Master's thesis

NTNU
Norwegian University of Science and Technology

Simen Sælevik Tengs

State of the Art Learned Index Structures

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2022

NTNU
Norwegian University of
Science and Technology

Simen Sælevik Tengs

State of the Art Learned Index Structures

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2022

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Abstract

”Learned Indexes” have been a hot topic within the database community after the release of the first learned index, which introduces machine learning to solve the indexing problem that have for a long time, and still are, dominated by traditional structures like B-trees. There have since been created several learned indexes, some of which are created for specific purposes, and some that prove to be very flexible. These flexible learned indexes may serve the same purpose as the B-tree, by supporting a wide variety of workloads and still providing great performance. In this thesis, we provide a technical analysis of two state of the art, general purpose learned indexes; ALEX and PGM-Index. These structures have many of the same characteristics while being built on different approaches. We also cover two learned indexes that have given inspiration to, and introduced many important concepts that are being used by the state of the art learned indexes.

We create a benchmark for the two state of the art learned indexes, by using the authors own implementations of their respective structures in C++. We show that we are able to reproduce similar performance results as reported by the authors, as well as provide a novel side by side comparison of ALEX and PGM-Index on the same data distributions and query workloads. We found that ALEX produced better query performance overall for read-only workloads. PGM-Index performed more consistent for all data distributions and workloads, and had better write performance than ALEX on one of the data distributions.

Preface

I want to thank my supervisor Svein Erik Bratsberg for being a great resource along the way while writing this thesis.

When I first came upon the concept of learned indexes I was instantly intrigued by how well studied concepts of mathematics and machine learning can be utilized to efficiently index data. After examining and testing some of the open source implementations of various learned indexes, I wanted to know more about how and why I got the results I was getting for different datasets. This motivated the research and testing of the state of the art, general purpose learned indexes.

Simen Sælevik Tengs,
25.06.2022

Table of Contents

List of Figures

Chapter 1

Introduction

The term 'Learned Indexes' have been a hot topic within the database community since the exploratory work "The Case for Learned Index Structures"[18] was published by Tim Kraska(MIT) and google in 2017. Indexing in database systems have for a long time, and still is, dominated by robust and reliable data structures like B-trees and its variants. Although reliable, and with set error bounds, these structures does not assume anything about the data distribution, and thus can't take advantage of any patterns in the data. By abstracting index structures to be viewed as models, with a simple input key and outputting a position with a given error bound, the traditional indexing structures can be replaced by learned models with these same properties, and also take advantage of the data distribution. Different learned indexes have been shown to be able to beat traditional data structures by several orders of magnitude in both pure speed throughput and in model space occupancy [10][12][17] [18][21].

1.1 Purpose and motivation

Since the first learned index paper surfaced in 2017, there have been created several different learned indexes, with different approaches and purposes. There are many aspects to consider when creating a learned indexing model, where one may negatively affect another. Some examples are model build times, model space occupancy, pure speed throughput, dynamic workload support and utilization of specific hardware. Some learned indexes specialize in one or more of these aspects, while others are more flexible.

Examples of the learned indexes which have specific purposes are LISA[19] and Radix Spline[17]. LISA is a learned index that can index spatial data, with an idea to be able to replace the R-Tree, as R-trees may have problems with being able to handle very big data loads, which are becoming increasingly more important. Radix Spline is a learned index that specializes in short build times, as it only requires one pass over the data to model it, and is built for read-only workloads. ALEX and PGM-Index are two indexes that stands out when it comes to flexibility. Both structures are fast, have low space consumption and can handle a variety of different workloads. ALEX and PGM-Index are in our estimation the two state of the art, general purpose learned indexes, which have the characteristics that are the most likely to potentially replace the most used traditional structure in today's DBMS systems; The B-tree. There is an incentive to potentially be able to replace a B-tree with a learned index in the future, as performance in distributed database systems is becoming ever so important with the increasing growth of big data.

There have been published several comparisons between learned indexes and traditional data structures for both speed and space occupancy [10][12][17] [18]. The comparison of learned indexes against traditional index structures is not in focus for this thesis. There have also been created a benchmark for some learned indexes and traditional structures, called SOSD-Benchmark[21]. ALEX was unfortunately not released at the time when SOSD-benchmark was created. For this reason, we contribute by creating a micro-benchmark, where ALEX and PGM-Index are tested and compared against each other on the same system, on the same datasets and query distributions. We also provide a technical analysis of the approaches of ALEX and PGM-Index, as well as the Learned Index[18] and the FITing-Tree[13], which are two learned indexes that played an important role by introducing many important concepts from mathematics, statistics and machine learning the state of the art learned indexes builds upon. The chapters are in the chronological order of the release of the learned index structures that is covered. This is done with the aim that the reader can get a good understanding of how the approaches is built up, and how they have improved over time.

1.2 Research questions

Research question 1 (RQ1): *How is general purpose learned indexes constructed to be able to support dynamic workloads with great performance?*

Research question 2 (RQ2): *How does the performance of the state of the art learned indexes PGM-Index and ALEX compare against each other?*

1.3 Thesis content

In Chapter 2, we provide background information and explain terminology that will be useful later in the thesis. Chapter 3 covers the first learned index structure; Learned Index. We cover the approach of this novel structure, as well as the ideas and motivations of the authors. Chapter 4 covers the learned index FITing-Tree, which uses a different approach than Learned Index. For both Chapter 3 and 4, we explain concepts and terminology which is adopted from other fields. In chapter 5 and 6, we cover the approach of PGM and ALEX. This includes the design structure and the approach to query operations. We aim to provide simple explanations and visualizations of the important structures and algorithms. In chapter 7, we present the results of the benchmark of ALEX and PGM-Index, as well as some additional tests for a different configuration of PGM-Index. We also cover the method used for conducting the tests, the limitations of the benchmark, and a discussion of the results. Chapter 3 through 6 addresses RQ1, while chapter 7 addresses RQ2.

Chapter 2

Background & Terminology

2.1 Convex hull

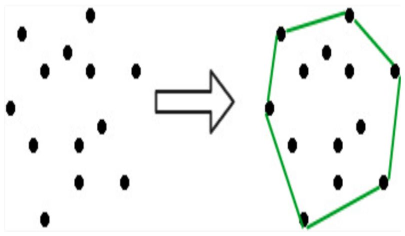


Figure 2.1: Convex Hull[8]

In geometry, a convex hull of a set of points in euclidean space is the smallest *convex* that contains it, which is the smallest subset of 2D points. It can be visualized as the shape that is enclosed by a rubber band stretched around the subset, as depicted in Figure 2.1.

2.2 Index structure

An index structure is structure that is used to optimize the performance of a database by efficiently storing keys. The goal is to optimize query performance by reducing the amount of disk accesses and operations required for processing queries. Index structures should be able to solve the *fully-dynamic indexable dictionary problem*, which is to be able to store a multiset of keys, and be able to efficiently run a series of query operations on the set. Examples are lookup, insert, delete and predecessor. the B-tree and its variants remain the most used data structure in commercial database systems where all these kinds of query operations are required.

2.3 B+ Tree

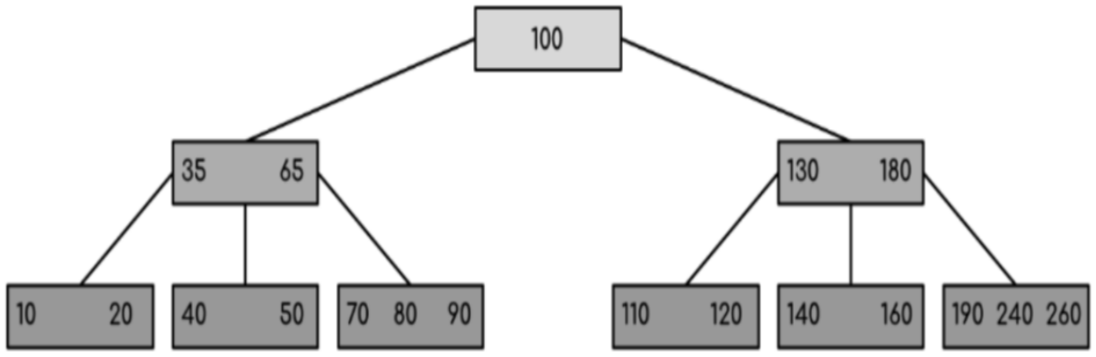


Figure 2.2: B-tree

The B+ tree is a self balancing tree where all its nodes are sorted to support efficient traversal and search. A node in a B+ tree may have more than two children nodes, but always has at least $m/2$ children, where m is the order of the tree. The data is located at the leaf nodes of the tree, containing keys with pointers to data. A standard B-tree have pointers at internal nodes as well as for data nodes, while a B+tree only have pointers at the leaf level. The amount of space allocated at each leaf node is usually the same as a logical page on the system. By having multiple elements per page more elements are kept in the memory's cache, further improving performance. The B-tree have a time complexity of $O(log(n))$ for all query operations, where n is the dataset size. Every time an element in the tree is mutated, the tree balances itself. The reason why the time complexity is logarithmic in relation to the dataset size, is because every time you traverse one level down the tree, the amount of data that has to be processed to find the key you are looking for is halved.

2.4 Indexes represented in the Cartesian Plane

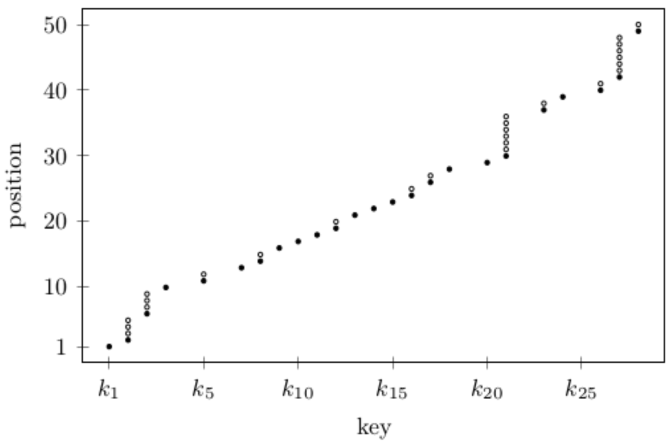


Figure 2.3: A dictionary of ordered keys, represented as 2D-points[31]

Index structures usually store its keys in sorted order to be able to efficiently search and support range queries. When a set of keys are in sorted order and represented as 2D-points in a Cartesian plane, the indexes can be interpreted as models that are mapping the keys to their rank in their sorted order[31]. State of the art learned indexes use linear models to approximate keys, which is made possible by the fact that the indexes are represented in this way.

2.5 Why overfitting is desired for read-only databases

In statistics, a *fit* refers to how well a model is approximating a target function. Fitting is a common term in machine learning when explaining how well a model learns and generalizes to new data. If a model is *underfitting*, the model is too simple for the given problem and won't be able to learn the training data or generalize to new data. On the other hand, if a model is overfitting, the model is too complicated and is learning all the irregularities of the dataset. When the model is overfitting, it will be great at modeling the training data because it knows both the general pattern and the irregularities, but will be bad at generalizing for new data because it has learned the irregularities as concepts. Figure 2.4 shows some examples of over- and underfitting in a typical categorization problem.

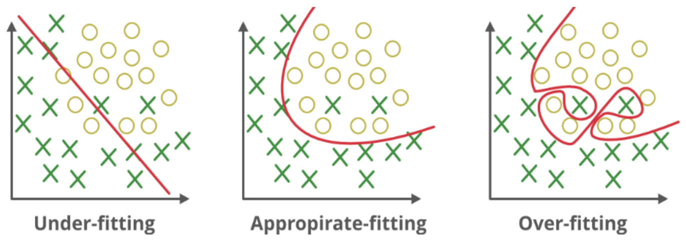


Figure 2.4: Statistical fit - examples [23]

In machine learning, overfitting is usually something to be avoided. In contrast, for read-only databases, overfitting is actually preferred in many cases. What in an indexing scenario would be the equivalent of the *training data*, is the actual data that should be indexed. There is nothing outside the dataset that needs to be generalized over. This is an important aspect of using learned models to index data to keep in mind, especially because it is contrary to the traditional aims of machine learning models. Another good example of this is the B-tree, which is really good at overfitting. This is because being a simple regression tree with simple if-statements to divide the space, it meets the overfitting conditions of being great at learning the ”training data”, but would fail to generalize to new data.

Chapter 3

The first learned index structure

”The Case For Learned Index Structures”[18] have laid the groundwork for a whole new field of study, by introducing machine learning to the decades old field of indexing databases. The first learned index structure will simply be referred to as Learned Index. The key idea behind a ”learned” index structure is that instead of storing the indexes in traditional structures, like B-trees, machine learning models can ’learn’ the distribution of a dataset. Given an input key, the model would then predict the relative position of the record, and take advantage of the patterns in the data when predicting the position. This idea of utilization of the dataset pattern was a very important premise the authors had for building Learned Index. An example they have used repeatedly is that in the hypothetical case where your dataset is a set of continuous integer keys, the B-tree would not be necessary, and would only add an unnecessary layer of complexity. This is because the key itself could be used as an offset. The complexity in this case would drop from $O(log(n))$ to a simple $O(1)$. This example is an extreme case, but it clearly shows how knowing the data patterns can optimize complexity.

In this chapter, we give insight into how structures like B-trees can be replaced learned models with the novel approach used in Learned Index. We explain terminology and key ideas from other fields that have been directly or indirectly applied in Learned Index, some of which have only been briefly mentioned in the original paper[18]. We also look at the key insights and results the authors got from their novel experimentation with Learned Index. Learned Index is an important work because it have paved the way for all learned indexes that have come after, and the model architecture used in the first learned index have been optimized and built upon in new projects that we cover in the next chapters.

