piecewise linear regression model. The key idea for the model is to learn the sort order of the data and structure of the data to predict the location efficiently.

A simple yet surprisingly powerful model for a given case is a linear regression model such as $a + bx$ to predict the location. It could, for example, be the case if you have an index for identification numbers, where the identification numbers are added by 1 for every new insert. Then it is possible to access the memory location directly by computing the function $f(x) = a + bx$, where x is the key. We do not need to traverse tree index. An illustration of a learned index is given in Figure 3.1.

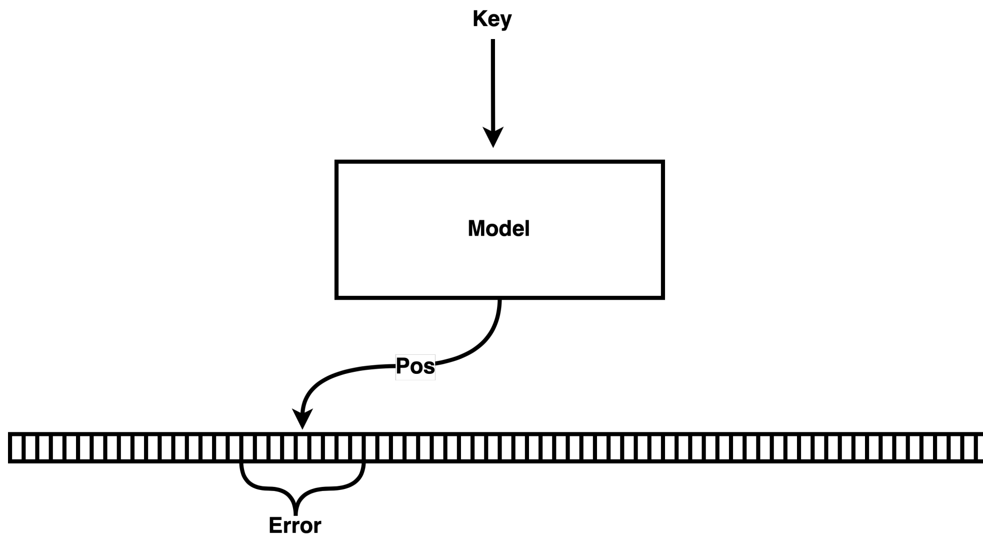


Figure 3.1: Illustration of a learned index. The goal is to predict the memory location given a key. The model predict the position with a given error.

3.2.1 Cumulative Distribution Function

Since the data is a sorted array, the data could be considered as a CDF, where the function estimates the likelihood of observing a key smaller or equal to the lookup key. Effectively, a model in terms of an index is approximating a Cumulative Distribution Function (CDF) with the following formula: $p = F(Key) * N$ where p is the position estimate, $F(Key)$ is the estimated CDF, and N is the total number of keys. Ultimately, this means that indexing does learn the data distribution and that traditional indexes such as B-trees learn data distributions. The difference is that traditional indexes learns the distribution, but does not take advantage of it, by assuming a general distribution.

CDFs come in all shapes, and therefore, the model that learns the data distribution must handle a wide range of different shapes. In Figure 3.2, a series of different CDF is displayed to give insight into how the CDF could look like.

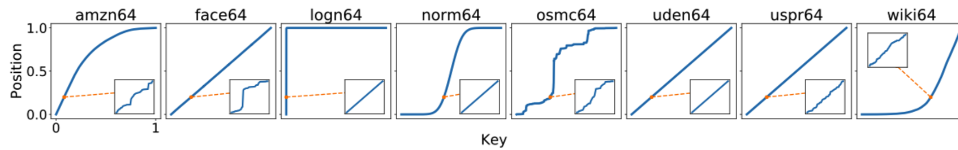


Figure 3.2: The CDF from a series of different 64-bit datasets. Amzn is book sales, face is user ids, logn and norm are lognormal (0,2) and normal distribution respectively), osmc (uniformly sampled) is sampled locations and wiki is edit times at Wikipedia. Uden is dense interger and uspr is uniformly distributed spare integers. Image collected from "SOSD: A Benchmark for Learned Indexes"[16]

3.3 Why so fast?

Learned Indexes are data-aware indexes that aim to learn how to create an efficient index structure based on the data. In some of the ideal cases, such as a running number in a fixed interval, e.g. [1, 1 000 000 000], the memory location of the data can be given by the pattern of the data, and no index structure is needed. In general, the goal of learned indexes is to take advantage of the data distribution and find a mapping between the data and the memory location.

In most cases, there exist a pattern in the data that could be detecte extracted from the dataset. Depending on the degree of the patterns detected, we can reduce the amount of needed index structure which will significantly improve the access time and reduce the disk footprint. In the example above of the running number, we would go from the time complexity of $O(logn)$ to $O(1)$ for both access time (read, insert, update and delete) and disk footprint.

3.4 Is learned indexes provably better than classic indexes?

Learned Indexes is a complicated topic when it comes to performance and does have its limitations. While B-trees is a well-understood data structure, highly tested and used, learned index is a new competitor. Until recently, there existed no evidence that learned indexes are provably better than classic indexes (e.g., B+-trees). Recently, the first proof that learned indexes are provably better than traditional indexes was released [17]. They proved that the particular learned index PGM-index [2] answers queries as fast as a B+-tree while improving its space

to $O(n/B^2)$, where B is the number keys possible to fit in on page in a disk. In comparison, a B+-tree, typically takes $\theta(n/B)$. They also constitute that learned indexes are an effective and robust choice for modern applications on big data, where space compression and query efficiency are mandatory.

Chapter 4

Approaches

Since the first learned index, RMI (2017), we have seen various approaches to improve learned indexes. In the beginning, learned indexes were based on neural networks as a first layer and the linear regression models as the next layers. We have later seen a switch to the use of piecewise linear regression models. Support for updates, removal, and insert has also been implemented. Lately, we have also seen learned index in spatial indexes and in LSM-trees.

4.1 The RMI

RMI was developed in the first paper on learned indexes. The model is based on the observation that reducing the prediction error is easy, but the difficulty is building models with high accuracy for the last-mile search. They proposed a Recursive Model Index (RMI), a hierarchy of models. At each stage, the model predicts to a new next model until the final model predicts the position. It will efficiently divide complex sub-ranges into smaller sub-ranges to make use of more specialized models. Also, each model does not limit how many records it covers, unlike B-trees. Another benefit is that there is no search process required in between stages. This approach opens up an exciting space for learned indexes, TPU/GPU acceleration. The entire index could be represented as sparse matrix multiplication, as there is no search between stages.

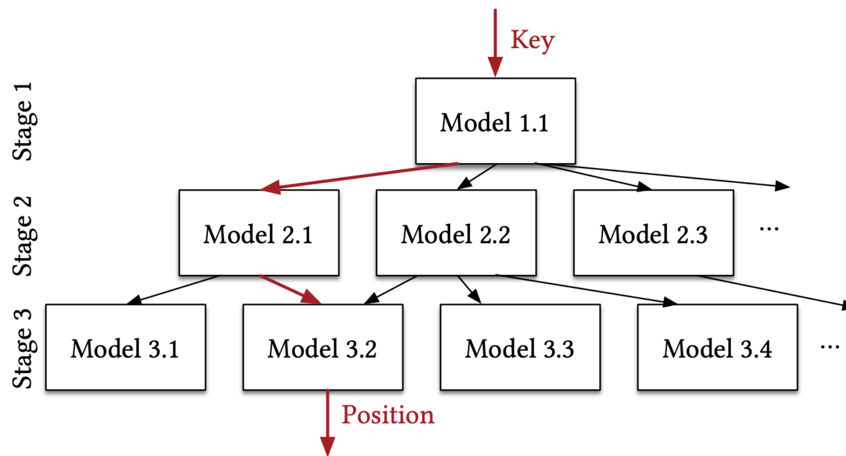


Figure 4.1: An example of the RMI index could be used with use of multiple stages. Model 1.x predicts the correct model 2.x and model 2.x predicts the correct 3.x. Model 3.x predicts the position. Illustration is extracted from [4]

It is important to empathize that the RMI does not have to be a tree. Figure 4.1 is just one example of an RMI index. Using the RMI, we can also choose to create a custom model for each different model. E.g., the first model (model 1.1 in the example) could be a Neural Network, and the rest of the models can be linear models. At the end of the prediction (last stage), the model will predict a minimum and maximum error for the binary search. It is also possible to create a hybrid model with a B-Tree, which allows us to bound the worst-case performance to a B-Tree ($O(log(n))$).

Performance

Based on an optimized configuration of the RMI index and the B-tree, we can measure a performance boost of up to 2x times on particular weblogs and 3x times on a map dataset. If we instead tune the index for low memory footprint, we can see a decrease from 12.92MB to 0.15MB on web data and 13.11MB to 0.15MB on map data, where the performance of the RMI is on par or better compared to the B-tree. For map data tuned with a low memory footprint, the index delivered 2.7x better performance. However, it should be noted that there exist critics on the performance results, claiming the index performs especial good on the dataset chosen. In Chapter 5, we will take a deep dive to compare the performance of RMI

against other learned index approaches and also traditional indexes.

Drawbacks

The index does not support updates, inserts, or delete operations, making it a read-only index. The model also requires some time to be trained. Although in the paper, they referred to the training time as relatively fast. In theory, the index should support multi-threading or GPU/TPUs acceleration, but no experiments have been executed to test the performance of GPU/TPC acceleration in this exploratory research on learned indexes.

4.2 FITing-Tree

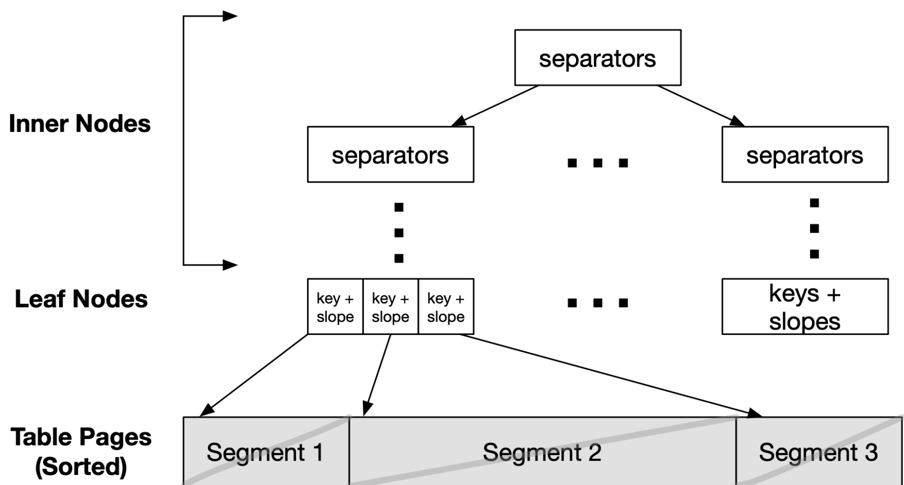


Figure 4.2: The design of the FITing-index. In the example illustration, the index is clustered. Support for non-clustered index, is added by including an new layer, which is an additional "indirection layer". Essentially, an array of pointers. Illustration is extracted from [18]

FITing-Tree [18] is a paper that surfaced in 2018, co-authored with the primary author of the original paper on learned indexes. FITing-Tree uses a different approach to achieving a learned index. The index is using piecewise linear functions, which includes a bounded error specified when constructing the tree. The error is tunable, which makes it possible to set a tradeoff between space consumption

and performance. Like a series of learned indexes, the FITing-Tree index also can reduce space consumption by orders of magnitude. In contrast to RMI, FITing-Tree supports efficient insert operations. A significant feature of the index is that it supports paging, making it possible to store data in different regions on a disk. Usually, learned indexes store the data in a contiguous memory region. With the support for paging, they also support unclustered and clustered mode.