3.1 Semantic guarantees

In the paper ”The case for learned index structures”[18], the authors state that range index structures, like B-Trees, are already models. This is an important premise for potentially being able to replace structures like B-trees in real world of database management systems. In in the case of an in-memory, read-only database with a B-tree structure, you abstract the B-tree to a simple model, one may look at the B-tree as a structure that essentially just takes in an input and maps it to an output. In the B-Tree case, the ’model’ takes in a look-up key and maps it to a position in a sorted array of records. This position is normally not the exact position of the record, because for efficiency reasons it is common practise to only index every nth record of a dataset. For B-Trees this usually means indexing the first key of every leaf node, which often is the size of a logical page within the continuous memory region where the sorted array resides. This way, the B-tree can reduce the amount of indexes it have to store without big performance costs. After finding the right page, there have to be run a search until the exact record is found, which means that the B-tree is a mapping from an input key to a position with a min-error of 0 and a max-error of the page size, with a guarantee that the records exists within the page. Thus, the idea behind creating the first learned index was that you could replace the traditional structures with other machine learning models, including neural nets, as long as the same error-bound guarantees are provided.

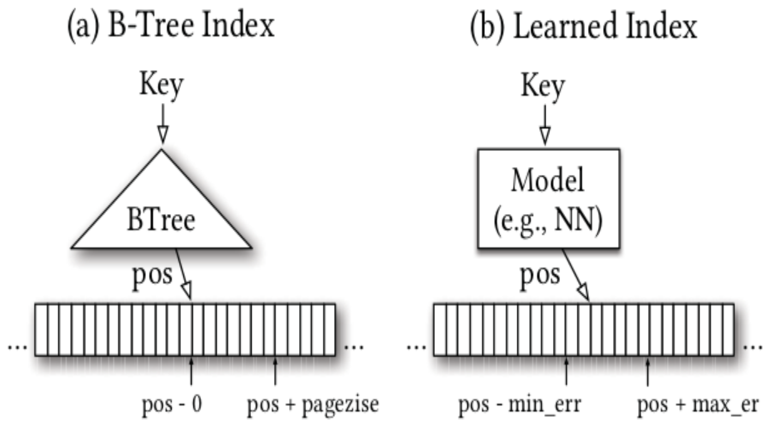


Figure 3.1: Why B-Trees are models

The B-tree provides the error bound for the stored keys, but for new data it has to be re-balanced. The same would be the case for ML-models, where the model has to be ”re-trained” for new data. By executing the model for every key, and saving the best and worst and best prediction, you get error guarantees similar to a B-tree.

3.2 Assumptions and limitations

In this section, we mention some important assumptions and limitations the authors of the first learned index had when creating Learned Index. Some of these points are important to keep in mind when reading the rest of the chapter, especially when analysing the results of the novel experimentation with the first learned index.

1. **All the indexes are run with a dense and sorted in-memory array**

   The data that are being indexed in the tests are dense arrays that are sorted by key. These type of indexes are quite common in in-memory databases, but does not translate well to all type of scenarios, such as write-heavy workloads. We discuss this further in the next chapters.

2. **All the experiments are read-only**

   The approach and experimentation with the Learned Index only focuses on reads. Some learned indexes that have been published after the first learned index, have tackled the challenge of allowing inserts and updates. The high cost of training machine learning models may make inserts seem like a difficult thing to implement in a learned structure, because the models should have to be re-trained for new data.

3. **Model training time is not a part of the experiments**

   The machine learning models used in the first learned index, ranging from linear regression models to neural networks all needs to be trained. The time it takes to train the models are not a part of the experiments that the authors conducted. Instead, they created a framework called The Learning Index Framework(LIF). LIF generates different index configurations, optimizes them, and tests them automatically. For neural networks, LIF extracts all the weights from a Tensorflow model[29], and generates index structures in C++ based on the model specifications[18]. This way, what is being measured is only the time to actually run the index, and removes the training time from the equation. Model training time will be an important factor when considering learned indexes as replacements for traditional structures in database systems in the future. The authors state that for simple distributions, like a randomized dataset, the models that are being used are also simple, and only needs a few passes over the data. This only leaves a few seconds of training time for a 200M record dataset.[18]. For neural networks, training time may take up a few minutes for complicated patterns.

3.3 Approximating the Cumulative Distribution Function

The authors of the first learned index [18] observed that when a machine learning model maps an input key to a position within a sorted array, it becomes an approximation of the Cumulative Distribution Function(CDF). Therefore, the term CDF in the realm of learned indexes means the mapping from a key to a relative position within a sorted array. The term is very similar to the CDF from statistics, where the CDF of x is the proportion of keys less than x. Figure 3.2 represents an approximation of the CDF [21].

As stated earlier, the B-tree is also a mapping from an input key to a relative position within a sorted array. For this reason, the B-tree may also be viewed as a way of remembering the CDF of a dataset. If the B-tree is indexing every nth record, the error bound of the B-tree's CDF would then be $n-1$. In ML-terminology, you can look at a B-Tree as a simple regression tree.

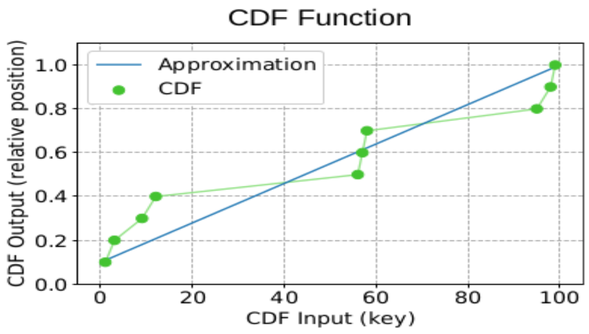


Figure 3.2: Approximating the CDF - example

One way of finding the starting point of a key in a sorted array, is to find the probability for there to exist a key which is lower or equal to the lookup key. This can be written as $F(key) = p(X \leq key)$. Simply multiplying this by the number of keys N will give the estimated position within the sorted array; $p = F(key) * N$. If the CDF model over the empirical data distribution is perfect, every estimate would then produce the actual position within the array.

3.4 Model type

3.4.1 A single deep or wide model

When creating the first learned index, the first approach that was considered was to use one single model to index all of the data in the datasets, and training the model end-to-end. An important question to consider was whether they should use a wide or a deep model. When indexing data, wide and deep models have their own advantages and disadvantages in terms of execution cost and what kind of data patterns it's good at modelling. To make the case for using both deep and wide models for indexing clear with edge-case examples, let's say you have a really wide model of linear models, and a very deep neural network. The wide model should be able to model any small nuances in the dataset because each linear equation forms a regression line(CDF) through a small subset of data, which should not give very large error bounds within each sub-range of data, even though there could be irregularities in the dataset. The deep model would be better for complicated patterns and bigger ranges of data, where the CDF is smooth and without irregularities. This is visualized in Figure 3.3, where the whole pattern looks smooth, and could likely be modeled well by a neural network, whereas the sub-range in focus would be better modeled by linear models because of the linear tendency of the data distribution at that range.

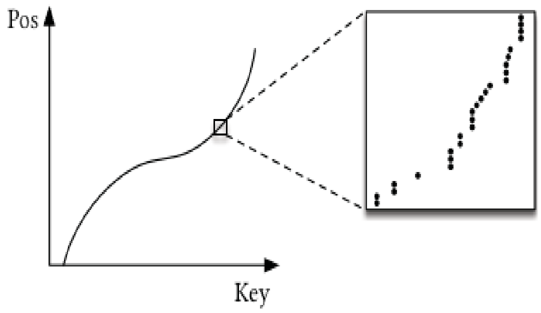


Figure 3.3: Indexes as CDF [18]

The cost for each of these examples above can be quantified by the number of pure math operations needed to execute the models. A linear regression model can be represented as $(y = mx + b)$, where y is the prediction and x is the lookup key. This type of model only requires one multiplication, one addition and if needed one rounding. The wide model would then cost two times the width of the model in pure math operations.

For the deep model, the amount of math operations needed to execute the model equals the width of the model squared. For a simple neural network this usually consist of at least multiplying the input data with a weight to represent the signal strength, and then applying an activation function to modulate to neurons activity [2]. For a deep model this is executed in many layers, where the output of one layer is the input to the next. Every additional layer are usually called the hidden layers of the network. This can be represented by a square matrix multiplication, which means that for every intermediate layer in the model, there has to be executed an additional square matrix multiplication.

Even though using a single model is possible, it is not optimal. An important reason for this is that it does not take advantage of the capacity of the memory, as the above mentioned models is very tiny compared to a standard B-tree. For example, a linear model consisting of four 8-byte double values, are much smaller than the internal nodes in the B+tree, which stores keys and pointers. Because of this, the authors of the first learned index had to find a way to increase the model size to improve the accuracy, without a massive decrease in performance in terms of speed [4]. The reason why we laid out some simplified examples of single models and their execution cost, is to get a better understanding as to why ramping up the size of these models to fit the size of the memory will result in a huge decrease in performance because they would demand so many math operations to execute one index. This problem became the important design-space challenge the authors had to figure out.

Another reason why using a single model is problematic, is that it becomes hard to reduce the prediction error in a very big dataset to a size where it would then be sufficient to execute a last-mile search. In an example used in he original paper [18], the authors state that to reduce the prediction error from 100M to an order of hundreds is difficult with a single machine learning model, while reducing he prediction error from 100M to 10k, and from 10k to 100 is much easier. This is because the model then only has to model a subset of the original data size.

Because of the challenges, they decided to utilize an idea called *hierarchy of experts*, with inspiration from the work ”The Sparsely-Gated Mixtrure-of-Experts Layer” [27]. The key idea is that instead of training one single model, a hierarchy of models is trained, where each model specialized in one subset of the data. This is an idea that has been well researched in the machine learning community, and in the next sub chapter we briefly explain the origins of the mixture of experts work.

3.4.2 Hierarchy of experts

The term ’Hierarchy of Experts’ were introduced all the way back in 1994 by the Jordan and Jacobs[16]. They proposed a system to solve nonlinear supervised learning problems by dividing the input space into a nested set of regions, and to train a number of models, each of which specializes in one of the regions. They called each of the specialized models ’experts’. The system also contained managing models called ’gating networks’, which looks at the input data, and decides which model to rely on for that particular input data. The *hierarchical mixture of experts* architecture is essentially a tree of experts and gating networks, where all the experts sit at the leaf nodes. The architecture is shown in Figure 3.4. Each expert produce an output vector for each input vector. The gating networks receive a vector as input and produce outputs that are a subset of the data from the input space.

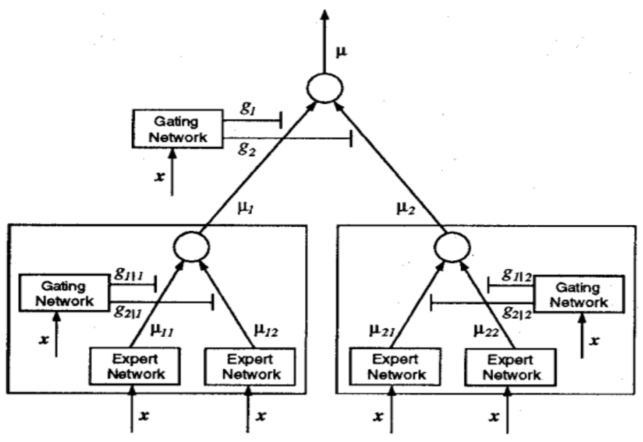


Figure 3.4: Two layer hierarchy of experts network [16]

This approach was better than just averaging models in a way that does not depend on the particular training case. Because the models specializes in a subset of training cases, each model does not need to be trained on data where they are not picked by the gating networks. This way they can ignore all the data that they are not good at modeling. The key idea is to make each expert focus on predicting the right answer for the cases where it is already doing better than the other experts. Generally, this system does not take very good advantage of the data for smaller datasets, as the data have to be fragmented over all the different experts. This means that for small dataset, it can't be expected to do very well. It can, however, make very good use of very large datasets.

The authors of the first learned index considered a similar approach to the original work of hierarchy of experts[16] when trying to train the model end-to-end. The main challenge with using this approach is the way the gating networks choose which expert to give the input to. They choose the next expert by multiplying the input by a trainable weight matrix, and then applying a *Softmax* function. The Softmax function is a generalization of the logistic function to multiply dimensions, and is used to normalize the output of the network to a probability distribution that is being used to choose the experts. This becomes very inefficient when the system has a large number of models, e.g 100.000, which is the amount they used in Learned Index. The performance would suffer because a search has to be performed to find the max of the soft-max at every gate.

3.5 Recursive Model Index(RMI)

Since the release of the mixture of experts work, there have been published a lot of literature on modeling multiple experts in machine learning. The particular model that the authors of the first learned index ended up basing their model on is called ”Sparsely-gated Mixture-of-Experts Layer” [27]. The main difference from the original mixture of experts work, is that this sparsely-gated model, as the name suggests, have trainable gating networks that only selects a sparse combination of experts to process for each input. This inspired the Recursive Model Index(RMI) approach. Simply explained, the RMI architecture is a tree-like structure of models, where the model on top takes in a key as input and produces a position prediction as an output. The model then uses that output to select another model. This process occurs recursively until the last model predicts a position very close to the keys correct position in the array. A simple representation from [18] is shown in Figure 3.5. The whole model is recursively whittling down the amount of data that the each model is expected to model well. This is done stage-wise, very similar to a B-tree.

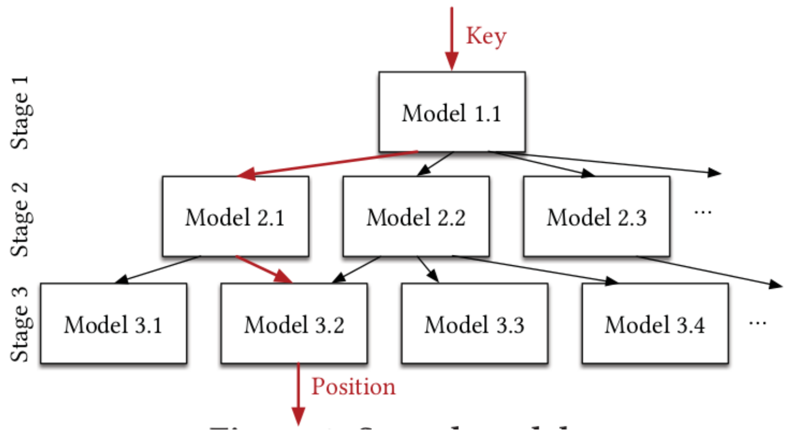


Figure 3.5: Staged Models [18]

One key idea with the RMI is that it can have different types of models in the hierarchy of models. The models can range from simple linear functions to neural networks. What type of model to choose for a given range of data depends on the complexity of the data distribution in that range. We look closer at the implications of using different types of models in the next section. If for a given input key, the model is only choosing linear functions for each stage until it predicts a position, the whole model then becomes similar to a piece-wise linear function. The difference is that in this case it's not necessary to perform a search of which piece it's in at any point. Instead, the model is selecting the next model based on a series of multiplications. This idea is the foundation of the learned indexes FITing-Tree and PGM-Index.

Formally, the model for predicting a position $y \in [0, N)$, given an input-key x, can be written as $f(x)$. The *training loss* formula for the top layer L_0 in the RMI is then the following: $L_0 = \sum_{(x,y)} (f_0(x) - y)^2$ [18]. In machine learning terminology, the loss function is an evaluation of how well a model is evaluating data. There exists many different types of loss functions, each specified for the each model type. The RMI deals with trying to predict a continuous index y based on an input x, which is called a regression prediction. The goal is therefore that the training loss for the model should be lowered for each stage down the RMI hierarchy.

If the prediction is far off from the actual data, which is the actual index in this case, the loss function would give a high number. The training loss formula is computing the squared difference between the index prediction and the actual index. Because of the squaring of the loss, predictions that are far away from actual index are penalized heavily in comparison to less deviated predictions. To put this in simple terms, what this formula is computing is how far off the actual index is it possible to get by going down one stage in the hierarchy.

After training the model at stage 0, the model k at stage l then have the prediction formula: ${f_l}^{(k)}(x)$. Similar to stage 0, the loss function for all models in stage l is:

$$L_\ell = \sum_{(x,y)} (f_\ell^{(\lfloor M_\ell f_{\ell-1}(x)/N \rfloor)}(x) - y)^2$$

Figure 3.6: Model loss (L_l) [18]

We see here that the function ${f_l}^{(k)}(x)$ is recursively predicting a position based on the previous prediction ${f_{l-1}}^{(k)}(x)$. Each stage is trained in iterations with loss L_l until the model is complete[18]. Each iteration is one pass over the data. A big difference from the RMI to a B-tree, and to a tree structure in general, is that each model in the RMI does not have to be the same size, like each node in the B-tree. This is because multiple models at one stage may choose the same model in the stage below, as shown in Figure 3.5, where both Model 2.1 and Model 2.2 chooses Model 3.2. As with the hierarchy of experts work[27], the predictions can be looked at as the next model that should know how to model that key-input even better. To clarify - the different models in the RMI are each trained on a subset of the data, but the predictions they make are actual position estimates, not a range.

An important benefit of the RMI architecture, is that the size of the whole model does not affect the cost of executing an index. Because the output from one stage is used directly at the stage below, the model does not need to perform any searching between the stages. This way only the models that are used to predict the index needs to be activated when running the index, which means that the model complexity does not affect the model size. As mentioned earlier, with no searching between the stages only leaves a series of multiplications. The whole index can therefore fit into a sparse matrix multiplication. This leaves a huge potential for performance benefits by using TPU's and GPU's to calculate the indexes in the future.

3.6 The potential for GPU's & TPU's for learned indexes

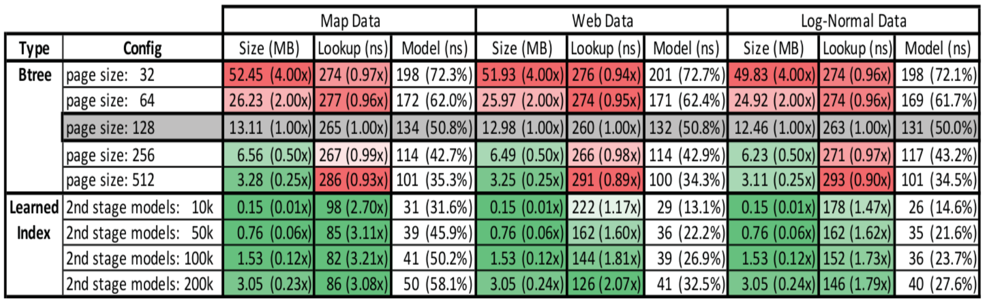
As stated earlier, a concern relating to using neural networks to learn the distribution of datasets, is the high cost of executing and scaling the large amount of math operations needed. Graphic Processing Units(GPU) and Tensor Processing Units(TPU) are being increasingly upgraded, and are the important back-bones to be able to meet the high computational demands of neural networks. Because of the performance benefits of utilizing hardware for neural networks, and the fact that most CPU's have SIMD capabilities, the authors of the first learned index[18] speculates that in the cost of executing ML models might be negligible in the future, and states many times that GPU/TPUs will make the idea of learned indexes even more valuable. All the experiments with the first learned index was run with normal general instruction CPU, and even then showed promising results.

Google's and Nvidia's TPU's can perform thousands of neural net operations per second[2]. Building the index structures to compliment and use the rapidly developing processing in the future leaves a huge potential for performance increases. An example of utilizing hardware for learned indexes is the learned index APEX[20], released in mid February of 2022. APEX is an index structure that is optimized for Persistent Memory(PM). APEX offers high performance, persistence, concurrency and instant recovery. This learned index is based on the learned index ALEX[10].

3.7 Results from novel experiments with the first learned index

The results showed in Figure 3.7 shows the performance of a 2-stage RMI model versus a read-optimized B-tree. The 4 rows for the Learned Index shows the results for different stage sizes - how many models there are in the second stage. The B-tree is run with 5 different page-sizes. The B-tree with a page size of 128 keys is grayed out as the reference point, as this configuration gave the best results for the B-tree. For all B-Tree experiments they used 64-bit keys and 64-bit payload/value[18]. All of the datasets in

Figure 3.7 are Integer datasets, where two of them are real world datasets containing map data and weblogs, both with 200 million entries. The third dataset is a synthetic dataset with a logarithmic distribution of 190M unique keys.



		Map Data			Web Data			Log-Normal Data		
Type	Config	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	52.45 (4.00x)	274 (0.97x)	198 (72.3%)	51.93 (4.00x)	276 (0.94x)	201 (72.7%)	49.83 (4.00x)	274 (0.96x)	198 (72.1%)
	page size: 64	26.23 (2.00x)	277 (0.96x)	172 (62.0%)	25.97 (2.00x)	274 (0.95x)	171 (62.4%)	24.92 (2.00x)	274 (0.96x)	169 (61.7%)
	page size: 128	13.11 (1.00x)	265 (1.00x)	134 (50.8%)	12.98 (1.00x)	260 (1.00x)	132 (50.8%)	12.46 (1.00x)	263 (1.00x)	131 (50.0%)
	page size: 256	6.56 (0.50x)	267 (0.99x)	114 (42.7%)	6.49 (0.50x)	266 (0.98x)	114 (42.9%)	6.23 (0.50x)	271 (0.97x)	117 (43.2%)
	page size: 512	3.28 (0.25x)	286 (0.93x)	101 (35.3%)	3.25 (0.25x)	291 (0.89x)	100 (34.3%)	3.11 (0.25x)	293 (0.90x)	101 (34.5%)
Learned Index	2nd stage models: 10k	0.15 (0.01x)	98 (2.70x)	31 (31.6%)	0.15 (0.01x)	222 (1.17x)	29 (13.1%)	0.15 (0.01x)	178 (1.47x)	26 (14.6%)
	2nd stage models: 50k	0.76 (0.06x)	85 (3.11x)	39 (45.9%)	0.76 (0.06x)	162 (1.60x)	36 (22.2%)	0.76 (0.06x)	162 (1.62x)	35 (21.6%)
	2nd stage models: 100k	1.53 (0.12x)	82 (3.21x)	41 (50.2%)	1.53 (0.12x)	144 (1.81x)	39 (26.9%)	1.53 (0.12x)	152 (1.73x)	36 (23.7%)
	2nd stage models: 200k	3.05 (0.23x)	86 (3.08x)	50 (58.1%)	3.05 (0.24x)	126 (2.07x)	41 (32.5%)	3.05 (0.24x)	146 (1.79x)	40 (27.6%)

Figure 3.7: Learned Index vs B-Tree [18]

3.7.1 Optimal model types for a 2-stage RMI

The authors found that for the top layer in a 2-stage RMI, neural networks with two hidden layers and 8-16 nodes in width worked the best. To tune the neural nets in the first stage, they ran a simple grid search. A grid search defines a search space as a grid of hyperparameter values and evaluate every position in the grid to find the best possible configuration for the model. For the second stage, simple linear models worked the best. This fits the presumptions described previously where a neural network would be better on the first pass because of the complexity of the data distributions, and for the last-mile prediction, linear models are optimal, as the added complexity of the neural networks are not necessary.

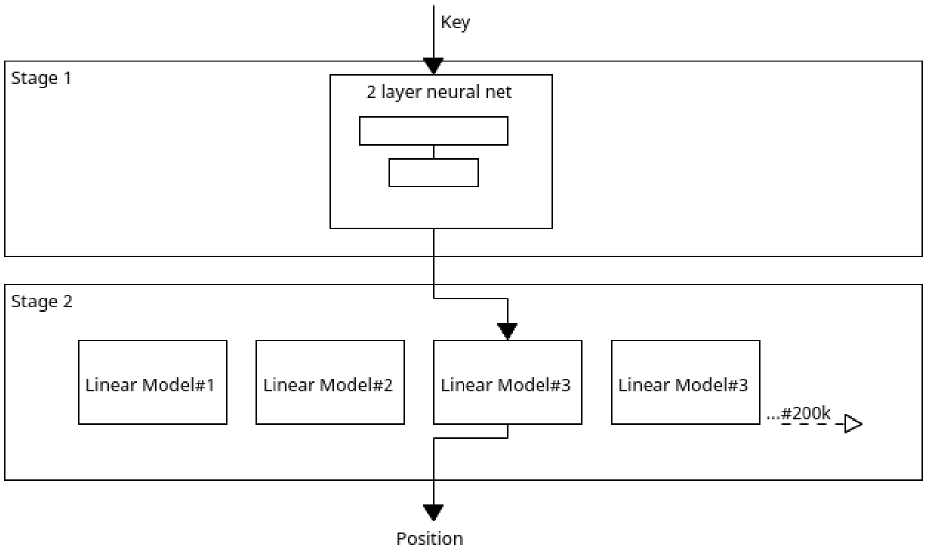


Figure 3.8: Two stage RMI - example

Figure 3.8 shows a simplified example of a 2 stage RMI with a two-layer neural network at the first stage, and 200k linear models at the second stage. For the three different datasets the lookup times with this RMI configuration were 86ns for the maps dataset, 126ns for the weblogs data set and 146ns for the logarithmic data distribution. For almost all the different configurations, the learned index is up to 1.5× 3× faster in speed, and up to two orders of magnitude smaller in size. We see that the amount of models in the second stage has a lot of impact on both size and lookup time. For the map data the lookup time does not change a lot for the different second stage sizes. This makes sense because the keys in the map datasets are longitudes of user events, which are mostly linear and have few irregularities[18]. This scenario shows that when the distribution is easy to learn the amount of models in the last stage has less impact on performance. The weblogs datasets have unique request timestamps as keys, and are much harder to learn, which explains why adding more linear models increase performance to such a degree.

Chapter 4

FITing-Tree

The first learned index created much interest within the database community, and created interest for people to continue to build upon the ideas from [18]. One of the challenges with the RMI that were addressed, is that the model size and query time trade-off can be quite difficult to control because the size and the performance of the RMI depends on the actual distribution of the input data and the complexity of the models that are being used. This motivated a novel index structure called *FITing-Tree*[13]. The FITing-Tree is the first learned index to use piece-wise linear functions, with a error bound that is defined at construction. FITing-Tree uses a $B+tree$ to index the linear functions. Just like the RMI, it combines learned models and traditional structures, the difference being that the RMI uses B-trees as a fallback method if the learned models are not able to model the distribution sufficiently well, while the FIT-ting Tree uses the B-tree in its actual index structure.

The piece-wise linear functions used in the FIT-ting tree approximates the CDF of the data the same way the models used in the first learned index does. The error bound is therefore a fixed constant that defines the maximum distance between the prediction and the actual position of the key that is being looked up. This index structure has two parameters to manage the space-time trade-off, and finding a suitable error bound constant. These parameters are is lookup latency(*ns*) and storage budget(*MB*)[13]. The novelty of this index structure is the ability to specify this space-time trade-off to *FIT* a given scenario based on the error bound specification, as well as having support for paging and inserts. The structure is therefore addressing some of the important key challenges of Learned Index.