The main goal with the FITing-Tree is to reduce space consumption by providing the same performance or better than traditional full and fixed-page indexes. To achieve this, FITing-Tree approximates the CDF function using a piecewise linear approximation. Choosing polynomial approximation functions would give better precision, but linear functions are significantly less expensive. One of the main ideas in FITing-Tree is to define an error between the CDF and the approximation, then use this error to define segments. Within a segment, there is no larger error than a specified error threshold. The error between the approximation and the CDF is defined $error = max(|pred_pos(k) - true_pos(k)|) \forall k \in keys$, where $pred_pos$ is the predicted position and $true_pos$ is the actual position. Often, when making approximations, it is normal to use the least square error, but the least square error does not guarantee a certain error and is therefore not used here.

The design of the FITing-Tree is illustrated in Figure 4.2. There are three different layers in the design. The first one, the inner nodes, is the same as a B+-tree by default. The user of the index can choose to use any other index data structures. It means it is possible to use other indexes optimized, for example, for read-only, which could improve the performance. When we reach the leaf nodes, we need to calculate the approximate position. The leaf nodes contain pointers to a segment, a slope of the segment, and the distance to the starting key. Once the approximate position has been calculated using this linear regression, a local search has to start. It could, for example, be a binary or linear search. A key advantage with the segments, compared to, for example, nodes in a B+-tree, is that that the data is partitioned into variable-sized segments, which means each segment covers a variable number of pages. Support for unclustered (non-clustered) index, or secondary index, is included by adding a fourth layer to the design. They call this additional layer the "indirection layer" and "Key Pages". The layer is an array of pointers. Although adding a new layer to support the secondary index adds over-

head, the overhead is significantly lower than for a non-clustered B+-tree due to fewer leaf and internal nodes in the FITing-Tree.

Updates and retraining

The segmentation algorithm in FITing-Tree is based on an existing algorithm for linear piecewise segmentation, Feasible Space Window (FSW) [19]. The algorithm does not select the optimal amount of segments, but does perform faster. The reason to select this particular segmentation algorithm is to achieve the particular performance properties they want in FITing-Tree, such as fast insert, update, and low space consumption. When they evaluated the algorithm with an optimal algorithm with a runtime of $O(n^2)$, compared their algorithm with time complexity of $O(n)$, the number of generated segments was relative to the optimal algorithm.

Performance

FITing-tree is able to navigate between using a particular lookup latency, given a configured threshold (in ns), or a particular storage budget (in MB). Evaluations show that the index performs comparable to other full index structures but consumes less space by orders of magnitude.

Drawbacks

FITing-tree is a simple yet elegant learned index. Sadly, no open-source implementation exist, which makes it hard to compare the performance to other learned indexes. A disadvantage with the design is that in-place inserts can cause large key shifts if the data is linear or large segments of the data are linear. One of the ideal datasets for a learned index is a linear dataset, as described in Chapter 3, and the FITing-tree adds here an unfortunate overhead. FITing-Tree addresses the issue by implementing a second insert strategy called delta inserts. Another drawback is that if we compare the index to a B+-tree, FITing-Tree is not able to handle the same write load.

Summary

Overall, the index can achieve performance on par with traditional index structures but can reduce storage footprint by orders of magnitude. It is an excellent

example of how learned indexes could be used for index compression. It also shows how existing indexes could be used in learned indexes. The index also relies on existing work done within functions approximation (segment algorithms). It also puts a second view on how learned indexes could be used for time-series data to leverage trends in the data, although it does not explicitly predict new inserts using deep analysis.

4.3 LISA

LISA is likely the first fully-fledged learned index for spatial data [20] and is an acronym for Learned Index structure for Spatial dAta. For a long time, R-trees have been the most popular choice for storing spatial data in a database. Lately, the R-tree has been challenged by increased data, also known as big data. For massive datasets, the R-tree index could today use more space than the dataset itself. In combination with rapid updates and inserts in data and the requirements to keep the tree fresh, the authors of LISA claim that R-tree and its variants fall short for indexing big spatial data. They, therefore, present LISA, a novel learned index structure for disk-resident spatial data. The index is the first learned index for spatial data that support KNN and data updates. Previously, a learned index ZM [21] was introduced that is a read-only index. The goal with LISA is a new index that aims to improve the performance and also be a replacement for R-trees by providing a fully-fledged index for spatial data

The index is created mainly by four steps, based on using shards. They define a shard to be a preimage of an interval, given a specific mapping function. The first step is to create grid cells, and then the second step is creating a mapping function, then a shard prediction function, SP. The last part is to create local models for all of the shards. The process is illustrated in Figure 4.3. In the first part, when the grid cells are generated, the goal is to cover every part of the space evenly. Then in the mapping process, a mapping function is created, which maps the spatial keys to a 1-dimensional space. Then the interesting parts start, which is the shard prediction function. The function is the learned component that makes the index a learned index. The shard prediction function has the mapped key as input. The output is the shard id. The model they train for the function could be considered as a regression model. The argument for use of the regression model is that it is

hard to implement a neural network. Instead, they propose to use a monotonic piecewise linear function. In the last step, a local model is created to address the issue of more keys than a single page. The model keeps track of the pages that overlap with the shard and provides a lower- and upper bound to search for a mapped value.

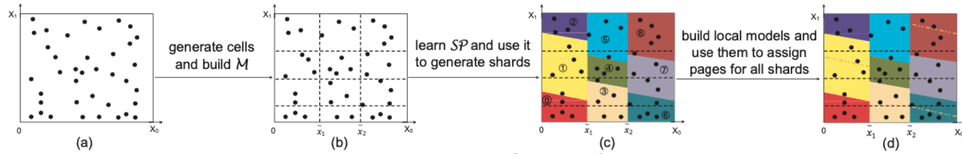


Figure 4.3: The process of creating the index. Illustration is extracted from [20]

Performance

LISA performance is not as impressive as other learned indexes and slightly improves the performance space usage R-tree. It does, however, reduce the response time and IO. For IO, it significantly reduces the number of required calls. Overall, the index slightly improves the performance over R-tree. There is also a test where the index outperforms the KNN-performance of R-trees, with a speedup of 100%.

Drawbacks

Although LISA delivers promising KNN-performance results, the index still suffers on overall performance and delivers slightly better performance than state-of-art. The index is also missing support for spatial joins and closest pairs query. Propper benchmarking of the index against other state-of-art indexes is still missing, using a broad range of datasets. More research is also at optimizing the learning part of the index by using a more optimized model.

In general, LISA is a promising step towards implementing learned indexes for spatial data.

4.4 PGM

Piecewise Geometric Model index [2] (PGM index) was introduced in 2020. It shares some of the design similarities with the FITing-Tree, where both consider-

ing the Piecewise Linear Approximation problem. However, it addresses several of the issues with FITing-Tree. The PGM index comes with three different variations. The first index variant is a distribution-aware, the second compress its succinct space footprint, and the third is a multicriteria variant that auto-tunes itself. The user has to select the variant it wants to use, and only one. The index being distribution aware is an exciting feature, which means it can adapt to the distribution of the queries. Along with the interesting variants, the PGM index makes substantial performance claims. For example, the index matching the same query performance of a cache optimized static B+-tree but improves space usage by 83x. PGM uses 1140x less space in a dynamic setting and improved query and updates time performance by 71%. It is also possible to tune the index using one variable, as seen with the FITing-Tree index. The variable is a max error tolerance. Compared to FITing-Tree and RMI does not have any fallback structures, which makes it a, as they claim, fully-learned index.

The authors claim a uniform improvement of performance compared to RMI in query and space occupancy, with 15x faster construction. The authors of RMI have later stressed that this is false and that the authors of the PGM index used an unoptimized index of RMI that only included linear models [22]. At the time, RMI was not released as a publicly available source code, and therefore, the authors of PGM had to implement the index based on the details from the RMI paper. A note from the author of this paper is that the RMI paper clearly states the use of neural networks in the first layer. For a better comparison between the performance of the indexes, see Chapter 5, or [22] for more details.

The general idea behind PGM is to take an array of items, then create segments with a maximum error, ε. We repeat this process until all items is covered by a segments. Now we have an array of segments. Then the first key of each segment is the item in the new array. We repeat the process of creating segments until we have one segment left. The process is done bottom-up. We start with the items and then create the structure one level up at a time. See Figure 4.4 for a visual image of how the structure looks like. Each segment contains a linear model that estimates the position of an element (the rank) with a maximum error ε. Compared to B+-Tree and FITing, no search cost grows with the node size. Instead, the nodes can be considered as routing tables and is a fixed size independent of the

node size. To find the optimal number of segments, PGM uses an algorithm that uses $O(n)$ optimal time, and space [23]. Compared to the sub-optimal number of indexes for the FITing-Tree, the use of this algorithm could reduce the space consumption by up to 67% [2].

Inserts are implemented with two different approaches in the PGM index. The first case is when items can be added to the last segment. This happens when the new item has the largest rank (e.g., time-series data). We can then add the item in constant time $O(1)$, given that the segment still preserving the ε guarantee. While, for general, insert, the time complexity is $O(log n)$. A deep dive into the details of how the index is created is described in the paper [2].

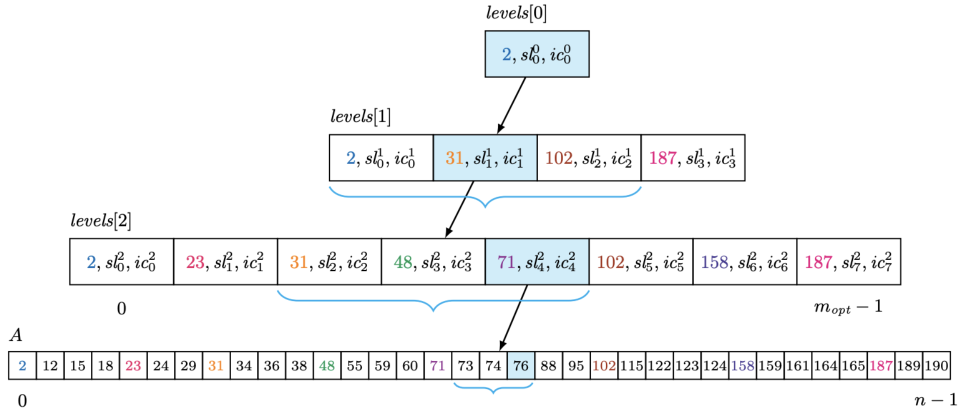


Figure 4.4: An example of how the PGM index could look like. At each level, we search within a range of $[p-\varepsilon, p+\varepsilon]$, where p is the predicted position from the last model. Illustration is extracted from [2]

Performance

The index delivers strong performance results and uses a fraction of the memory of a traditional index. The index supports different variants with different performance characteristics. Compared to traditional indexes, they measured memory reduction in up to 4 orders of magnitude while achieving the same or better query performance using the space-optimized variant. They were also able to improve the latency over the B+-tree with the dynamic PGM variant, with an improvement between 13% to 71%.