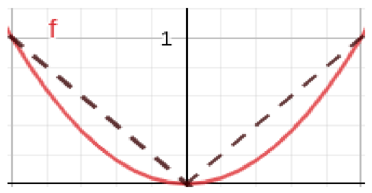
The FITing-Tree introduces many important concepts that have been adopted by the learned index *PGM*, which we cover in the next chapter. This is the main reason why we cover this the FITing-Tree in this thesis. Even though the authors of the PGM-index took inspiration from the FITing-Tree, they state in the original paper that; *"The computation of the linear models residing in the leaves of the FITing-tree is sub-optimal in theory and inefficient in practice."[12].* In this chapter, we look at the key concepts of the FITing- tree's design, how it manages to allow for inserts and its performance.

4.1 Piecewise Linear Approximation(PLA)

Piecewise linear functions are a well studied concept of mathematics and statistics. A piecewise linear function contains a number of linear segments, also called pieces, defined over an equal number of intervals. The entire piecewise linear approximation of a dataset can be expressed with this formula:

$$F(x) = \begin{cases} a_0 \times x + b_0 & \text{if } x < p_0 \\ a_1 \times x + b_1 & \text{if } x \geq p_0 \text{ and } x < p_1 \\ a_n \times x + b_n & \text{if } x \geq p_1 \text{ and } x < p_2 \\ \dots & \\ a_n \times x + b_n & \text{if } x \geq p_n \text{ and } x < p_n \end{cases}$$

A simple example of a piecewise linear approximation with $n = 2$ segments on the distribution $f(x) = x^2$ can be expressed like this:



$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

Given an error bound ε, each segment should only consist of data points that are at most ε away from the adhering linear approximation line. In other words, what defines a segment is the region of data that can be reached by a linear equation at most ε way from the prediction. The size of the error bound ε will therefore determine the size of each segment and the total amount of segments needed. This means that there exists an optimal number of segments for a piecewise linear approximation with a given error bound ε for a given dataset. There exists several *optimal* piecewise linear approximation algorithms, but they are often expensive in terms of memory and/or execution runtime[13]. One example of this is a dynamic programming algorithm[24], which have a runtime complexity of $O(n^3)$ and memory complexity of $O(n)$. In the next chapter, we see how the PGM-Index actually have an algorithm that have a linear time complexity while finding the exact optimal number of segments based on an error constant ε.

4.2 FITing-Tree design

As a default, the FITing-Tree have a B+ tree at its inner nodes. This may, however, be replaced by other structures. Instead of having fixed sized pages at the leaf nodes, as with a standard B+ tree, the FITing-Tree have linear functions at each leaf node to approximate one region of data. Each linear model points to a *segment* of data that is at most the error bound ε away from the linear approximation. Each segment is sorted by key, but successive segments does not have to be allocated contiguously, like the the pages at the leaf nodes of B-trees. Figure 4.2 shows an example of a clustered FIT-ting tree model. The FIT-ting tree saves memory consumption by only needing to save the starting key of a segment, the slope of the linear function and a pointer to a segment needs to be saved at each leaf node, much like how a B-tree saves memory consumption by only needing to index the first record of every page. The structure also supports non-clustered index by adding a fourth layer in the design, in which they called the indirection layer. The indirection layer is a list of pointers that is the same size as the data, but sorted by key. This secondary index method can be used to improve performance for queries over attributes that are not sorted and contains duplicates. When discussing further, we have focused on the clustered FITing-Tree.

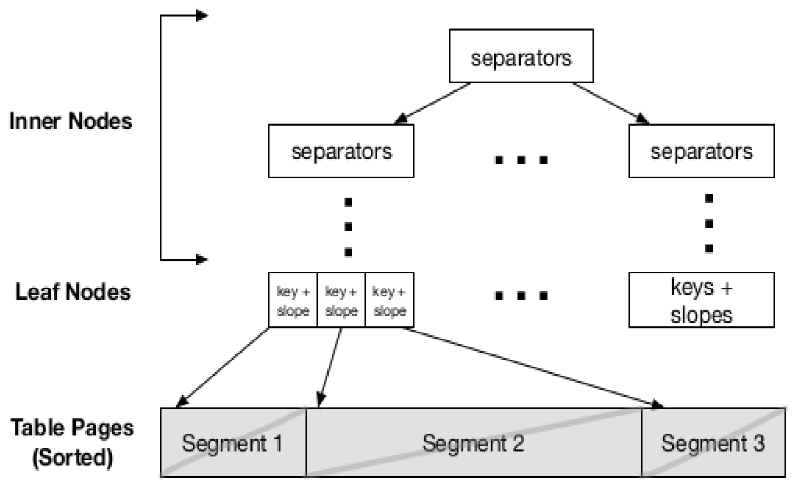


Figure 4.2: Clustered FITing-Tree model[13]

4.2.1 Segmentation algorithm

To support inserts, while staying efficient, the authors of the FITing-Tree found that they needed to find a segmentation algorithm that is linear in time and only requires one pass over the data. All optimal segment algorithms were therefore out of the question. They found a solution that was quite fast, had constant memory usage and guarantees the maximum error specification[13]. The FITing-Tree utilized an algorithm called the *Shrinking Cone*. The shrinking cone algorithm is built bottom up, by extending the each segment without violating the error bound ε.

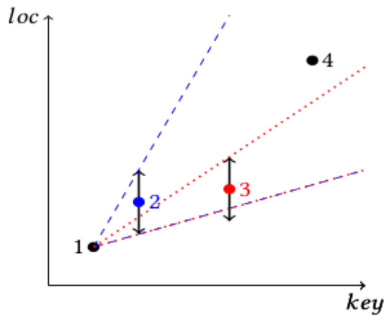


Figure 4.3: Shrinking cone[13]

The algorithm they made is very similar to the an algorithm called *Feasible Space Window (FSW)* [24], the difference being that Shrinking Cone only considers increasing functions, which in the case of indexes would be sorted datasets. The shrinking cone algorithm may also create disjoint segments[13]. The first step in algorithm is to create a segment with a origin point. The next point that is added creates a ”cone” from the origin point that is the error constant ε away from the second point. This process continues until a point is not within the cone of a previous point, then that point becomes the origin point of a new segment. Figure 4.3 shows an example of Shrinking Cone, where point 4 is outside the previous cone and does therefore not satisfy the constraints. This means means that there will be at least one point within the previous points that does not satisfy the error threshold if the segment contained all those points. The new point will therefore instead become the origin point of a new segment. They evaluated the amount of segments in the ShrinkingCone algorithm on real world datasets, and compared the results to the optimal amount of segments. They found a ratio difference was around between 1.05 in the best case and 1.6 in the worst case.

4.3 Lookup & insert strategy

Lookup

As stated earlier the FITing Tree uses a B+ tree at it's base - so to perform a point lookup, the first step is to traverse the B+ tree from the root by using standard traversal algorithms[13]. The runtime complexity for finding the segment that the input key belongs to is $O(log_b((p))$, where b is the number of pointers to child nodes within the tree, also called the fanout of the tree, and p is the number of segments. Then, the approximated position within the segment are given by simply subtracting the key from the start element of the segment, multiplied by the slope of the segment;

$$pred\_pos = (key - segment\_start) \times segment\_slope$$

After the finding the approximated position, a binary search is performed until the true position is found. For range-queries, there are often many segments that needs to be searched, which means that for a clustered FITing-Tree where the segments are aligned contiguously, the FITing-Tree can search all the start positions of contiguous segments until the start key is outside the range-request, and retrieve all the indexes between the start and the end of the requested range.

In-place insert

The authors of the FITing-Tree considered two different insert approaches, the first one being an in-place strategy. In applications where the database always needs to remain sorted, new records needs to be inserted *in-place*, which means that the index is inserted into its correct position at insertion-time. This is quite normal for many databases, and a typical B+tree utilizes an in-place insertion strategy by leaving empty space in each page for new records. When the page is full, the leaf node where the full resides splits in two, and the resulting changes propagates up the tree. It becomes more complicated for the FITing-Tree because of its error bound guarantee. Because the error of the inserted key and the interpolated prediction is not known, there has to be performed a check anywhere the key moves. Naturally, the keys surrounding the inserted key also needs to be moved to either side to remain a sorted order. This means that in the worst case - all the keys in the segment needs to move, all without violating the error guarantee. This process appears to be awkward as you have to check the error for all the keys that should be moved. The solution they came up with is to extend every page size to be the segment size $+2\varepsilon$. The segment is always placed in the middle of the page, such that keys can be moved in either direction without violating the error guarantee specified by the user. The whole in-place insert strategy is shown below:

- Locate the position within the page where the index should be placed
- Check if the index is to the right or the left of the middle of the segment
- Shift all keys one step in the opposite direction
- When the page fills up:
  - Re-approximate the segment with the Shrinking Cone algorithm
  - New segments that are created are propagated up the tree, and old segments are deleted

Delta insert

When order guarantees are not important, a delta insert strategy can be used for better performance. The delta insert approach is simpler and more straight forward than inserting in-place. Instead of moving keys, every segment has its own sorted buffer where new keys are added. When the buffer is full it is merged with the original segment and re-approximated. It is important to note that the result after re-approximation could be the same as it was before, if none of the keys violates the error bound ε. This method does change how to run an index, because the error threshold needs to add the size of the buffer to the user specified error bound. The extra error is added in the background without the user knowing. This approach drastically removes overhead by not having to move keys around, and have a better overall performance than the in-place strategy, as shown in the tests on real world datasets from [13]. It shows that the delta strategy performs better for higher error thresholds, which makes sense because then the buffers are larger and the structure can perform fewer merging operations and re-approximations. For lower error thresholds, the in-place performance worked the best. This is because when the error is smaller there is created more segments with fewer keys within each one, thus, having fewer keys that needs to be shifted for each insert.

4.4 FITing-Tree performance

The lookup performance of the FITing-Tree is comparable with a implementation of a B+tree called the STX-tree[28]. The performance was comparable to that of a traditional structure, but it did not offer significant increase in performance for any index size for any of the real world datasets that they tested on. The space consumption, however, were reduced in up to four orders of magnitude. This clearly shows how a learned index may be used for index compression. The delta-insert strategy had a comparable performance to a B+tree that uses fixed size paging, but had a significantly worse performance than a full B+tree implementation. The FITing-Tree could not handle as big of a writeload as the full B+ tree. The reason for this is because of its need to split the nodes when they reaches its maximum size. The FITing-Tree also had a slightly lower insert throughput than a B+ tree with fixed size paging, because of its need to re-execute the segmentation process.

The FITing-Tree has a simple, yet elegant design, and works great for compression. The lookup and insert throughput is, however, not that impressive. The FITing-Tree have a high search cost that only depends on the disk-page size, which means that it cannot take full advantage to the trends in the data distribution. This search cost grows with the node size, which slows down the query operations[12].

Chapter 5

PGM-Index

Both Learned Index and FITing-Tree brought much interest to the authors of the Piecewise Geometric Model Index(PGM-Index)[12]. Like the FITing-Tree, the PGM-Index is a structure consisting of piecewise linear approximations, but addresses some of the FITing-Tree's key limitations. The PGM-Index offers updates and inserts and is therefore fully dynamic, while constructing a recursive system of linear models that all have the optimal number of segments based on a given error constraint, by utilizing both learning and geometry. The PGM-Index does not rely on any traditional data structures, and is therefore the first *fully learned* index, unlike the RMI and the FITing-Tree. The structure is flexible, and offers different configurations, including *Compressed PGM-Index*.

In this chapter we look at how the recursive structure of linear models is constructed, and how running indexes on the structure works, how the they construct the optimal number of segment in linear time($O(n)$. We also cover the three different configurations of PGM, one of which we use for our tests in Chapter 7. Because of its flexibility and performance both in speed and compression, we believe that the PGM is the state of the art learned index in its line of approach.

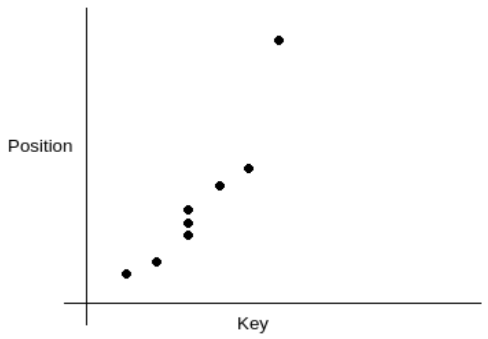
5.1 Optimal Piecewise Linear Approximation

As discussed in Chapter 4, there exists an optimal number of segments in a piecewise linear approximation(PLA). The optimal number of segments is the minimal amount of segments needed to approximate a set of points with a given error bound ε. The FITing-Tree used the heuristic approach ShrinkingCone with a linear time complexity $O(n)$, but yields a number of segments which is comparable to the optimal number of segments at best, and performed even worse in practise. The FITing-Tree authors only considered complex algorithms to find the optimal amount of segments, such as dynamic programming, which yields a time complexity of $O(n^3)$. The authors of PGM-Index noticed that the problem of finding the optimal amount of segments have been well studied for other problems, such as lossy compression and similarity search in time series[6][7][5].

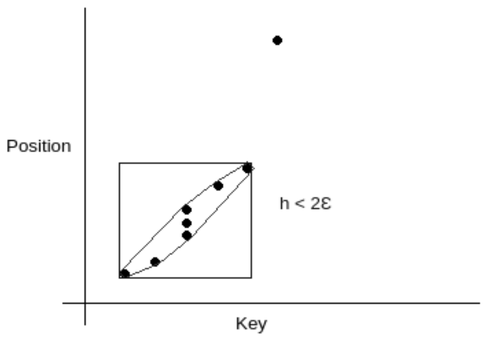
5.1.1 Geometric segmentation algorithm

The segmentation algorithm used in the PGM is a implementation of the optimal segmentation algorithm from [32], which makes use of the result that for a set of points that is non decreasing in their x-coordinates, there exists a streaming algorithm that constructs the optimal number of segments with a linear time complexity $O(n)$, based on an error constant ε. These algorithms make the use of geometry, and take the optimal PLA-model problem and reduce it to a geometric problem of constructing *convex hulls* of a set of points.

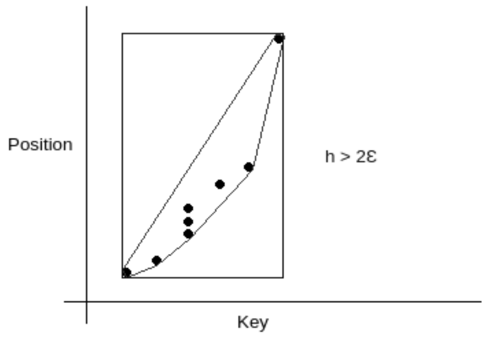
With a sorted set of keys k, the segmentation process starts by encapsulating a convex hull around the key k_i to $k_i + 1$. For every increment of i, a check is performed to see if the height of the convex hull is lower than 2ε. If the rectangle enclosing the convex hull is within the bound, the set is extended with the new key $k_i + 1$. When a key is outside the bound, a segment is constructed by the line that cuts the rectangle of height 2ε in half. To clarify; the straight line $y = ax + b$ is then the actual segment, and is saved as two floats and one key.



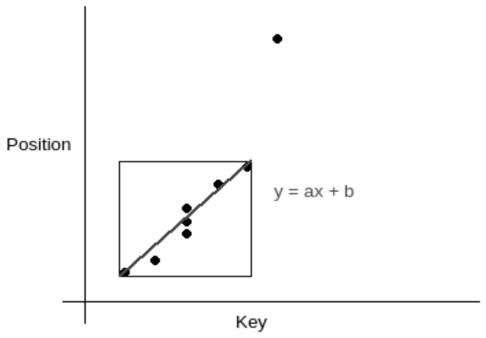
(a)



(b)



(c)



(d)

Figure 5.1: Segmentation process

Figure 5.1 shows a visualisation of the segmentation process. Figure 5.1a shows a set of keys in a Cartesian plane, and sub figure 5.1b shows a convex hull in red wrapped around the seven first keys after iteration $i = 6$ from the first key k_0, and a rectangle in blue that is still within a error constraint 2ε. After incriminating i one step further as shown in 5.1c, the height of the rectangle is no longer within the error bounds. Therefore, a straight line is formed in the previous iteration, cutting the rectangle in two equal halves, as shown in Figure 5.1d. This is now the first segment, and the process is restarted again from the next point. This process takes linear execution time and space usage $O(n)$.

5.2 Model structure

5.2.1 Indexing the PLA-model

After having the optimal PLA-model, the next step is to index the different segments within the model. The indexes for the segments are necessary for being able to search for a specific key among the segments/linear models. As discussed, the FITing-Tree used a B+tree as a default structure to index all the segments. The PGM could also have used structures like B-trees on at its inner nodes to be used for searching among the keys, but it would then not fully take advantage of the fact that a single linear model; $y = key \times slope + intercept$ takes a constant space and query time. The authors of PGM found a way to construct a recursive system where every level, including the root node of the structure, only consist of one PLA-model.

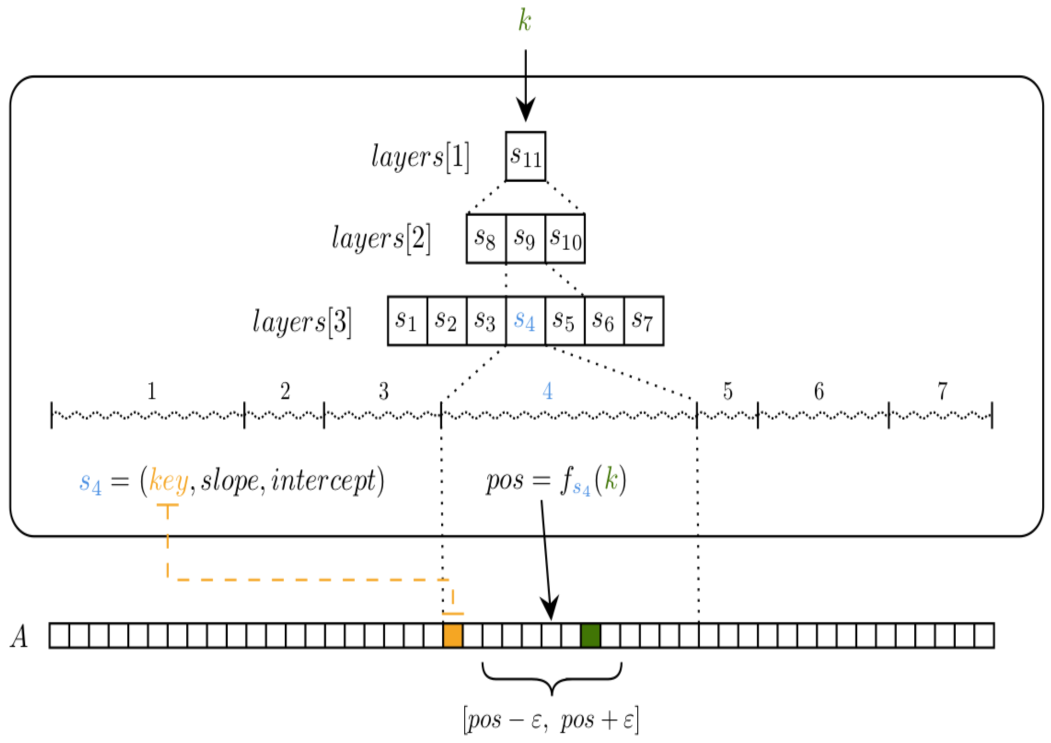
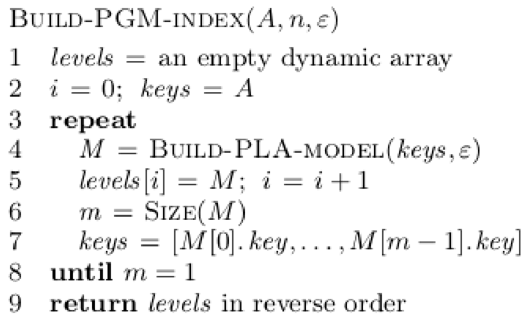

k
$layers[1]$
s_{11}
$layers[2]$
s_8 s_9 s_{10}
$layers[3]$
s_1 s_2 s_3 s_4 s_5 s_6 s_7
1 2 3 4 5 6 7
$s_4 = (key, slope, intercept)$
$pos = f_{s_4}(k)$
A
$[pos - \varepsilon,\ pos + \varepsilon]$

Figure 5.2: Mapping from key to position in simplified PGM-index[31]

The process of building the structure is quite simple. The first step is to execute the optimal segmentation algorithm on an array A based on error ε. This will form the bottom layer in the recursive structure, consisting of the linear equation(key, slope and intercept) of all the segments. Then, the first key of every segment is put together to a new sub-array of A. The segmentation algorithm is then computed on this sub-array to form a new layer. This process is done until the segmentation process only yields one single linear model. The PLA-model with one single layer will form the root node of the recursive structure, as shown in Figure 5.2, where $s11$ is the final PLA-model constructed from the algorithm.



```
BUILD-PGM-INDEX($A, n, \varepsilon$)
$levels$ = an empty dynamic array
$i = 0$; $keys = A$
repeat
    $M$ = BUILD-PLA-MODEL($keys, \varepsilon$)
    $levels[i] = M$; $i = i + 1$
    $m$ = SIZE($M$)
    $keys = [M[0].key, \ldots, M[m-1].key]$
until $m = 1$
return $levels$ in reverse order
```

Figure 5.3: PGM-Index Construction Pseudo Code[12]

The PGM structure is built bottom-up. This is interestingly, in total contrast to the RMI, where the construction of the recursive structure begins at the root node. What this difference means practically is that with the RMI, the most general model for the whole dataset is constructed first, and then the construction process is iterated, the models get increasingly more precise, whereas in the construction of the PGM index, the most precise model is built first, and the models are getting increasingly more generalized as the process is iterated.

5.2.2 Running a lookup operation

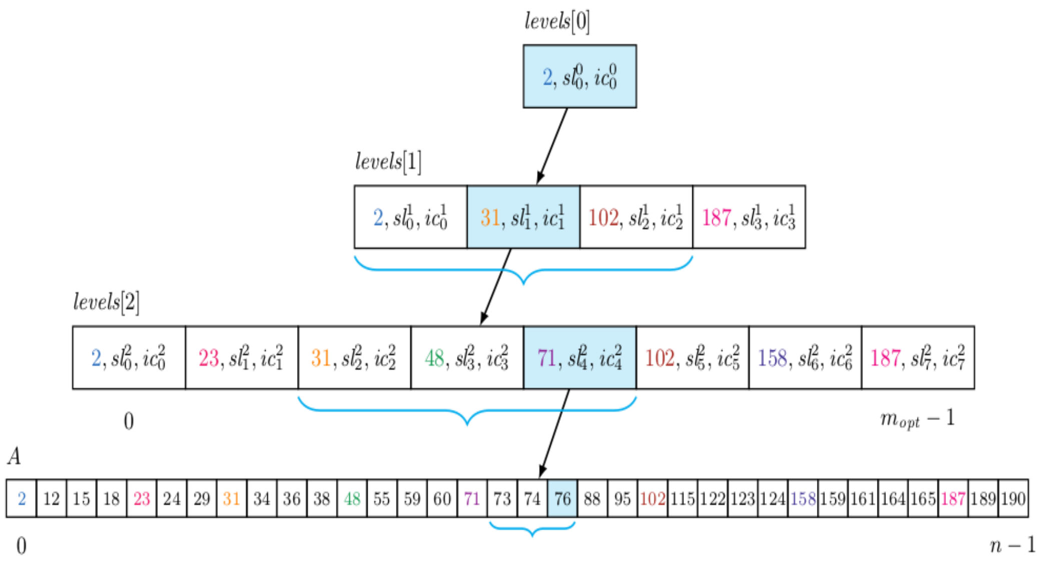


Figure 5.4: Lookup operation on PGM-index[12]

Figure 5.4 shows an example from [12] of a lookup operation on a key k = 76. A good way of getting an understanding as to how the recursive PGM structure works with no indexing of the segments is to go through an example. A detailed walk-through of the query operation can be found in [12], but to get an easier understanding, we provide a concise walk-through with further explanation of the the steps, while only using the most necessary mathematical notations. We go through the query operation shown in Figure 5.4 of looking up key $k = 76$ on array A, with error guarantee $\varepsilon = 1$ in the following steps:

1. $levels[0]$

   The process start at the root node of the structure($levels[0]$). The key $k = 2$ which is already in the equation from the building process, is replaced with the lookup key $k = 76$. The slope and intercept remains the same. Then, the linear equation is computed.

2. $levels[1]$

   The result y of the linear equation at the root node is the position of the next level. In this example $y = 1$, so we find $Level[1][1]$. From this position, a binary search is performed to find $k = 76$ from position $[y - \varepsilon$ to $y + \varepsilon] = [1-1, 1+1] = [0,2]$. The searching range from position $0-2$ are depicted with the light blue bracket. From the search we found that key $k = 76$ falls between 31 and 102. level[1][1] is then computed after replacing key $k = 31$ with lookup key$k = 76$.

3. $levels[2]$

   The result y of the linear equation at level[1] equals to 3. We therefore find level[2][3]. We then search for $k = 76$ from position $[y-\varepsilon, y+\varepsilon] = [3-1, 3+1] = [2,4]$. From the search in the previous step we found that key $k = 76$ is bigger than the key $k = 71$ in position 4, and since this is the right-most position within our error bound, we compute levels[2][4] after replacing key $k = 71$ with lookup key $k = 76$.

4. Array A

   Because the last linear model was computed from the bottom layer in the structure, the linear equation equals a position in the actual array A at most ε away from from the actual position. The equation in $levels[2][4]$ equals to 17, so we search for the lookup key from position $[y - \varepsilon\ ,\ y + \varepsilon] = [17-1, 17+1] = [16, 18]$. The binary search within A finds the correct key at position 18, which is ε away from the prediction from $levels[2][4]$.

5.3 Insertion strategy

As with the FITing-Tree, insertions becomes more complicated for PGM-Index than with traditional structures, because of the error guarantee. An insertion may have a time-complexity of if a key can be inserted at the end of an array A while not violating the error constraint, which is the case in some real world scenarios like for time series. For many scenarios this is not the case. When a key is to be inserted into a random position in array A, an insert strategy based on the merges from the logarithmic approach of the Log Structured Merge Tree(LSM-tree)[25] is used. The algorithm is modified to work for a learned index, and takes a logarithmic insertion time.

The approach starts by considering a series of PGM indexes built over sets $S_0..,S_b$ of of keys. These sets are empty or have a size of 2^i where i ranges from 0 to the worst case $\theta(log(n))$. For an input key x, the first empty set S_i is found and a new PGM-index is constructed with the union of that first empty set, all the previous sets and x. The merged set becomes the new set S_i, and the other sets are emptied. After this merge, x is in its correct position in the sorted set S_i. The union have a linear execution time because all the sets are sorted.[12]

The reason why the inserts take $O(log(n)$ execution time is because the maximum amount of merges for a given any inserted key is$\theta(log(n)$, and the full set of merges might include all inserted keys. The linear time complexity $O(1)$ of each merge then gives a time complexity of $O(log(n))$ for every insert.