Drawbacks

No implementation of the distribution-aware variant of PGM is available. Another disadvantage is that PGM is built bottom-up, which is claimed and verified by the authors of RMI, is less effective than building it top-down.

4.5 ALEX

Summer of 2020, the ALEX index [1] was released. A paper from MIT and Microsoft. The goal of the index is to add updates, insert, short-range queries, and deletions and base it on the RMI index. They claim to beat the original RMI index by up to 2.2x on performance with up to 15x smaller index size. Compared to B+-trees, ALEX beats by up 4.1x and never performing worse and up to 2000x smaller index size.

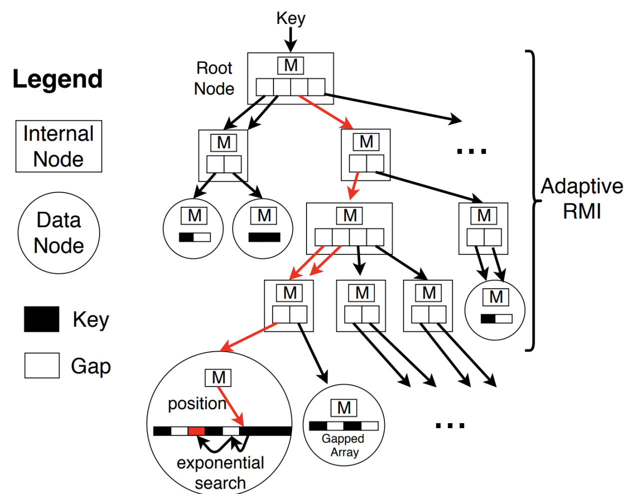


Figure 4.5: Design of the ALEX index. The structure of the tree can dynamically adapt to the dataset. Illustration extracted from [1].

ALEX employs a cost model that predicts lookup and inserts operation latency and uses it to adapt the RMI structure. It makes the RMI structure dynamic, in contrast to the RMI index, where the structure was static. Compared to the RMI index, ALEX uses a similar design and extracts many ideas from a B+-tree. Like a B+-tree, ALEX uses per node leaf. It allows ALEX to expand and split dynamically. Another critical element ALEX includes is the Gapped Array, which improves the search and inserts time in the arrays in the leaf nodes. The goal of ALEX was

to achieve competitive insert time with B+-tree while improving the lookup time over the B+-tree, but also RMI. It also aims to build an index that uses less storage space than RMI.

ALEX achieves the goals by implementing the design like a B+-tree with linear cost models at each node. The cost model is used to adjust the height of the B+-tree and potentially split, expand or retrain the parts of the tree. The tree is built with two different node types, internal nodes and data nodes, where data nodes are leaf nodes. An illustration of the design is displayed in Figure 4.5. Each of the nodes has a linear model which predicts a position. In the internal nodes, the model point to an array of pointers, just like we have in a B+-tree, but the model computes the position in the array of pointers. The number of pointers in each internal node can vary, as the goal of ALEX is to have internal nodes where the distribution is relatively linear. In each of the data nodes, we find a Gapped Array, which essentially is an array with strategically placed space between the items. When a lookup finds a leaf node, a search will begin. In contrast to B+-tree, ALEX uses exponential search over binary search. Also, in contrast to RMI, ALEX places items where the model in the data node predicts the item should be, while RMI does not change the position of records.

To search for an item in the index, we can simply compute the position at each internal node until we reach the data node (leaf node), as the internal node has perfect accuracy. The search begins in the data node, where the item's location is predicted, but we have to execute an exponential search in this last level to deal with an error in the prediction. The prediction of the model in the data node does not contain an error bound. If we want to insert an item, it becomes slightly more complex. We compute the right data node in the same way as with lookup, but it could either be full or not once we reach the data node. If it is full, the cost model is used to either expand the node and retrain or split it either sideways or downwards. The cost model is used to find the right decision, and we can insert the item in a non-full data node. To insert the item in a non-full data node, the model in the node predicts the location. Then we need to ensure we still maintain sorting order in the node and use exponential search to find the correct position. If the location we find is a gap, we can simply insert it. If it is not a gap, we need to shift the items by position in the direction of the closest gap.

ALEX also implements a bulk load algorithm, where the index is grown from the root node. For each node, a fanout tree is built locally. Each node in the fanout tree represents a possible child of a node in the ALEX tree.

Performance

ALEX has a time to traverse to a leaf of $O(log_m p)$, where m is the maximum node size for both internal nodes. p is the minimum number of partitions for the index, given the size of the index. Search in the data node (leaf node) has a worst case of $O(log m)$. For search, the worst case is $O(m)$, while the authors expect $O(log m)$ for most cases.

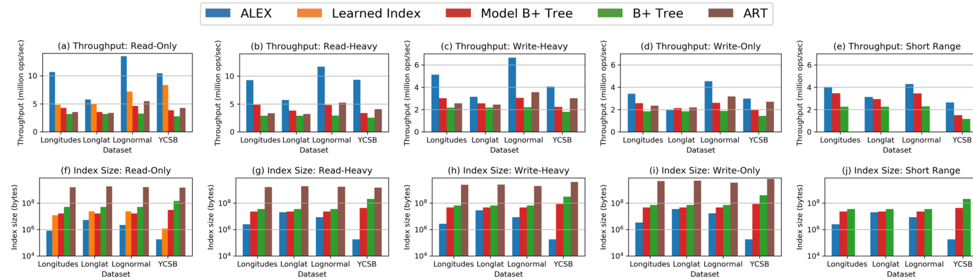


Figure 4.6: Performance of ALEX compared to RMI [4], B+-trees and ART [10]. Illustration extracted from [1]

ALEX authors compared the optimized versions of the index against RMI, B+-tree, model enhanced B+-tree and ART with a few different workloads and datasets. The evaluations showed that ALEX achieved up to 4.1x higher throughput and 800x smaller index size compared to the B+-tree. Compared to ART, 3x higher throughput and 8000x smaller index size. For write speed, the results were even better. 4.0x higher throughput and 200x smaller index size versus the B+-tree and 2.7x higher throughput and 3600x smaller index size versus the ART index. The bulk loading time of ALEX uses a significantly longer time than the B+-tree and uses 50% more time to build. The ART index uses an even longer time to build and uses around 50% more time than ALEX. Throughput results from the original paper are displayed in Figure 4.6.

Drawbacks

As with many other learned indexes, ALEX does not support secondary storage. The authors have mentioned in the paper that they intend to add this in the future. They also want to add concurrency control techniques tailored to the index and future work performance.

4.6 Bourbon

Bourbon is a different learned index that is an LSM-tree [24]. The paper is an initial study of how it is possible to use learned indexes for LSM-trees and how it possible to combine them with LSM design components. They study WiscKey [25], a state-of-art LSM system, and focus on how a new learned-based LSM-tree can be implemented. Since most of the LSM tree is immutable, read-only indexes work great. It means that LSM-tree can work fine with a read-only index and therefore does not require a learned index with efficient insert support.

Although, in theory, learned indexes seem to be a good fit for LSM, they encounter a few challenges. One of which is the support for a function that can support a variable amount of keys (as the original RMI only supported a fixed number of keys). Another challenge they encounter is that the model is built too early.

The authors conclude that the benefit of using learned indexes for LSM-tree is to reduce the overhead of indexing the data. It cannot reduce the data-access time. While using a lookup latency breakdown, it is not uncommon that the index time could count for around 15-50% [24]. Later, in Chapter 5, we will see how Google measured dramatically improvement of throughput using LSM with learned index. Google explains that it is due to decreased index size and side effects to cache and block decompression.

The authors conclude the initial study with that more studies and experiments are needed to thoroughly understand the utility of learning approaches for LSM-trees. However, Bourbon is a significant step forward for using an LSM with a learned index in a production environment.

Performance

Based on the numbers in the paper, the authors expect a 1.23x-1.78x performance increase over the state-of-art LSMs.

Drawbacks

It uses a simple learning model, which might be ineffective. Other models, such as the RMI and others presented later in this Chapter, could be more suitable and can provide better performance. RMI will, for example, provide better read performance but require more time to train.

4.7 RadixSpline

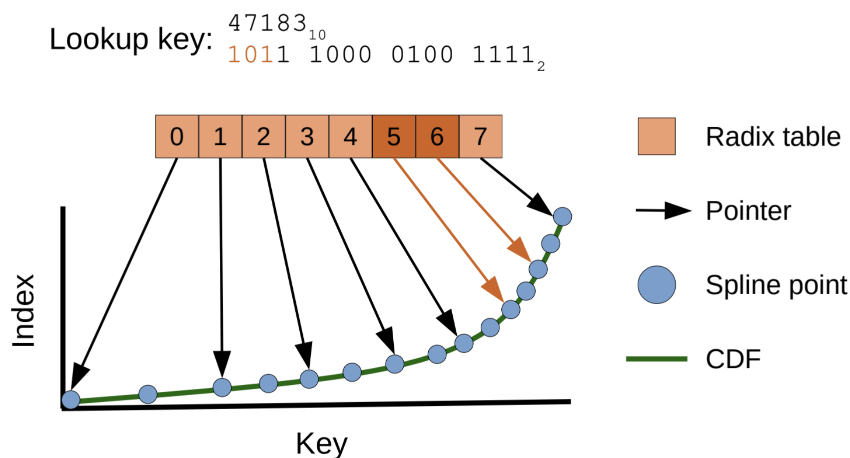


Figure 4.7: Example of the radix spline index. The most significant bits is highlighted, which is 101 (5). Then a search is performed in the spline point between the pointer and pointer + 1, which is highlighted in the illustration. Illustration is extracted from [26].

RadixSpline [26] takes a different approach to learned indexes and argues that some applications do not need individual updates. In some cases, building the index efficient is more important. It could be highly relevant, for example, in big data systems, where LSM-trees are often used. Each data file in an LSM-tree typically stores an index or a filter. These files are periodically merged with other files. An ideal time to re-build a learned index is when this merge process occurs. This process is usually done asynchronously, and the process using a traditional

approach is an expensive operation. These factors make this an ideal use case for learned indexes, and especially this optimized index. As mention in the earlier sections, both FITing-Tree and PGM provide a single-pass build solution that could also fit this case. The advantage with RadixSpline is that it is the first learned index with a single-pass constant amount of work per element, $O(1)$, in contrast to $O(log_i(n))$ where i is the number of levels for other indexes.

The index is created using two steps. The first is to fit a linear spline to a CDF with a certain error bound. A model is created, S, that predicts the location p_i given a key, k_i, and and an error, e, which gives the model $S(k_i) = p_i \pm e$. The model is created using an algorithm called GreedySplineCorridor [27] and produces a set of spline points, such that a linear interpolation between the points never creates an error larger than the error e. The points are displayed as an example in Figure 4.7. The second step in creating the radix spline index is to build the radix table. It is a table used to find the two spline points that are associated with a certain key and is based on the Node256, the largest node type in ART [10].

Performance

RadixSpline is an excellent example of how it is possible to make learned indexes that work great with big data systems. The insert time and build time makes it ideal for LMS-Trees. Already, a preliminary experiment with RadixSpline in RocksDB, substitute B-tree, shows a significant improvement in memory reduction by 45% and 20% improved read time. The write time goes up to around 4%, but still could be worth it for many use cases. The reduction of memory usage makes it possible in the future to use larger Bloom filters and increase cache, making it even more efficient.

The index showed fast building time and lookup times, compared to state-of-art indexes, such as ART. For comparison, see Figure 4.8

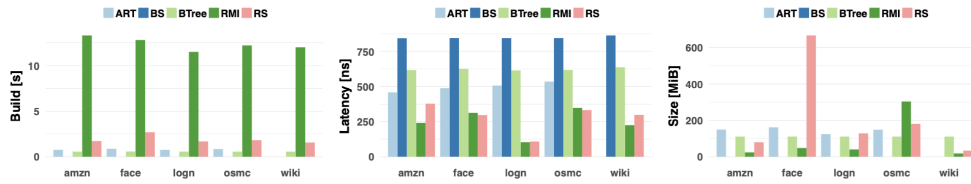


Figure 4.8: Performance results for lookup-optimized index configurations. RS is short for RadixSpline. Illustration extracted from [26]

Drawbacks

The index can be impacted by skew, which could make the index ineffective. Another drawback, as seen in Figure 4.8, the index size can spike to achieve the best lookup performance. The index also becomes less efficient with large indexes or with many outliers.

4.8 Tsunami

Tsunami is a recent paper on an in-memory learned index from 2021 that outperforms read-only multi-dimensional indexes. The index is the successor to a previously learned index named Flood. Tsunami employs advanced features, such as automatically optimizing against the workload and the dataset. The index can learn query skew and optimize the index against the skew. Previous work within learned indexes for multi-dimensional indexes, Flood, did not support skew in multiple parts of the dimensional space. It also suffered when correlated data was present. Tsunami elegantly solves the issues and achieves significantly better performance. Compared to traditional indexes, such as the K-d tree, Tsunami can achieve up to 11x faster query processing and 170x smaller index size, which pushes the boundaries for what is possible. The performance results from Tsunami represent a breakthrough for learned indexes. The index will have a significant impact on analytic engines in the future and also on filter performance in databases.

For the modern database solution, filtering data is essentials. Today, techniques such as multi-dimensional index, secondary indexes, and clustered index are used. The different techniques deliver inconsistent performance on different datasets and queries. Tsunami tries to solve the issue of inconsistent performance by automatically adapt itself to the workload and the dataset. The index also addresses

the shortcoming that an admin has to tune each approach, and no approach dominates the others.

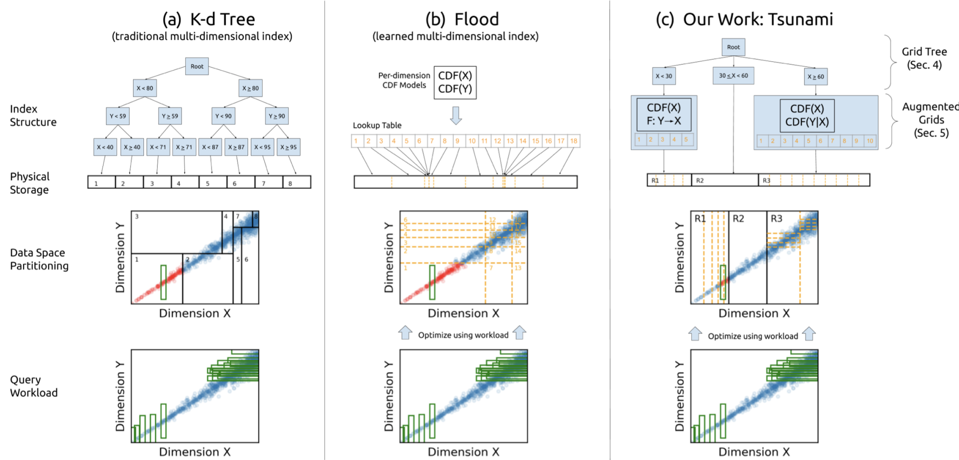


Figure 4.9: The illustration visualize the number of points that are required to scan using the different indexes. K-d tree is not optimized for the workload, and Flood is not optimized for the query skew and correlation. Tsunami has equally sized regions for the workload and adaptive to the query skew. The number of points to scan is then significantly less. Illustration extracted from [28]

Compared to the traditional index, K-d tree, Tsunami have some significant differences. In a K-d tree, the data is partitioned based on a threshold in each region. When a query is executed, each of the regions that are affected by the query is selected. Then all points in the regions affected are scanned. Since the K-d tree is created and maintained without knowing the query workload, the partitions are not optimized based where it is high query load. If the partitions are smaller, more fine-grained in areas where the query load is high, we can achieve much better performance. This created the general idea behind Flood, which optimizes against the average query and optimizes the average selectivity in each dimension. The flaw with this design is that the non-average queries will need to scan many partitions, which will degrade the performance for other queries. To solve this, Tsunami implements a grid tree and an augmented grid. The grid is a space-partitioning decision tree that divides the space into regions. We have an augmented grid in each region, which essentially is the Flood index structure updated to support correlations. The comparison between K-d tree, Flood, and Tsunami is visualized in Figure 4.9.

The Flood index, which essentially is the augmented grid in Tsunami, is a multi-dimensional index that uses RMI as the domain to divide each dimension into equally-sized partitions. Flood also features advanced features, such as a machine-learned-based cost model.

Performance

Tsunami delivers consistent performance and outperform Flood. It achieves up to 6x faster query processing. Compared to the fasted non-learned index, 11x faster. In addition, the index can quickly adapt to new workloads by running re-optimizations offline. Experiment shows that it was able to re-organize over 300M rows within 4 minutes. In terms of scaling, Tsunami can scale better with multiple dimensions than traditional indexes.

Drawbacks

Currently, Tsunami only supports read workloads. In theory, the index can support inserts but yet not implemented. Tsunami is also in memory but could be supported in the future. There is also a range of further improvements that we will likely see in the future, such as handling more complex correlations and performance improvements.

4.9 Comparison

All of the different approaches to achieving learned indexes have advantages and drawbacks. Some of them work especially great for LSM-trees, while others are optimized for read-only or as a drop-in replacement for B-trees. Learned indexes are at an early stage. The approaches covered here have been developed over just the last four years. Sadly, many of the different indexes have not been released as open-source or are missing publicly optimized versions online. This makes it especially hard to compare the indexes. However, in the next chapter, Chapter 5, we cover an in-depth review of the performance of learned indexes. In the rest of this chapter, a comparison of the different properties and approaches is described.

4.9.1 Properties

In Table 4.1, a comparison of the different properties related to the learned indexes are listed. In Table 4.2, the time complexity for the indexes is listed. For many learned indexes, the time complexity could be challenging to find. Mainly due to advanced cost models and prediction-based models are making it more challenging to model. For reference, a B+-tree is also listed. In contrast to B-trees, some learned indexes such as Alex grow as unbalanced trees, which could be very deep in theory.

Index	Year[1]	ML technique	Operations			Training approach	Open Source[2]
			Insert	Remove	Range scan		
RMI	2018 (2017)	Any	✗.	✗	✓	Top-down	✓[3]
FITing-Tree	2020 (2018)	Piecewise Linear Regression	✓	✗	✓	Bottom-down	✗
LISA	2020 (2019)	Piecewise Linear Regression	✓	✓	✓	-	✗
PMG	2020 (2019)	Any [4]	✓	✓	✓	Bottom-up	✓[5]
ALEX	2020 (2019)	Piecewise Linear Regression	✓	✓	✓	Top-down	✓
Bourbon	2020 (2020)	Piecewise Linear Regression	✓[6]	✗	✓	Top-down	✗
RadixSpline	2020 (2020)	Piecewise Linear Regression	✗	✗	✓	Bottom-up	✓
Tsunami	2021 (2020)	Any [7]	✗ [8]	✗	✓	Top-down	✓

[1] Year in parentheses is the year it first surfaced.
[2] Some of the open-source is non tuned variants, for example ALEX [1], see [16].
[3] Open-source code not released with original paper. First released in 2019 with SOSD benchmark [29] [16].
[4] Large part of LSM-tree is immutable. The index support inserts, however the index is built in batches.
[5] No implementation of the distribution-aware variant is released.
[6] The index support any prediction mechanism, but best result has been achieved with Piecewise Linear Regression
[7] Uses RMI as a model, which essentially could use any model.
[8] Can support inserts in the future.

Table 4.1: Properties of the learned indexes

Index	Complexity		
	Create	Insert	Lookup
RMI	$O(\ell n)^1$	-	?
FITing-Tree	$O(n)$	$O(\log_b p) + O(buff)^2$	$O(\log(s)$ ^{3}
LISA	?	?	?
PMG	$O(n)$	?	$O(\log(s))^3$
ALEX	$O(n\log(n))$	?	?
Bourbon	?	?	$O(\log(s))^3$
RadixSpline	?	-	?
Tsunami	?	-	?
B+-tree	$O(n\log(n))$	$O(\log(n))$	$O(\log(n))$

[1] ℓ is the number of stages.
[2] p is the number of pages.
[3] s is the number of segements for the given index.
? means unknown time complexity.

Table 4.2: Time complexity of the learned indexes based on details from papers.

4.10 Complex models versus linear models

As we can see in Table 4.1 the different machine learning techniques used are very similar. Most of the recently learned indexes are based on piecewise linear regression. To achieve high performance and low memory footprint, the research has focused on simple linear models so far. As seen with the different approaches, except the original RMI model, some type of linear model is deployed. Training a neural network comes at a high cost and might have to be retrained. The ALEX paper included an exciting footnote to argue why they used linear models instead of neural networks in their RMI-based index. In private communications with the authors of the RMI paper, they concluded that added complexity of a neural network for the root model was usually not worth it. Early experiments with PGM also suggest that the benefits from using shallow neural networks are small [30]. A reduction of models was achieved, but the overall space usage was not reduced due to the cost of the neural network.

Research on different models, such as more complex models, deserves further research. ALEX suggests that the index performs poorly when the distribution is

hard to model using linear regression and that a possible further research direction could be to focus on different types of models, such as polynomial regression [31]. We know that there are pretty specific patterns of operations to an index for certain types of usage. For example, inserts could follow sinus wave looking graph depending on the time of the day. By using more complex models, the pattern could be recognized, and a more straightforward index structure could be used.

4.11 Bottom-down versus top-down

In general, learned indexes are either trained top-down or bottom-up. Indexes such as PGM [2], and RS [26], fits the bottom layer to a certain accuracy, then build the subsequent layers on top. Models that are built bottom-up might have the disadvantage that search for each level is required, and we can get more cache misses and branch misses [22]. The authors behind the first paper on learned indexes, RMI, have experimented with different bottom-up approaches usually found them to be slower than top-down [22] [4].