5.4 Compressed PGM-Index

PGM offers a configuration called Compressed PGM-Index, which should lower the space consumption of the index compared the the default configuration. We test the performance/space usage trade-off of the compressed configuration compared to the default PGM in Chapter 7. As seen in the construction process of the PGM, there are created many ε-approximate segments that all consists of linear functions; $key \times slope + intercept$. The process already yields the optimal number of segments, so the way to compress the space consumption even further is to compress each segment by compressing the slopes and intercepts at each segment.

The $intercept$ in a segment s_j can be compressed by utilizing the fact that an $intercept_j$ finds a key k in s_j computing the segment with the intercept equal to $k - key_j$. this intercept can then be stored as an integer, thus saving space[12]. The intercepts can be accessed in $O(1)$ time by utilizing the succinct data structure from [26], by exploiting the fact that n is always bigger than the intercept, thus being able to reduce the problem of accessing the intercept to that of finding the rank(intercept, n).

The compression of the slopes are more difficult, and the authors had to construct a novel algorithm to solve the compression problem. The algorithm creates a list of all the slopes, and encodes the slopes into one of the slopes that is higher in the hierarchy of segments, as the slope is then still guaranteed to be within the scope of the higher level segment. The number of slopes that needs to be saved is then reduced. The algorithm finds the smallest amount of unique slopes that needs to exist in the PGM-index while still preserving the error bound ε.

Chapter 6

ALEX

ALEX is an in-memory, updatable learned index that builds upon the key insights and RMI approach used in Learned Index[18], by recursively constructing a tree shaped structure of learned models. ALEX addresses some of its key challenges, and introduces some novel technical contributions. One of the main drawbacks of the original RMI was that it was not suited well for supporting Dynamic workloads. The main reason for this was that it used a sorted and dense array. The cost for moving records around to make space for new inserts would become high, much like we saw with the FITing-Tree's in-place strategy. With inserts in the original RMI, the model predictions would also get worse as the dataset changes over time, because they are trained for a certain areas in the distribution and would need to be retrained often. In this Chapter, we cover the technical contributions of ALEX, and its key differences from Learned Index. We go through how these contributions makes up an updatable data structure with great performance, as well as cover the process of running different query operations on the structure.

6.1 Technical contributions

ALEX does not include neural nets

One of the characteristics of Learned Index was that the models residing in the RMI structure could be a variety of different models, ranging from linear models to deep neural networks, depending on what gave the best performance at each level. Usually neural nets were only used for the top level of the RMI for complex patterns. The authors of ALEX learned from conversations with the authors of [18] that the performance gain of having the option to choose neural nets over linear models were not worth it because of its added complexity. The authors of ALEX also state that they have independently verified this[10]. ALEX does therefore not continue with this hybrid index layout, and instead only uses linear models at every level of the structure.

Exponential search

After the predictions from the linear models, there has to be performed a local search within the error bounds to find the exact position of the record. The authors of ALEX found that exponential search worked better than the binary search method used in Learned Index. They found that when the models are accurate enough, and the predictions were close to the correct position, there was really no need for the nodes within the structure to know the actual error bounds because of the use of exponential search.

Model-based insertion

Before inserting a record, ALEX runs a prediction on the key to check for what position the key would have been predicted to reside in. They called this method *model-based insertion*. Learned Index never modifies the position of keys in order to fit the models. The Model-based insertions increase the accuracy of the models, and therefore decreases the lookup delay.

Gapped Arrays

ALEX uses a way to allocate free space at the data nodes at the leaf level of the structure, which is called Gapped Array. The idea is similar to *An adaptive packed-memory array*[3], which is an array of size $\Theta(N)$ with N elements that keeps gaps in between the elements such that only a minimal number of records needs to be shifted for every insert or delete. The B+Tree also keeps allocated space for inserts, but have the space allocated at the end of the array. Gapped Arrays work especially well in conjunction with exponential search. A bitmap is stored to efficiently be able to skip gaps when searching within a node, that checks whether a position in the array is a gap or a record. The searching is always executed over the bitmap instead of the actual array for lower space consumption.

ALEX is constructed with bulk loading

ALEX supports bulk loading operations. Bulk loading is a concept that is used when you need to support importing of large amounts of data in a short period of time, and is done through the structure of the particular database system. ALEX uses bulk loading when constructing the structure, either at initialization or at retraining. The Bulk loading is a greedy algorithm based on linear cost models that for each node decide if the nodes should be internal nodes that is used for redirecting, or data nodes. The Bulk load algorithm also decide the fanout of the tree.

Dynamic RMI

One of the drawbacks of the original RMI is that it had a static dept of two or three. If ALEX used a static RMI, it would hurt the insert performance if the distribution is hard to model[10]. ALEX can be updated dynamically at runtime by using linear cost models that decide if nodes should update. ALEX therefore creates linear models when its needed, and the structure grows deeper and wider until the data nodes approximate the same number of keys [11].

6.2 Model structure

The ALEX structure is similar to the recursive structure of Learned Index, where the position computed from one node is used to find the next model. One key difference, as stated above, is that ALEX only contains linear regression models at all levels of the structure. A model of ALEX's are shown in 6.1. The computation from the linear models choose which pointer to traverse down the structure. In contrast to PGM-Index, ALEX's data nodes only stores two floats for the slope and intercept, and no default key. All the nodes which is above the leaf level(internal nodes), contain an array of pointers to children nodes in addition to the linear regression model. The amount of pointers are always a power of 2, which allows the nodes to split without retraining the nodes at lower levels. The First learned index does not require nodes to split, because of not having so possibility of inserts. We cover ALEX's expansion and splitting mechanisms for its nodes in Section 4.

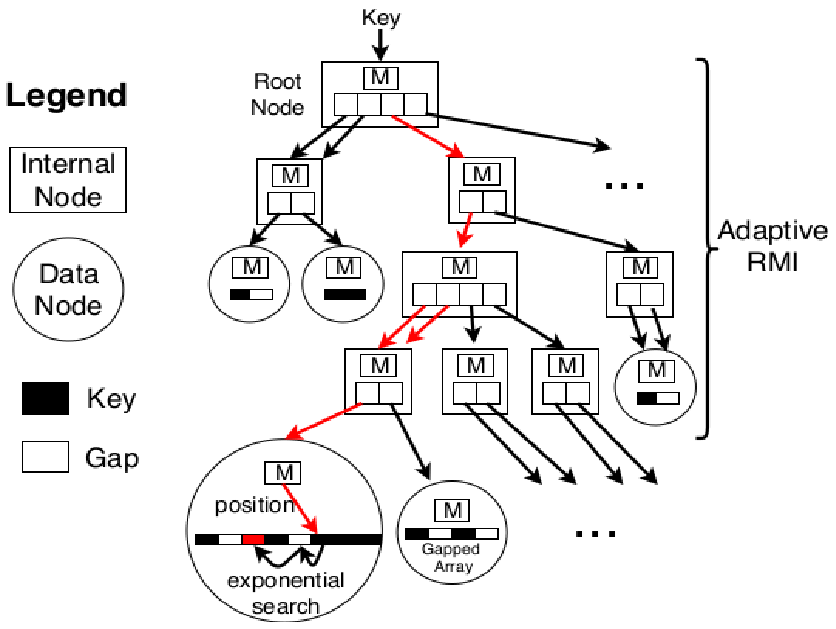
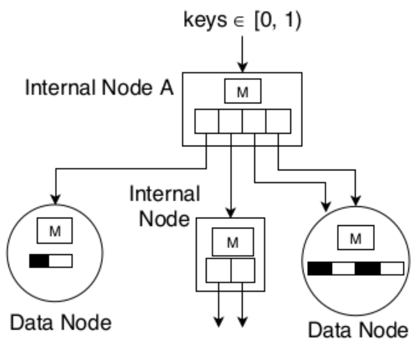
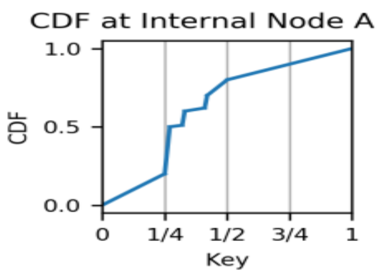


Figure 6.1: ALEX Design[10]

The RMI in the first Learned Index have models that are fit to the data distribution, which leads to the fact that the more accurate the models are, the data nodes become more similar in size. Having nodes of equal size is not the goal of ALEX. Instead, the goal of the models is to partition the keys such that the data nodes at the leaf level have a data distribution that is as linear as possible, such that the linear models can effectively fit the keys. ALEX reaches this goal by having a flexible way to partition the space, by finding out what part of its data have a linear trend. Figure 6.2a shows a visualization of how an internal node may split its key-space $[0,1)$ to two data nodes and one internal node. The RMI in the first index would either assign all of the key space to only data nodes or only internal nodes. Figure 6.2b Shows the CDF of the key space $[0,1)$. The first data node is responsible for keys from$[0,1/4)$, and the second data node are responsible for the key space from $[1/2,1)$, both of which have linear trends.



(a) Data nodes for key space$[0,1)$



(b) CDF for key space $[0,1)$

Figure 6.2: Internal nodes allow different resolutions in different parts of the key space[10]

6.3 Query operations

6.3.1 Lookup and range queries

Performing a lookup operation on ALEX is straight forward. Figure 6.1 shows a lookup operation, where the red arrows indicate the traversal down the model structure. Given an input key, the internal node computes a location in the array of pointers. This process continues until a data node is reached. The linear model in the data node computes a prediction of the lookup keys position within the array of the data node. If the predicted position is not exact, exponential search is then performed until the exact position if found. Performing range queries are performed by performing a lookup for the lowest key in the lookup range, and then performs a scan over the bitmaps of the next data nodes until the last key in the range is found.

6.3.2 Inserts

To insert a record, the key that is to be inserted is used to compute the traversal from the root node to the data node, the same way as a lookup is performed. When a data node is reached, model-based insertion is performed by executing the data nodes linear model to predict where in the Gapped Array the key should be placed. If the predicted position is occupied by another key, a new gap is created, by shifting the other keys in the direction of the nearest gap. The key is then inserted into that gap. If the predicted position is already a gap, the key is inserted and the process is complete. If the data node is full, ALEX expands or splits the data node to create more space. We cover these mechanisms in the next section.

Insertion in the Gapped Array is similar to *Gapped Insertion Sort*, or *Library Sort*. Standard Insertion Sort have a complexity of $O(n^2)$, because each sort takes $O(n)$ time. They showed in [**insertion-sort**] that when introducing evenly distributed gaps, the insertion time complexity is $O(logn)$ with high probability. This also applies to insertion into the Gapped Arrays of ALEX. This insertion complexity is interestingly the same as for the PGM-index.

6.4 Node expansion and node splits

If the distribution of keys changes after several inserts, then some leaves will become overloaded with data. The nodes in ALEX's structure therefore needs to be able to expand or split to make space for new inserts. The authors implemented a metric to decide when a data node should split or expand. They implemented upper and lower density limits which are a number between 0 and 1, where 0 is empty and 1 is full. The default limits in ALEX are set to be 0.6 and 0.8. Its important that the data nodes never actually becomes full, because the performance of inserts would suffer as the number of gaps in the data node decreases. A B+tree also keeps a density metric, and have a similar upper and lower density limits for its leaf nodes[14].

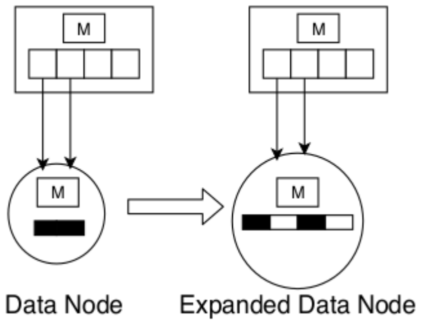


Figure 6.3: ALEX Node Expansion[10]

To expand a node, a new Gapped Array is allocated with the size of the number of allocated key divided by the lower density limits. The data node is at at the lower density limit after expansion with the added gaps. Figure 6.3 shows a simple example of a node expansion. The nodes can use this expansion mechanism until the nodes have reached a *maximum node size*. If the node violates the upper density limits while also having maximum node size, a node split is performed. The new nodes after a split are responsible for half the keys each.

A split may be done sideways like in a typical B+Tree, or downwards. A downwards split converts the old node to a inner node and creates a number of child nodes. This is very similar to what happens in the initial construction of the ALEX structure. The result is that ALEX grows increasingly deeper, possible slowing down queries until a total reconstruction is performed[11]. A node may also split before the maximum node size is reached if is is more efficient. This is calculated with a linear cost model that evaluates the lookup performance based on *(a)* average number of exponential search iterations, and *(b)* average number of shifts for inserts.[10].

Chapter 7

Benchmark

In this chapter we run a series of tests on the two general purpose state of the art indexes. In the method section, we explain how the tests are conducted and what types of datasets are used and why. The tests are run with three different workloads, which is read-only, 50/50 read/write and write only. The results show how the two indexes performs for all the different workloads, data distributions and dataset sizes. The y-axis of the graphs are logarithmic, so for some tests it may be hard to see the magnitude of the differences of some results. We therefore added some additional visualizations of the results where only one specific workload or data distribution is showed. For PGM-Index we have added additional tests for the compressed configuration, as well as model build times. ALEX's performance metric is in operations per second, while for PGM-Index we use nanoseconds per operation. At the end we show the results of ALEX and PGM compared against each other, and discuss these results. For the comparison all metrics are converted to nanoseconds per operation.