4.12 Secondary indexes

Learned indexes can be used as secondary indexes as well. We then use the same techniques as with B-trees by using indirection. Learned indexes will then have the advantage of a smaller index size and faster query time. Another option could be to use Hermit [32], an ML-enhanced mechanism for succinct secondary indexing.

Chapter 5

Performance of learned indexes

The performance promised by learned indexes does not always reflect an accurate picture. Some papers have been criticized [29][22] to use datasets that follow clear patterns or test the indexes in very particular use cases. In this chapter, we will go into detail on the performance of learned indexes. Where do they perform well, and where do they suffer? How do learned indexes perform compared to traditional indexes?

5.1 Practical example - Google Bigtable

Much criticism against learned indexes is centered around the lack of metrics on how learned indexes perform in real-world situations and how learned indexes can deal with dynamic workloads [33]. Recently, in 2020, Google tested the performance of a learned index in a real-world situation in Bigtable. Bigtable is the proprietary data storage system that Google uses in services and applications such as Google Maps, YouTube, and Gmail. It has been argued that learned index structures only improve requests by nanoseconds and that the time used to index data is a small part of the over all time it takes to process a request.

In the study, they modified the originally learned index, the RMI. Since Bigtable is block-based, they have to change the index from assuming that all data is stored in the same continuous array in memory to support block-based files. They modify the index to define which records are stored in which block. They also prevent searching in multiple blocks after a prediction. When introducing learned indexes, the write performance will not be improved. Since the data is moved

into the index and trained in the background during SSTable creation, the write performance will not be affected. However, read-performance will be improved.

5.1.1 Evaluation

Using 15 clients and five servers running each 4GB RAM and 256 million rows of data, the index reduced the latency by 36% mean latency for point lookups. Within the 99% percentile, the reduction in latency was by 38%. For range scans, the latency was unaffected in the 99% percentile, while we see a reduction of 22% in the mean latency. They also measured the throughput increase, and we can see an increase of an incredible 55% for point lookups and 28% for range scans. In the 99th percentile, we see an increase of 54% and 56%, respectively. The comparison between the learned index and the existing solution in Bigtable, a two-level index, is displayed in Figure 5.1.

They conclude that the reason for the decrease in latency is the elimination of disk access to fetch index blocks. The increased throughput is due to the reduced amount of blocks to access, and that big table has to decompress every block as it is compressing the data heavily. It is common in other disk-based systems as well.

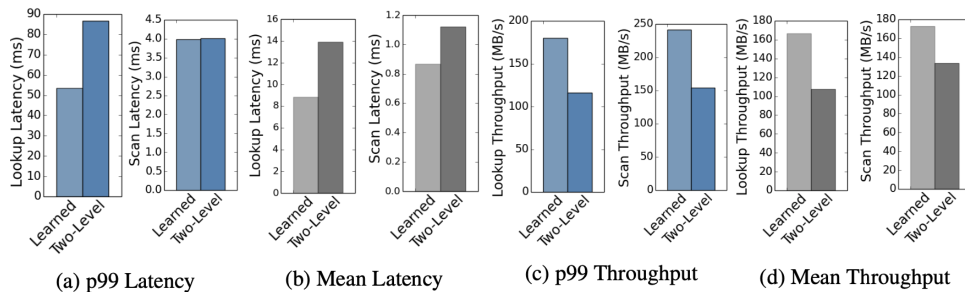


Figure 5.1: Performance comparison between learned index (RMI) and the regular two-level index in Bigtable. Illustration extracted from [33]

5.2 Practical example - Learned indexes in DNA sequence analysis

Recently, a paper on using learned index in DNA sequence search was published [34] from DSAIL MIT. It is a preliminary proof of concept, where a new learned index, named LISA (Learned Index for Sequence Analysis), is introduced. It should

not be confused with the spatial index with the same name, LISA. The LISA for DNA covers the exact search problem for DNA sequence alignment in genome analysis, where the exact search is presented as the bottleneck for the performance. Initial experiments show that it can improve today's searching performance of 4x of exact search. By the authors' knowledge, this is the first ML-enhanced algorithm to speed up this particular search problem with the same semantic guarantees as traditional algorithms. Using techniques from learned indexes, they show how it could be applied to DNA sequence search and present the first steps. The researchers is already working with Broad Institute of MIT and Harvard to implement the search algorithm in widely used applications for the genomics community. In the future, they hope this research could be used as a building block for future learn-based DNA sequence searches.

LISA learns the distribution of suffixes to speed up the search, in a similar fashion to learned indexes learns the distribution of data. Today's state-of-art algorithm to perform the particular search builds an FM-index (a method created by the same author as the PGM index). LISA enhances this method to enable the application of learning-based search. One of the key elements it does is to use RMI when processing chunks. The RMI model used here is to build bottom-up and therefore have the ability to bound the error. To fully understand the algorithm and its details, see the original paper [34].

In a follow-up paper on LISA, not yet released, early results indicate a 13.3x speedup for the SMEM (super-maximal exact match) search problem in DNA sequence search [35]. Again, using learned indexes. In this research paper, LISA performs slightly slower on exact search, by up to 2.2x speedup. Details from the new SMEM speedup are yet to be disclosed in an upcoming paper.

5.3 Difficulties when performance testing indexes

5.3.1 Tradeoffs

The performance of indexes is hard to quantify. One single metric does explain the complete picture of how the index performs. Indexes could be optimized for a range of different use cases, which often affect the performance of other features related to the index. For example, if the only goal was to optimize lookup time,

we could have used a lookup table and avoided cache misses. The downside is that we would have used a significant amount of memory to store the lookup table. A better measure to look at the lookup time is, therefore, also to consider the lookup time given the size of the index. A different factor that makes it even harder to measure tradeoffs is the building time which also puts new constraints on the index. With more time to build the index, it is possible to use this time to build it more efficiently. Making a read-only index also improves the performance of an index that supports inserts as well. It makes it more challenging to compare read-only learned indexes to traditional indexes.

5.3.2 Optimizations

When we look at performance results in papers, there could be multiple reasons why we often see such good results. Indexes could also be optimized against certain types of data. For example, RadixBinarySearch (RBS) could work great in some situations, but if we, for example, add certain types of skew, the method could perform very poorly. E.g., specific numbers of the bits could be nearly useless. Avoiding this type of skewed data in the datasets could significantly improve the results. Another example of how it is possible to improve the numbers, which is pretty standard, is to use a warm cache rather than cold. It will decrease the number of cache misses and improve the results significantly. Accessing a cached value could take around 10ns versus around 100ns for an uncached value [22].

5.3.3 Hardware specific optimizations

Another more complex topic is advanced optimizations against hardware. Indexes have significant performance differences when memory fences are enabled versus disabled, and according to [22], the difference when enabled could be a 50% slowdown. In an application with extensive computation between index lookup, enabling memory fences would not be beneficial. The issue with such hardware optimizations is that we get new index aspects that are often not adequately covered in the original papers. At the same time, the impact of various sets of features could have a significant impact when finding the correct index for a particular application.

Some new indexes are also created for new types of hardware, such as flash memory and NVM memory, which are essential properties but not always possible

to cover in a single benchmark.

A different property that is getting more important with indexes is the ability to scale with more threads. As mention in Chapter 3, Moore's law is dead. Much of the performance improvements expected to see in the future on CPUs are with more threads (cores). The increasing amount of data indexes have to handle puts pressure on developing indexes that efficiently handle multithreading. Indexes have to divide the workload on multiple threads or support used from multiple threads. In general, one important property to efficiently scale with multiple threads is few cache misses. An issue that occurs with many cache misses is that the latency will be bound by waiting for access to RAM [22]. A new future direction with learned indexes is that many of the indexes can run GPUs and could be converted to a sparse matrix multiplication which can be used to GPU/TCP accelerate the index.

5.3.4 Purpose when performance comparing indexes

Some learned indexes, like the RMI [4], is a read-only index and optimized for this case. At the same time, other indexes focus on different properties, such as fast insert, small index size, and disk-based index. Therefore, comparing indexes one by one does not always display the complete picture. Instead, comparing indexes with a specific use case gives more information. The currently available benchmarks for learned indexes are done in the context of read-only and are comparing optimized versions of the indexes against each other and other state-of-art indexes for lookup. It is important to stress that while some of the learned indexes are read-only, many of the traditional indexes are compared to support updates.

5.4 Performance comparison between indexes

As an effort to answer the question of whether or not learned indexes outperform traditional indexes on real-world data, a group of authors, including some of the authors behind various learned indexes covered here, created a special benchmark framework to measure the performance of learned indexes against traditional indexes, such as ART [10], B+-tree, and FAST [36].

5.4.1 Search On Sorted Data Benchmark (SOSD)

In their first paper, they release the benchmark framework Search On Sorted Data Benchmark (SOSD) [16]. The framework is specially designed to benchmark a given learned index against several different datasets with a variety of different CDFs. It is then possible to see how the index performs against traditional indexes as well. Although the index is specially designed for learned indexes, the benchmark works excellent for regular indexes as well. It is carefully designed with minimal overhead written in C++, which only adds an overhead of eight instructions and one cache miss per lookup.

In addition to the benchmark, they also released the first performant publicly available implementation of the RMI. Most likely with the help of the author of RMI itself, as he is listed as a co-author. The benchmark is available as open-source code [29]. It also has different modes, making it possible to measure counters such as branch predictions, instructions, and cache misses.

By default, eight different datasets are included in the benchmark. Some of them are real-world datasets, while other is just a sampled distribution. Each dataset is provided in a 32-bit and a 64-bit version with about 200 million records with very few duplications. The CDFs of datasets is displayed in Figure 3.2, in Chapter 2.

The lookup time results they found using the SOSD benchmark are displayed in Figure 5.2. As expected, with traditional data structures such as the B-tree, FAST and BinarySearch (BS), the performance is little affected by the data distribution. The interpolation search (IS) varies significantly due to data skew. When we look at the performance of the learned indexes, the RMI and Radix Spline (RS), we can see a more significant variation in performance between the different datasets. In general, we can see the lowest lookup latencies for the learned indexes. Another interesting takeaway is the size overhead, where we can see that learned indexes deliver as promised in their respective papers to have a very low memory footprint. It should be noted that build times from both of the learned indexes are significantly larger than for the traditional index types.

	ART	B-tree	BS	FAST	IS	RBS	RMI	RS	TIP
amzn32	n/a	529	773	**244**	4604	325	264	275	731
face32	**187**	524	771	229	1285	312	274	386	964
logn32	n/a	522	765	294	n/a	471	**97.0**	105	744
norm32	191	522	771	229	10257	355	**71.7**	**70.9**	884
uden32	102	521	771	228	**39.8**	333	54.2	64.2	176
uspr32	n/a	524	771	230	469	301	**153**	200	400
size overhead	47%	16%	0%	123%	0%	< 1%	3%	< 1%	0%
amzn64	n/a	601	804	n/a	4736	387	266	**288**	759
face64	391	592	784	n/a	1893	**337**	**334**	461	1232
logn64	309	597	784	n/a	n/a	753	179	**120**	454
norm64	266	592	785	n/a	10510	405	**71.5**	**70.5**	862
osmc64	n/a	599	785	n/a	95076	492	**402**	437	7186
uden64	112	592	784	n/a	**43.4**	344	54.3	53.9	193
uspr64	287	591	785	n/a	449	313	**169**	214	428
wiki64	n/a	608	802	n/a	7846	364	**222**	**218**	1019
size overhead	25%	16%	0%	n/a	0%	< 1%	3%	< 1%	0%

Figure 5.2: Lookup results in nanoseconds. Table extracted from [29]

The benchmark also measures different properties such as branch mispredictions, number of executed instructions, and cache misses. The paper authors did an experiment where they measured the different indexes on the amzn32 dataset, where FAST performed slightly faster than the learned indexes. As seen in Figure 5.3, even with a significantly more large number of cache misses, instructions executed, and branch mispredictions, the difference was only up to 31 ns between FAST and the learned indexes. They conclude that analyzing cache misses alone is not sufficient to understand search time.

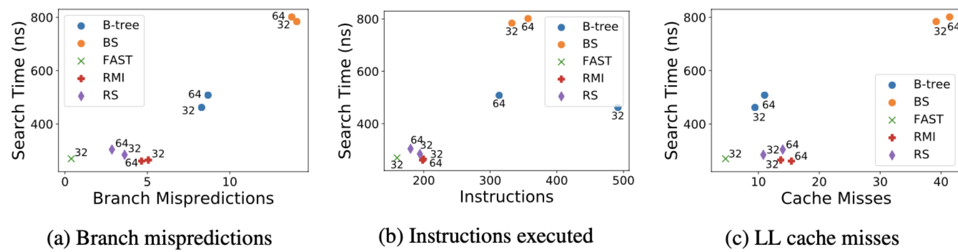


Figure 5.3: Performance counters. Illustration extracted from [29]

Both the RMI model and RadixSpline model is implemented using simple model (linear models as building blocks), yet they outperform traditional indexes, such as the B-Tree, which makes the authors of the SOSD paper conclude that

simple models are sufficient for efficiently learned indexes. As always, to achieve the best performance given a particular case, there is rarely one solution that fits all. In particular, interpolation search outperforms all of the other indexes, including learned indexes, for dense uniform integers. Learned Indexes, in this case, still perform around ten times faster than the B-tree. Due to the lack of updates, and the requirement for manual tuning on RMI and RadixSpline, the authors of the benchmarking paper suggest using ART and FAST for 32-bit keys and ART or Radix Binary Search for 64-bit keys. Later, after the paper was released, indexes including Alex [1], FITing-Tree [18] and PGM [2] has been released, which addresses the updating issue and making it easier to tune the index. Alex is one example of indexes that auto-tunes itself using a cost model.

5.4.2 Extension of SOSD

One year later, a new complete study on the performance of learned indexes was released by the same team behind the first paper. This time, they discard the use of synthetic datasets, as they argue that synthetic datasets either could be incredibly easy to learn as they could be drawn from a know distribution or very hard to learn. A completely random distribution could make it hard to learn something from and does often not reflect real-world datasets. To address the tradeoff issue between index size versus performance, they performed a Pareto analysis to find the Pareto optimal index. A Pareto optimal index is an index for which no alternative has both a smaller size and improved performance [22].

The paper covers a benchmark on four different datasets.

- **amzn**: Each key represents the popularity of a particular book
- **face**: Randomly sampled Facebook user Ids
- **osm**: Each key represents an embedded location from Open Street Map
- **wiki**: Each key represents the time an adit in Wikipedia was comitted

They benchmark the PGM- [2], RS- [26] and RMI- [4] index, and compare them to B-tree, IBtree [37], FAST [36], ART [10] and FST [38]. In addiation, they also test the performance against hash indexes, which is not covered here. Learned indexes, such as Alex [1], FITing-Tree [18] is not included, due to tuned implementation could not be made publicly available.

In extension to the Pareto analysis, they also analyze performance counters and other descriptive statics, CPU interactions, multithreading analyses, and build times. Since RS and RMI are read-only indexes, update and insert performance are not measured.

The results from the Pareto analysis is displayed in Figure 5.4. In the analysis, they are using 10 different configuration, where the size of the index and the lookup time varies. An interesting pattern we can see, is that the tree based indexes is non-monotonic. They become less effective after a certain size and starts to decrease their performance. In the paper, they argue that this is because performing binary search is faster than traversing a tree, given a certain point. The index size become too large.

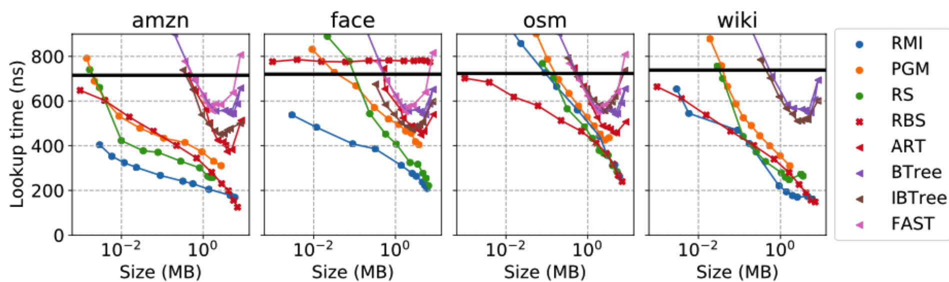


Figure 5.4: Pareto analysis of the indexes. Lookup time compared to space usage for learned indexes and traditional indexes. Illustration extracted from [22]

5.4.3 Why learned index structure perform well

When analyzing why learned indexes achieve such good performance, the picture is rather complex. Single metrics, such as instruction count, branch misses, cache misses, model size, or the accuracy, could not fully describe the picture alone but as a combination. They statistically test the explanatory factors and find that cache misses, branch misses and instruction count explained 95% of the variation [22]. Further, surprisingly, they get a negative coefficient for branch misses. The present two different explanations. The first is that structures could be over-optimized to avoid branching. The second is that indexes that frequently experience branch misses benefit from speculative loads on modern hardware.

5.4.4 Multi-threading

They measure the number of cache misses along with increasing of threads. FAST is the index that benefits the greatest of multi-treading, with a 32x speedup, with 40 threads. FAST takes advantage of the effective overlap of computation with memory reads. They find that cache misses correlate with the speedup factor but is not always a direct factor. The learned indexes, along with RBS, achieved the best results. The performance results is displayed in Figure 5.5.

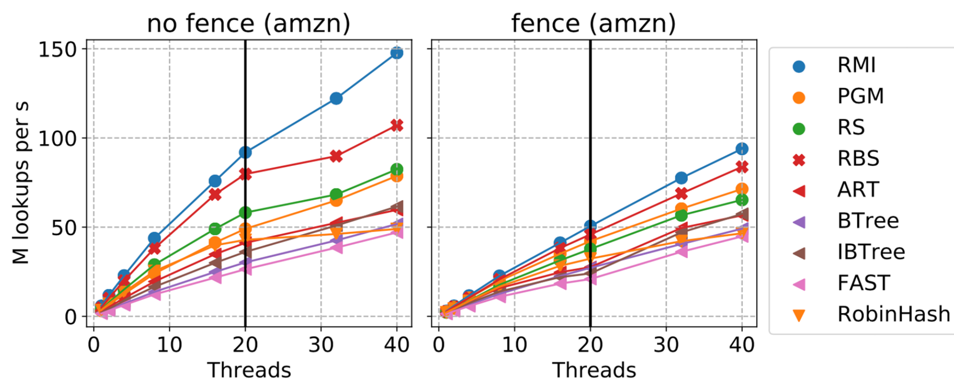


Figure 5.5: Multi-threading performance on the amzn dataset. With and without memory fence. Illustration extracted from [22].

5.4.5 Build times

Figure 5.6 shows the build times for the different indexes. In the graph, tuning of the indexes is not included, but it could take several of minutes. Tuning indexes is executed for both traditional and learned index, but take significantly longer with learned indexes compared to B+-tree due to slower build times. Surprisingly, the traditional indexes, FAST and the hash indexes RobinHash and Wormhole, had the longest build time. Learned indexes is just right after, with also significant build times. In theory, learned indexes could speedup build times, by enabling multi-threading which today no learned indexes support.

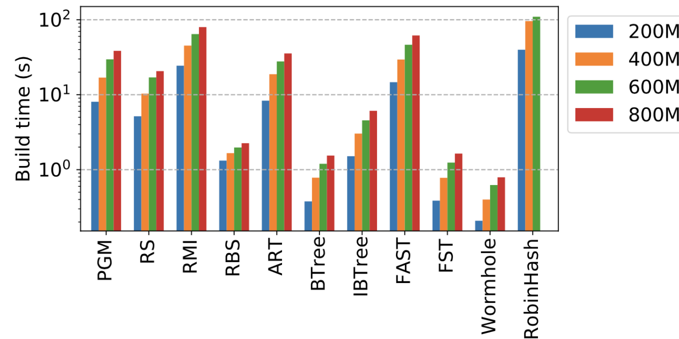


Figure 5.6: Build times of the different indexes, using the amazon dataset with different amount of keys. Log scale on the y-axis. Illustration extracted from [22].

5.5 Range search

In theory, most learned indexes will have similar performance for range search as B-trees, as the data is sorted. All of the described learned indexes here support range search. Learned indexes for hash indexes has not performed such well and therefore been little focus on developing

5.6 Summary

We have seen that learned indexes dominated the Pareto analysis by providing some of the best results. In some cases, other indexes performed better, but in general, learned indexes shows impressive results. The reason why learned indexes perform such well, is not due to a single factor, but can be explained using multiple metrics, where cache misses was one of the most significant factors. Although, learned indexes have the best performance / size ratio, the build times is significant larger than traditional indexes. RMI provides the best ratio, but also the largest build time. We saw that PGM and RS can achieve comparable performance, but at a significant faster build time. No benchmark of insert performance exist yet, but is expected to come when more indexes that support inserts release their optimized versions publicly.

Chapter 6

Limitations and future work of learned indexes

6.1 Criticisms and skepticism of learned indexes

Learned Indexes take a radically different approach to achieve efficient data access and claim substantial memory and performance improvements. For decades, the research has been centered around making an index that assumes very little about the data. Understandably, along with the excitement for learned indexes, there has also been significant skepticism [16]. When the first paper on a learned index was released, no open-source code existed, no theoretical proof, no benchmark tool, and no support for inserts. Today, there is multiple different implementations available as open-source code [2][1], theoretical proofs has been created [17], a benchmark tool has been developed [16] and there exist multiple learned indexes with support for inserts [18][2][1]. Another criticism against learned indexes is about the missing support for disk-based systems, where indexes today matter the most [29]. Today, FITing-Tree [18] addresses this critic by providing one of the first learned indexes that are disk-oriented.

There is still some skepticism that has yet to be addressed, mainly how learned indexes will perform in real-world scenarios. The practical example of where learned indexes have been tested in Google's Bigtable may be the closest we get at this point. In addition, LISA for DNA sequence analysis also gives us a pointer. There has also been raised a more general question regarding indexes if index size and speed improvements matter since, in practice, indexes only stand for a

fraction of the entire query execution. As seen in Chapter 5, index size matters. By dramatically reduce the size of the index by orders of magnitude, it is possible to move indexes to the main memory. A smaller index size also means fewer block fetches. As described by [3], distributed data systems, such as Google's Bigtable, faces these challenges with a limited amount of memory per index on each instance and large index size, which together put enormous pressure on the cache.