7.1 Method

7.1.1 Benchmark tests

Both ALEX and PGM-Index have open source implementations in C++[15][22], which we have used for the tests for our benchmark. We refer to a single test as the accumulated query performance for both either ALEX or PGM-Index for a specific query workload for a specific dataset distribution and size. Both projects offer the possibility to run the indexes on custom datasets and workloads. The projects each have unique options for running the indexes, i.e choosing different configurations for PGM-Index, and choosing bulk load sizes for ALEX. The projects have an option to run a simple *benchmark*, which runs a user-specified number of indexes. We have only made slight configurations to the source codes, which was solely to display additional information and to make the ALEX accept the same file format as we used for PGM. The tests have been run in their respective projects in isolation, but have been run with the exact same datasets and workloads. All the tests have been conducted on the same system: Linux-5.4.091-generic-x86 with an Intel Core i7-8565U CPU and 16GB of RAM. The tests have been conducted before the start of writing this thesis, during a preparatory project with course id IT3915.

PGM-Index benchmark

When running the C++ benchmark for PGM-Index, random indexes are generated, where each index is anywhere from 0 to dataset size -1. Each index is run and the PGM-Index returns a structure ”ApproxPos” containing the approximate position of the key in sequence and the bounds of a range with the size of $2\varepsilon + 1$. The sought key is guaranteed to be found if present. The indexes are run in batches, and we chose the same batch size of 10% of the dataset size for both PGM and ALEX. The dataset sizes ranges from 10 to 25 million, so the amount of queries per batch will range from 10 to 250000. We calculate the average performance of the query operations of all the batches for each test. We choose a few default values for the error bound ε for each batch, and calculate the average of those results to get a general result for each dataset.

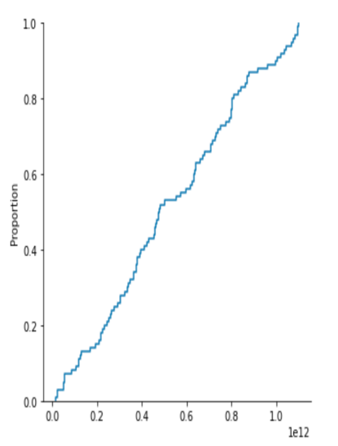
ALEX benchmark

ALEX's C++ implementation as mentioned also have a benchmark option. The benchmark takes in a handful of arguments, including bulk-load size of initialized keys, insert fraction and batch size. We choose the batch size as for PGM at 10% of the dataset size. When running the ALEX benchmark the cumulative throughput for all the batches are calculated. For the tests we have bulk loaded ALEX with 50% of the dataset size, except for the tests with the biggest dataset size of 25 million keys, as the computer used for testing could not manage it. For that specific dataset size we had to reduce the bulk load size to 30% of the dataset size. The bulk loaded keys are combined with randomly generated payloads.

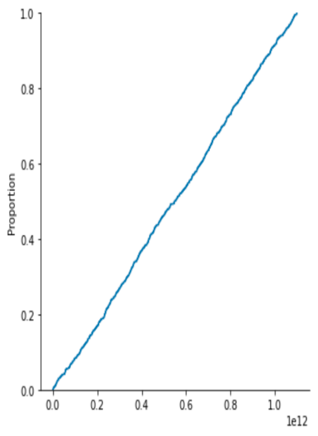
7.1.2 Datasets

The main tests will be run on two different synthetic data distributions that we created. The first distribution is simply signed integers with completely random values. The second dataset consists of pseudo-random signed integers that makes up a triangular distribution. We want to be able to see how well the two index structures manages to run query operations on two different patterns, one of which is uniformly distributed but has quite big gaps in the key values, and one which has a clearer pattern and is more dense. We go more into detail as to how and why we used these distributions below. For each of the distributions, we have made 7 datasets of different sizes, which all will be used for the tests. The dataset sizes are 10^2, 10^3, 10^4, 10^5, 10^6, 10^7 and $2.5 * 10^7$. Each of the 14 datasets have 64bit signed integer keys, and are stored in separate binary files. We sort the keys before saving them in the files. This is because neither ALEX nor PGM-Index have functionality to sort the datasets before running the benchmarks.

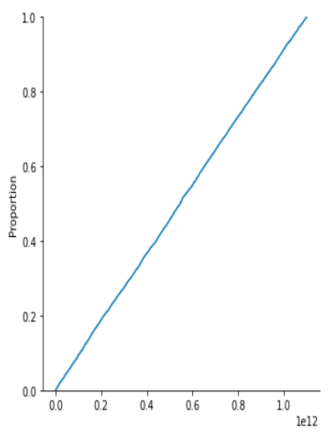
Uniform distribution



(a) 10^2 keys



(b) 10^3 keys



(c) 10^4 keys

Figure 7.1: Uniform distribution datasets

The first dataset consists of randomly generated integers which creates a uniform distribution of keys. The CDF of the distribution will resemble a straight line $[y = x]$ the bigger the dataset gets. This is visualized in Figure 7.1, that shows the CDF of the first three sizes of the uniform distribution. Both ALEX and PGM uses linear approximations to model data, so the linear trend in this dataset should be fairly easy to model. It should not matter that the value of the keys are far in-between in this dataset, because approximations capture trends in the data, and are agnostic to key density, where sparse keys can be captured as well as a trend with dense keys[12]. Even though this is a synthetic dataset, there are many real world scenarios where big datasets have a uniform data distribution. Examples are the YCSB dataset[9] and datasets containing randomly sampled Facebook ID's[30]. Both of these examples have been used to test learned indexes, such as ALEX in their original paper[10] and the SOSD-Benchmark[21].

Triangular Distribution

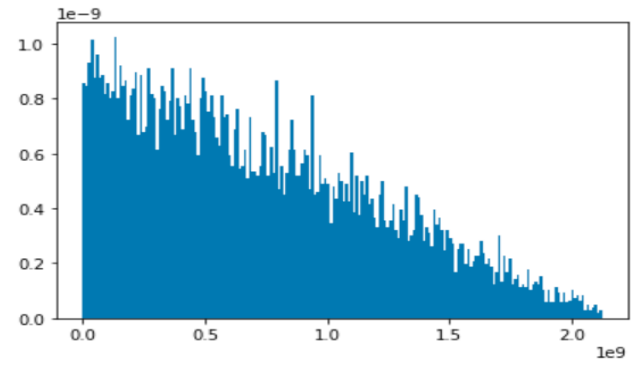
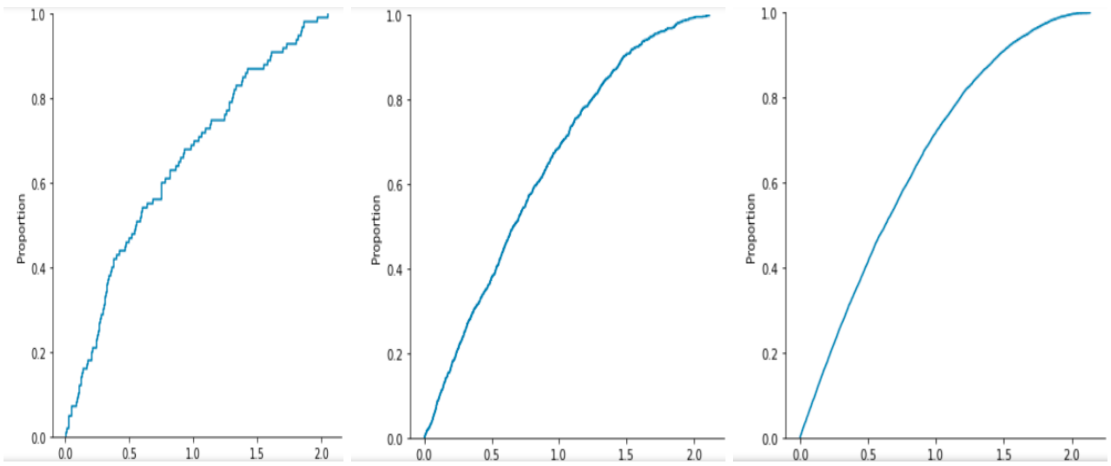


Figure 7.2: Triangular dataset of size 10^4 - Histogram

The second data distribution that we use for the tests are created with a triangular distribution of keys. A triangular distribution is a continuous probability distribution shaped like a triangle. The formula for creating the values doesn't create random values, but have more weight towards a certain number. The value of the keys will be closer to each other towards that number, and farther apart at the ends of the distribution. We use the Python package NumPy's[1] implementation of the triangular distribution. The formula is shown below:

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x}{(r-l)(r-m)} & \text{for } m \leq x \leq, \\ 0 & otherwise. \end{cases}$$

The parameters are chosen such that the numbers may be any integer, but the the lower and upper bounds are always filled out, but with more weight towards a certain number.



(a) 10^2 keys (b) 10^3 keys (c) 10^4 keys

Figure 7.3: Triangular distribution - CDF

When this data distribution is sorted by key, the CDF becomes a **logarithmic** distribution. This becomes more apparent the bigger the datasets become. This is visualized in Figure 7.3. A logarithmic distribution for a dataset is a typical real world scenario. An example of this are be timestamp indexes where each key represents a time of day, where a certain time of the day is posted more frequently than the rest. Another example are book popularity on amazon, where each key represents the popularity of a book. A logarithmic distribution have been used to test learned indexes in [10][21][12][13].

7.2 Results

7.2.1 ALEX

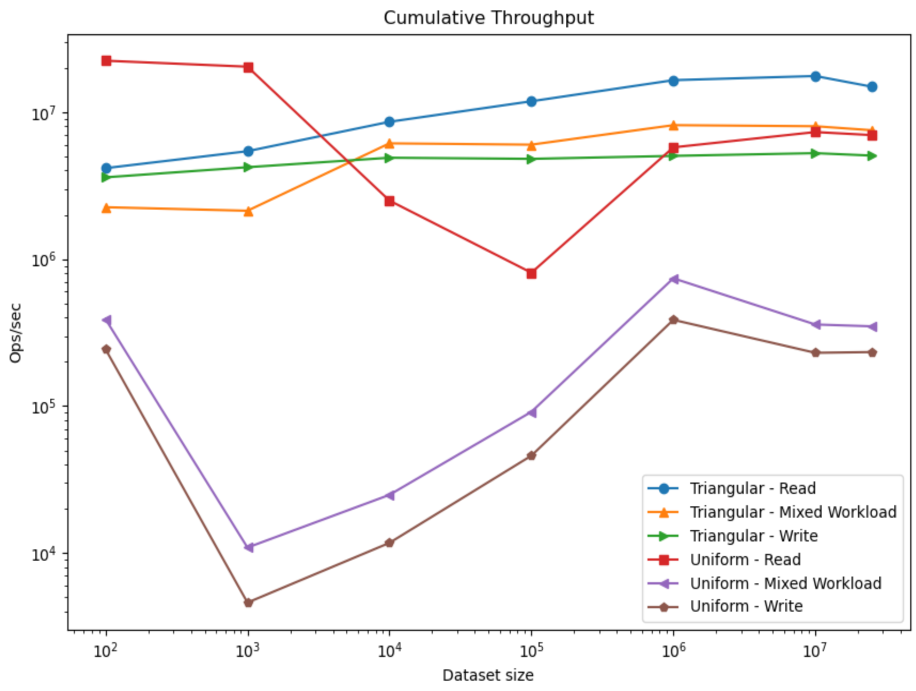
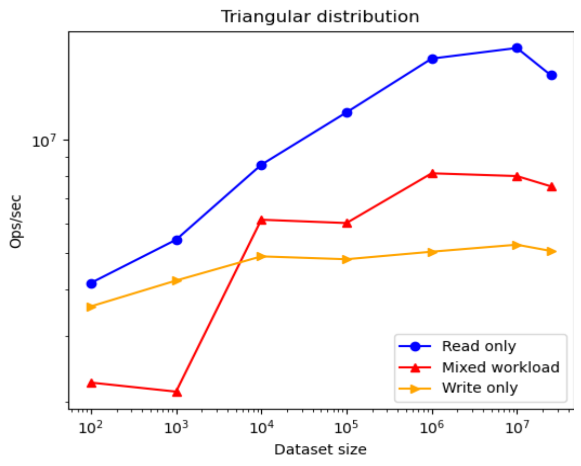
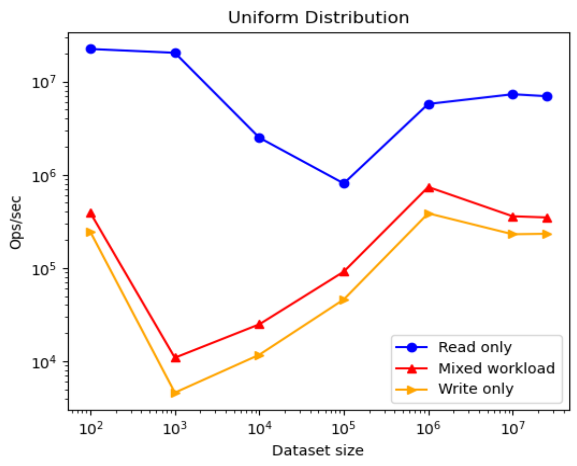


Figure 7.4: ALEX performance

Figure 7.4 shows ALEX's performance results. We are most interested in the largest dataset sizes, as they are the most relevant for real world big data scenarios, so we mostly refer to them when comparing results. ALEX performed significantly worse on the uniform dataset overall. The read performance on the biggest dataset were 7 million ops/sec on the triangular dataset and 14.9 million ops/sec on the uniform dataset, a difference of 2×. ALEX performed 233 thousand writes per second on the uniform distribution, while performing slightly over 5 million writes per second on the triangular distribution records, a difference of over 21×. The trend regarding performance for different dataset sizes seems to be that performance increases up until the size of 10^6 records, and then starts declining.