6.1.1 Learned index can not outperform tuned traditional data structures

In the article, The Case for B-Tree Index Structure [39], critics are raised against learned indexes in general, but specifically, the first version of RMI. The question of whether we need neural networks for learned models is asked. Since optimized versions of RMI were not available at the time, it was hard to criticize RMI on the performance results, but an approach is described. The authors of RMI have later claimed that the approach described had a negative impact on the B+-tree, and therefore still proving the point to RMI. Another question that is raised is that machine learning will have great difficulties with updates. Indexes such as ALEX and PGM proved that it is possible to implement efficient insert in learned indexes. In 2019, the same author that wrote the critics also wrote an article on why we should use learning when we can fit instead [40]. A different approach to learned indexes is described, where he builds a CDF, then fits a linear spline and fits a polynomial function to the spline nodes. He then compares the index to a B+-tree and a learned index and measures a lower error than the B+-tree and the learned index. All indexes use the same space consumption. The article has since not raised any fuzz, but it might sparked the idea behind RS, of which he is the co-author.

6.1.2 Learned indexes comes from implicit assumptions

Are learned indexes just based on implicit assumptions? A recent paper from Brown University in 2020 raises strong criticism against learned indexes [41]. To make their case, they introduce a new index that takes the same assumptions they mean many of the learned indexes also take; read-only, data sortedness, and range of the data. With the new index, Hist-Tree, they can beat RMI, PGM, and RadixSpline by 1.8-2.7x. They argue that the advantage learned indexes have by taking this assumption makes comparing traditional indexes and learned indexes

unfair. The paper delivers a strong case against learned indexes and the assumptions made, but further work is required to verify the results. When writing this thesis, the Hist-Tree paper has not yet been cited by others papers, and we do not know yet what the impact of the paper could be. Important takeaways from the paper are that we might compare apples to oranges when comparing learned indexes to traditional indexes. The author still supports learned indexes but doubts that the performance, currently, is better than traditional indexes. He also argued that for high-dimensional data, learned indexes are a better approach than traditional indexes [42].

6.2 Limitations and future work

Learned indexes are still at a research phase and currently yet to be seen in production. Even though some of the indexes are starting to be more advanced with more features, the indexes still need to prove that they could work in a real-world environment. With some even ready as a drop-in replacement for traditional indexes, such as the PGM, the improvement Google has seen on Google Bigtable, and the amazing performance with LISA for DNA sequence analysis, it is not unlikely that we will see learned indexes in production within a couple of years. However, before that, some issues need to be mined out.

One of the most significant issues with learned indexes was, for a time, the lack of efficient updates. The PGM index makes a leap forward here by delivering the first learned index with provable efficient time and space bounds for updates and range queries. For general inserts, the time complexity is on pair with B+-tree. It is expected to see further improvements. Building time for learned indexes using bulk loading, is still significantly higher compared to B+-tree [22]. Progress here is also expected to be seen in the future. The building time should also be related to the use case. For example, a read-only index might benefit from a longer building time, which might compress the index size even more or improve the lookup time. Models which require better read speed will likely still have significantly higher building time, while indexes made for faster insert might have improved building time over today's performance.

Measuring the performance of learned indexes is another topic, which requires

more research. The indexes are currently missing theoretical proofs behind its techniques, such as more precise worst-case amortization time complexity analysis. The paper "Why are learned indexes so effective?" [17] provided the first steps towards proving that learned indexes are probably better. However, there are still some open questions. Theoretical proofs are still missing and also the more practical side. Benchmarking learned indexes against traditional indexes, we are missing publicly available learned indexes to make a complete benchmark tool, especially for insert performance. There are also missing benchmark results for write performance. It also needs to be tested against more datasets.

Currently, we do not know how secure learned indexes are. It has been suggested by MITs Data Systems and AI Lab (DSAIL) that it may be vulnerable against adversarial attacks [43]. Research suggests that, in general, adversarial attacks on deep learning models should be a severe concern [44]. For RMI-based models that employ neural networks, this may be a threat that should be investigated further. DSAIL already proposes to address these security concerns.

6.2.1 Author thoughts

Learned indexes can have the capability to be the one solution that fits "all" truly. With inventions such as dynamic query handling seen in PGM [2], dynamic workload handling by Tsunami, a breathtaking reduction in memory usage with a difference of magnitudes of order and hyper optimizations against a particular dataset with RMI [4] to improve performance. To truly take advantage of several of the indexes, much manual tuning is required, which makes several of the learned indexes impractical. A future direction we might see with learned indexes is the ability to dynamically adapt to different use cases with minimal manual tuning, which we have already seen the start of with PGM and Tsunami, which support dynamic workload skew. Learned indexes can have the ability to learn its use case, optimize the index based on how it is being used. It could give the indexes a significant advantage over traditional indexes.

A use case where learned indexes could work especially great is big data systems such as with LSM-trees. As seen with the experiment Google did with Bigtable, in Chapter 5, learned indexes contributed to a significant improvement in both throughput and latency. It is a field where learned indexes could potentially

make a massive leap in performance and memory usage that perfectly fits the current downside of learned indexes. However, a small effort has been seen in this area, and there is strong reason to believe that future research here could be fruitful. In combination with the recent research within learned bloom filters [4][45][45] and other learned structures, makes it likely that new methods will replace the current solutions for big data with learned components.

There is also reason to believe that making indexes multi-threaded will get more attention. Moores law is dead, but we still expect to see huge performance improvements for GPUs and CPUs. They are rapidly increasing the number of cores. The importance of building indexes that efficiently scales on multiple cores is getting more critical. Some of the learned indexes can have a considerable advantage here by removing search processes required between the stages when searching for an item. It makes it possible to represent the learned index in the GPU/TPU as a sparse matrix-multiplication [4], and thereby GPU/TPU accelerates the indexes for search.

Another hardware-related direction we might see with learned indexes is native support for NVM memory, which further could accelerate the performance of learned indexes. When learned indexes getting closer to productions, recovery code would be essential to add. Using NVM memory, it would be possible to avoid adding much of the recovery overhead. Another advantage is the improved speed over storage devices such as spinning disks or SSDs. An exciting feature to learned indexes is to combine features from the BzTree [46], which is a latch-free NVM index with a learned index. Bw-Tree [47] is also a latch-free index that effectively exploits caches of modern CPU chips, where features of it might benefit learned indexes.

Although there has been some exploration between different types of machine learning models, learned indexes are just in their infancy and should be expected more exploration here with different types of models. Currently, primarily vanilla (shallow) neural networks and linear models have been successful. There are some indications that neural networks increase the index size in greater size than the benefits over linear models [2]. More research is expected on, for example, polynomial models [48]. Time series prediction, using machine learning

methods such as recursive neural networks [49], and attention-based methods [48] could further extend certain learned indexes to predict the future better and build an index optimized to handle the predicted workload.

6.2.2 Future ideas for research

Learned indexes can be a complex topic. Creating an entirely new learned index is a massive task. Fortunately, there exist multiple research ideas that could be explored, which are a bit smaller. From a personal perspective, implementing an index like RMI on a GPU/TPC would be an exciting experiment. At the time of writing this thesis, the number of publicly available versions of *optimized* learned indexes is limited. In the near future, more of the learned indexes might become publicly available. An in-depth benchmark on write performance and performance on real-world datasets could also be interesting. A different topic that is highly relevant is to explore learned indexes in LSM, where there is some research already [3][50]. A third good option is working on outliers. It is a well know problem with some learned indexes, such as ALEX [31]. A research direction here is to create some logic for handling extreme outliers. The last recommendation is to develop, implement, and test new models, such as polynomial models to model CDFs, attention-based LSTM, and shallow neural networks.

6.3 New direction in the field of databases

The team behind RMI [4], DSAIL MIT, is on a mission to transform database management systems and believes that learned indexes are just a first step. The overall goal is to replace components in database management systems with learned components. Just a year after the final version of the first paper on learned indexes, they released the vision paper SageDB [15], A Self-Assembling Database System (2019). SageDB represents a new vision within data processing systems by implementing machine learning at the core. It proposes a radical change in the way we make database systems. The idea is a database that leverages machine learning to model data distribution, workload, and hardware. I combination, this enables SageDB to learn the optimized access of data and execution of query plans. By taking advantage of patterns in data, databases can be able to learn and automatically optimize by creating structures that can take advantage of the patterns.

DSAIL MIT claims that we are just in the beginning of truly understanding the potential of learned models in context to how they can improve traditional algorithms and data structures in DBMSs [43]. Currently, the SageDB is under development at DSAIL MIT, which will enable research for model-driven data processing engines. Already, there is multiple publications on the way to achieve SageDB, including, the papers which is covered here RMI [4], ALEX [1], SOSD [16] and [22]. A collection of papers related to learned query optimizes and executors are also published, and also some others are related to learned indexes and data access, including multi-dimensional learned indexes, such as Tsnumai [28] which is also covered here.

Chapter 7

Conclusion

Learned Indexes follow the trend of using machine learning to optimize databases. We have seen that knowledge of the data stored in an index improves the performance of the index. With more advanced models and cost models in the future, it would be possible to build learned indexes that optimize the learned indexes not only for the data but also for how it is used. More advanced indexes could automatically optimize the index for the given hardware, query load, index size, and which operations to optimize. We have already seen example such as PGM and Tsunami that optimize the index based on the workload. For some indexes, updates are more critical than lookup performance. For others, only a subset of the data is getting rapdily updated or read. Learned indexes make a giant leap forward over traditional indexes to truly make a dynamical index that changes based on the use case without assuming general distributions of the stored data and how the data is used. Learned index has the potential to adapt itself to the use case, not the user that has to adapt the index to fit the use case. Therefore, learned indexes are on their way to make a good candidate for one solution that fits all.

In Chapter 5 we saw that learned indexes outperformed traditional indexes for reading performance. Learned indexes can deliver faster lookup performance while also maintaining a smaller index size. Learned indexes also provided some of the best results for multi-threading, but we can expect improvement as some of the indexes could be represented as sparse matrix multiplication on GPU/TPUs [4]. In the future, it is expected to see more indexes optimized for multi-threading as well. Previously, critics have been raised against learned indexes which claims the memory size does not matter [3]. We now see the importance of reduced index

size. Reduced index size makes it possible to move indexes to in-memory, make space for larger bloom filters, or improve caching. It also has side effects like reduced seeks, more straightforward block prefetching, and fewer block reads, which again also could optimize cache misses [3]. In real-world situations, we saw a 38% improvement for latency in point lookups and a 54% increment for point lookup throughput in Google's Bigtable.

Learned indexes recently entered the space of DNA sequence analysis and have already shown stunning results, which can have a significant impact. The new methods for SMEM search and exact search significantly speed up the process and dramatically cut costs. Already, the Broad Institute of MIT and Harvard, in collaboration with the authors of LISA working on integrating it into applications widely used.

Most of the original critics raised against learned indexes have been answered with new indexes and papers. Learned indexes now have the first theoretical proofs that it is faster than traditional indexes, open-source code implementations are available, inserts are supported, and a proper benchmark of learned indexes exists, but we are still left with one question. Are learned indexes just based on implicit assumptions? We might compare apples to oranges in some cases when comparing indexes with different properties.

With indexes such as PGM that is the first provable index that works as a complete drop-in replacement for traditional indexes, LISA that outperforms today DNA sequence-analysis search algorithms, RMI that showed big improvements for Google Bigtable and Tsunami that outperform today's multi-dimensional indexes, learned indexes proves its position and it is just a question of time before we see learned indexes in production. Over just a brief period, learned indexes have taken a huge leap forward and are rapidly developing. In the future, the database vision SageDB might be the future home for learned indexes. However, there is a far way. The indexes have provided very promising results and can fit well with a combination of other learned database components in future database solutions. Learned indexes provide today some of the fastest access times and seems to be the answer to tomorrow's requirements for efficient data access.

Bibliography

[1] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, 'Alex: An updatable adaptive learned index,' in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[2] P. Ferragina and G. Vinciguerra, 'The pgm-index: A fully-dynamic compressed learned index with provable worst-case bounds,' *Proceedings of the VLDB Endowment*, vol. 13, no. 10, pp. 1162–1175, 2020.

[3] H. Abu-Libdeh, D. Altınbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, A. Ly, C. Olston *et al.*, 'Learned indexes for a google-scale disk-based database,' *arXiv preprint arXiv:2012.12501*, 2020.

[4] T. Kraska, A. Beutel, E. H. Chi, J. Dean and N. Polyzotis, 'The case for learned index structures,' in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 489–504.

[5] R. Bayer and E. McCreight, 'Organization and maintenance of large ordered indexes,' in *Software pioneers*, Springer, 2002, pp. 245–262.

[6] Sewaqu. (2010). 'Linear regression,' [Online]. Available: `https://en.wikipedia.org/wiki/Linear_regression#/media/File:Linear_regression.svg`.

[7] Krishnavedala. (2011). 'A function (blue) and a piecewise linear approximation to it (red),' [Online]. Available: `https://en.wikipedia.org/wiki/Piecewise_linear_function#/media/File:Finite_element_method_1D_illustration1.svg` (visited on 28/05/2021).

[8] (). 'Reading (e)cdf graphs,' [Online]. Available: `http://docs.battlemesh.org/v8/ecdf.html` (visited on 08/06/2021).

[9] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, 'The log-structured merge-tree (lsm-tree),' *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[10] V. Leis, A. Kemper and T. Neumann, 'The adaptive radix tree: Artful indexing for main-memory databases,' in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, IEEE, 2013, pp. 38–49.

[11] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan and C. Shahabi, 'Big data and its technical challenges,' *Communications of the ACM*, vol. 57, no. 7, pp. 86–94, 2014.

[12] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma and R. Shen, 'Reducing the storage overhead of main-memory oltp databases with hybrid indexes,' in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1567–1581.

[13] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil and N. Tatbul, 'Neo: A learned query optimizer,' *arXiv preprint arXiv:1904.03711*, 2019.

[14] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh and T. Kraska, 'Bao: Learning to steer query optimizers,' *arXiv preprint arXiv:2004.03814*, 2020.

[15] T. Kraska, M. Alizadeh, A. Beutel, H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao and V. Nathan, 'Sagedb: A learned database system,' in *CIDR*, 2019.

[16] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska and T. Neumann, 'Sosd: A benchmark for learned indexes,' *arXiv preprint arXiv:1911.13014*, 2019.

[17] P. Ferragina, F. Lillo and G. Vinciguerra, 'Why are learned indexes so effective?' In *International Conference on Machine Learning*, PMLR, 2020, pp. 3123–3132.

[18] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca and T. Kraska, 'Fiting-tree: A data-aware index structure,' in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1189–1206.

[19] X. Liu, Z. Lin and H. Wang, 'Novel online methods for time series segmentation,' *IEEE Transactions on knowledge and data engineering*, vol. 20, no. 12, pp. 1616–1626, 2008.

[20] P. Li, H. Lu, Q. Zheng, L. Yang and G. Pan, 'Lisa: A learned index structure for spatial data,' in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2119–2133.

[21] H. Wang, X. Fu, J. Xu and H. Lu, 'Learned index for spatial queries,' in *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, IEEE, 2019, pp. 569–574.

[22] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann and T. Kraska, 'Benchmarking learned indexes,' *arXiv preprint arXiv:2006.12804*, 2020.

[23] J. O'Rourke, 'An on-line algorithm for fitting straight lines between data ranges,' *Communications of the ACM*, vol. 24, no. 9, pp. 574–578, 1981.

[24] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau and R. Arpaci-Dusseau, 'From wisckey to bourbon: A learned index for log-structured merge trees,' in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 155–171.

[25] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, 'Wisckey: Separating keys from values in ssd-conscious storage,' *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[26] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska and T. Neumann, 'Radixspline: A single-pass learned index,' in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2020, pp. 1–5.

[27] T. Neumann and S. Michel, 'Smooth interpolating histograms with error guarantees,' in *British National Conference on Databases*, Springer, 2008, pp. 126–138.

[28] J. Ding, V. Nathan, M. Alizadeh and T. Kraska, 'Tsunami: A learned multi-dimensional index for correlated data and skewed workloads,' *arXiv preprint arXiv:2006.13282*, 2020.

[29] (). 'Search on sorted data benchmark,' [Online]. Available: `https://github.com/learnedsystems/SOSD` (visited on 28/05/2021).

[30] P. Ferragina and G. Vinciguerra, 'Learned data structures,' in. Apr. 2020, pp. 5–41, ISBN: 978-3-030-43883-8. DOI: `10.1007/x78-3-030-43883-8_2`.

[31] Microsoft. (). 'Alex - a library for building an in-memory, adaptive learned index,' [Online]. Available: https://github.com/microsoft/ALEX.

[32] Y. Wu, J. Yu, Y. Tian, R. Sidle and R. Barber, 'Designing succinct secondary indexing mechanism by exploiting column correlations,' in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1223–1240.

[33] C. Tang, Z. Dong, M. Wang, Z. Wang and H. Chen, 'Learned indexes for dynamic workloads,' *arXiv preprint arXiv:1902.00655*, 2019.

[34] D. Ho, J. Ding, S. Misra, N. Tatbul, V. Nathan, V. Md and T. Kraska, 'Lisa: Towards learned dna sequence search,' *arXiv preprint arXiv:1910.04728*, 2019.

[35] D. Ho, S. Kalikar, S. Misra, J. Ding, V. Md, N. Tatbul, H. Li and T. Kraska, *Lisa: Learned indexes for dna sequence analysis*, Dec. 2020. DOI: 10.1101/2020.12.22.423964.

[36] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt and P. Dubey, 'Fast: Fast architecture sensitive tree search on modern cpus and gpus,' in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 339–350.

[37] G. Graefe, 'B-tree indexes, interpolation search, and skew,' in *Proceedings of the 2nd international workshop on Data management on new hardware*, 2006, 5–es.

[38] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton and A. Pavlo, 'Surf: Practical range query filtering with fast succinct tries,' in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 323–336.

[39] T. Neumann. (2017). 'The case for b-tree index structures,' [Online]. Available: http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html (visited on 28/05/2021).

[40] T. Neumann. (2019). 'Why use learning when you can fit?' [Online]. Available: http://databasearchitects.blogspot.com/2019/05/why-use-learning-when-you-can-fit.html (visited on 28/05/2021).

[41] A. Crotty, 'Hist-tree: Those who ignore it are doomed to learn,'

[42] C. 2021. (2021). 'Session 7: Data structures - hist-tree: Those who ignore it are doomed to learn,' [Online]. Available: http://cidrdb.org/cidr2021/program.html.

[43] DSAIL. (2021). 'Sagedb: A self-assembling database system,' [Online]. Available: http://dsail.csail.mit.edu/index.php/projects/.

[44] N. Akhtar and A. Mian, 'Threat of adversarial attacks on deep learning in computer vision: A survey,' *Ieee Access*, vol. 6, pp. 14 410–14 430, 2018.

[45] M. Mitzenmacher, 'A model for learned bloom filters, and optimizing by sandwiching,' *arXiv preprint arXiv:1901.00902*, 2019.

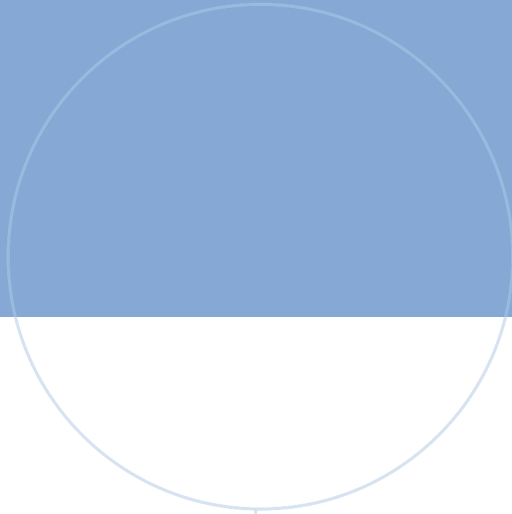
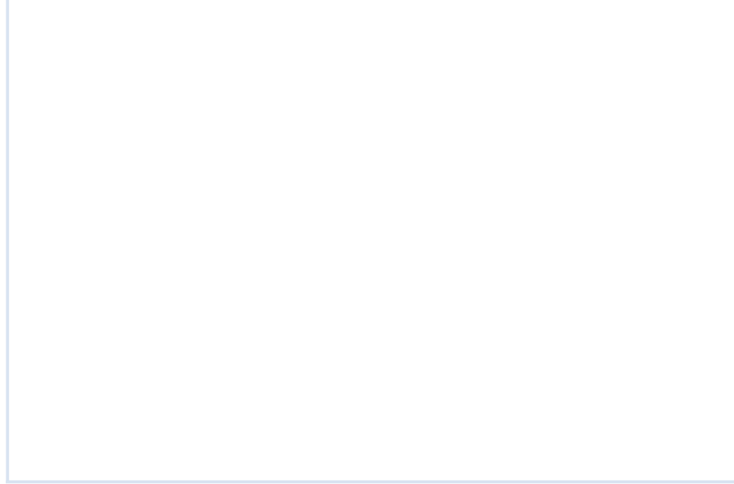
[46] J. Arulraj, J. Levandoski, U. F. Minhas and P.-A. Larson, 'Bztree: A high-performance latch-free range index for non-volatile memory,' *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[47] J. J. Levandoski, D. B. Lomet and S. Sengupta, 'The bw-tree: A b-tree for new hardware platforms,' in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, IEEE, 2013, pp. 302–313.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, 'Attention is all you need,' *arXiv preprint arXiv:1706.03762*, 2017.

[49] D. E. Rumelhart, G. E. Hinton and R. J. Williams, 'Learning internal representations by error propagation,' California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[50] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau and R. Arpaci-Dusseau, 'From wisckey to bourbon: A learned index for log-structured merge trees,' in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 155–171, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/dai.

NTNU
Norwegian University of
Science and Technology