(a) Triangular Distribution



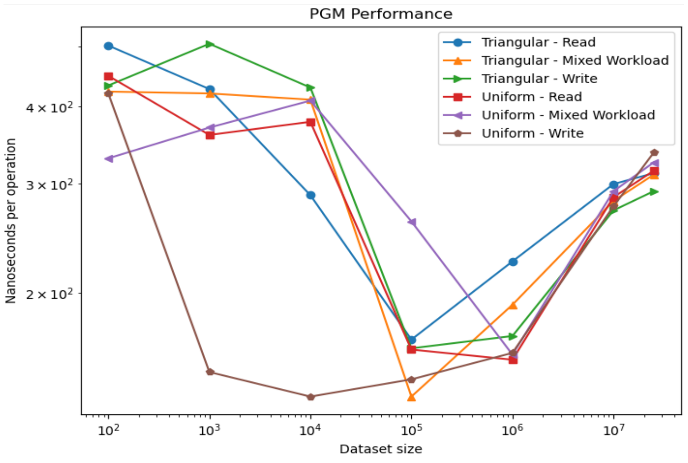
(b) Uniform Distribution

Figure 7.5: ALEX Performance - isolated dataset distributions

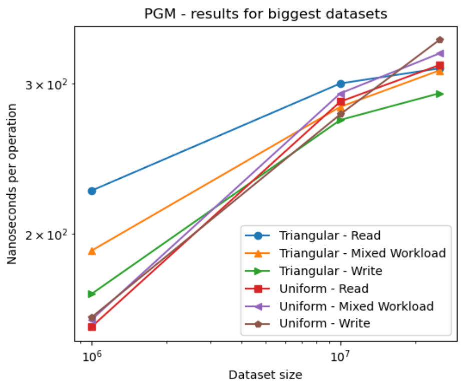
Figure 7.5 shows the results from the different data distributions in isolation. We see a trend where throughput starts decreasing from 10 to 25 million records in the triangular distribution, while being more stable at that same gap in the uniform dataset. The read-only workload on the uniform distributions is close to the results of the mixed workload of the triangle distribution at 8 million ops/sec, which is under half the amount of operations of the read-heavy workload of the triangular distribution.

7.2.2 PGM

Default PGM



(a) All datasets

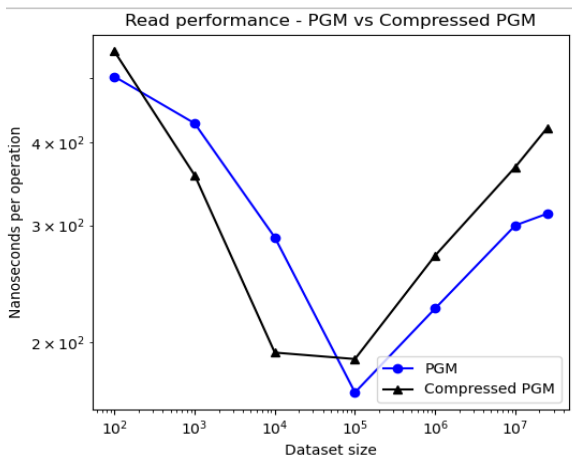


(b) Largest datasets

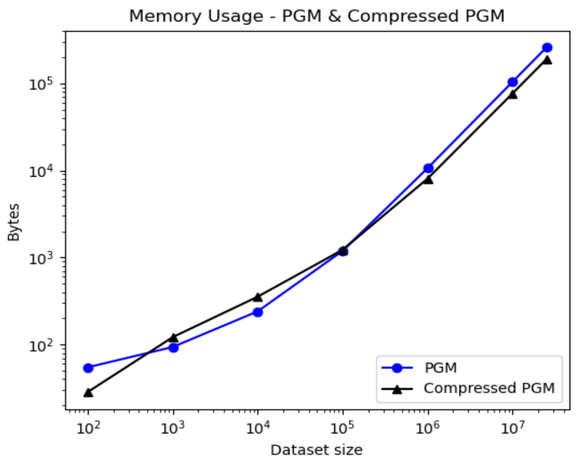
Figure 7.6: PGM Performance

Figure 7.6a shows the main performance results. There is a trend in which the performance increase up until dataset sizes of 10^5 and 10^6. For the biggest datasets, PGM are performing similarly with all the workloads. Figure 7.6b shows a magnified graph of the two biggest dataset sizes. interestingly, on the biggest dataset, PGM had the fastest accumulated query time with the mixed workload on the triangular dataset at 271.9*ns* per operation. At the biggest dataset, the difference in query time between the best and worst configuration was only 65*ns*.

Compressed PGM



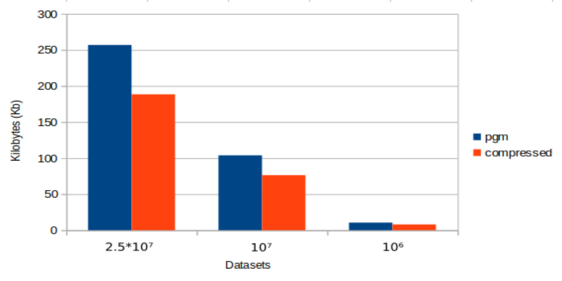
(a) Read performance



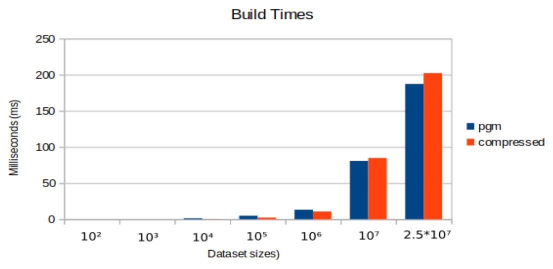
(b) Memory usage

Figure 7.7: PGM-Index Compressed PGM-Index

We tested and compared the default configuration of PGM-index and Compressed PGM-Index. Figure 7.7 shows the read performance and memory usage of the two configurations. An interesting observation is that for smaller dataset sizes, the compressed configuration is actually faster than the default configuration, as well as taking more space. From the dataset size of 10^5 and higher the compressed configuration takes up less space and performs slower than the default configuration, as we would expect.



(a) Memory usage



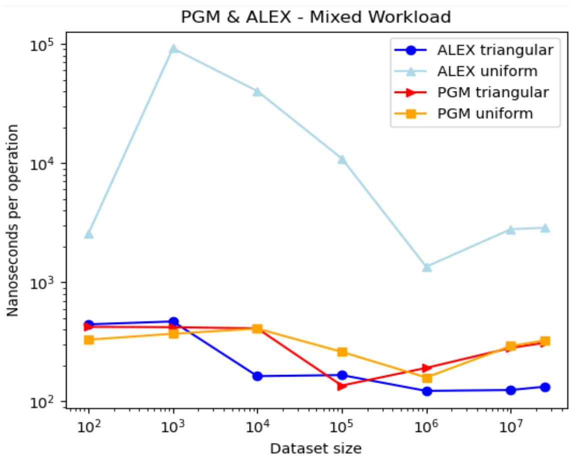
(b) Model build times

Figure 7.8

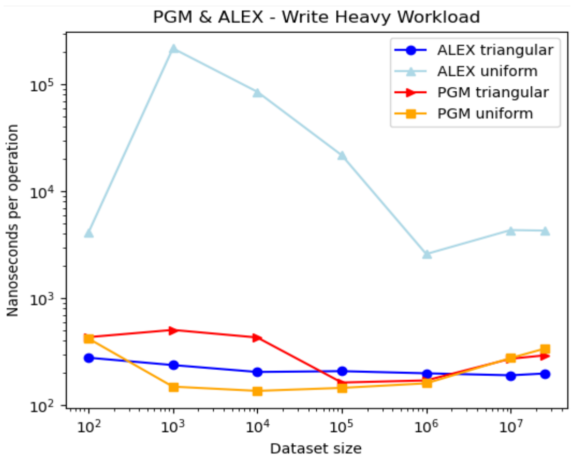
At 2.5 million records, compressed configuration had a 26.7% memory reduction, with 25.7% decrease in query time. For 10 million records, there were 26.4% memory reduction, but with only 18.3% decrease in query time. On the dataset with 1 million records, there were 24.189% memory reduction, with 16.76% decrease in speed. The trend seems to be that the compressed configuration works better for larger datasets. These tests ran on the datasets with the logarithmic data distribution. The results from the uniform distribution were similar. The results presented in [12] showed a memory saving of the compressed configuration of up to 52%, with the highest query time reduction of 24.5%. In our test with the 25 million records datasets the performance reduction were a couple of percent worse than their worst case.

There were a noticeable difference in model build times between default and compressed configuration, visualized in Figure 7.8b. The build time increases the larger the dataset becomes, with a maximum increase of 7.4% for the largest dataset.

7.3 ALEX & PGM-Index comparison



(a) Mixed Workload



(b) Write-only Workload

Figure 7.9: ALEX and PGM Performance

For the mixed and write-heavy workload, ALEX performs significantly worse than PGM-Index on the uniform distribution, but faster on the triangular distribution. The most notable trend is that PGM-Index performs very consistent between the two distributions, while ALEX is varying to a much higher degree between the data distributions. ALEX's query time on the triangular dataset is better than PGM for all three workloads.

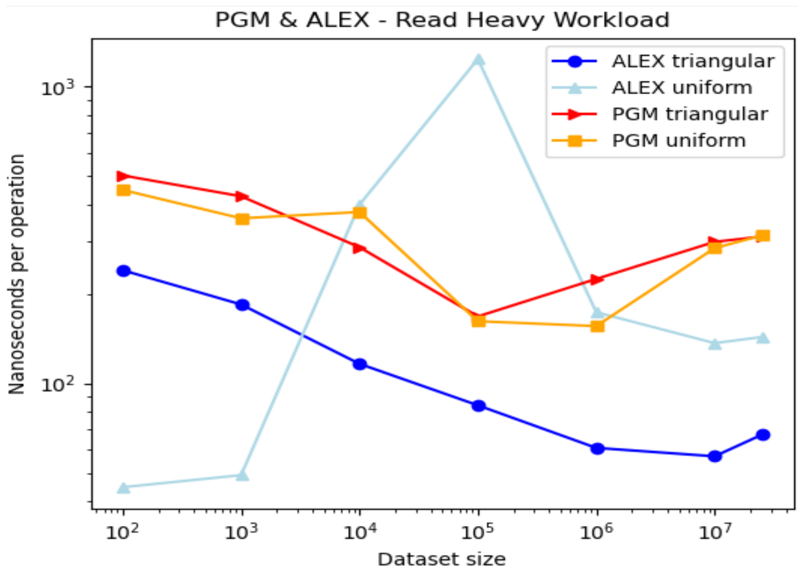


Figure 7.10: ALEX and PGM - Read-only Workload

For the read only workload, ALEX performs better than PGM on both datasets of 10 and 25 million keys. PGM-Index had consistent results for both datasets and performed slightly below and above 300*ns* for the two biggest dataset sizes. ran the indexes on slightly above and below 300 nanoseconds. The best read performance of ALEX with 67 nanoseconds per operation on the uniform distribution, $4\times$ less time than PGM.

7.4 Discussion

7.4.1 Limitations

There are some important limitations to the tests that we have conducted. The first limitation is the dataset sizes. As mentioned, we are mostly interested in results from the biggest dataset sizes, as they are the most relevant for real-world big data scenarios. The datasets used in the papers of ALEX and PGM, as well as the SOSD-benchmark have sizes of 190M keys and upwards. It would be interesting to see how if the tendencies of the results continued for even bigger datasets. The authors of ALEX states that the true power of ALEX is shown when used on really large datasets.

Another noteworthy limitation of our tests is that build times and memory usage stats are not shown when running ALEX[22]. Because of this we do not have a comparison of these statistics between ALEX and PGM in our benchmark. When comparing the two index structures, we only look at query time for the different workloads. To get the whole picture we need to also look at the space/time trade-off both of the two learned indexes. If these state of the art learned indexes, or similar structures, are to replace traditional structures like B+trees in modern database systems, these statistics have to be considered.

7.4.2 Query performance

Both PGM and ALEX is made for bigger datasets, and it is interesting to see how the results stabilize on the dataset of 10 and 25 million keys. The biggest takeaway from the read performance results is that that ALEX had the greatest read-performance overall for the most relevant dataset sizes, with a significant amount. One thing to note is that the rate at which the read performance is decreasing from dataset sizes of 10 million keys to 25 million keys is higher for ALEX than for PGM, so if this trend continues for bigger dataset sizes, the gap in read-performance would shrink. This does however not seem to be the case when comparing our results to the results from ALEX in their original paper.

In [10], ALEX had a throughput of 15 million reads/sec on a logarithmic data distribution, and around 10 million reads/sec on a uniform distribution. For our tests ALEX also had a read-performance of around 15 million reads/sec on a logarithmic distribution, while performing a bit worse with 7 million reads/sec on the uniform distribution. The trends on different data distributions are the exact same for our benchmark. The dataset sizes they used were over 7 times larger than our biggest datasets, showing that the distribution is the main factor in determining the read-performance.

The difference in performance between the different workloads for PGM-Index was very small. This is also true for ALEX on the logarithmic distribution, but not on the uniform distribution. For the write-only workloads [10] reports that ALEX had a throughput of slightly above 4 million writes/sec on the logarithmic distribution and around 3 million writes/sec on the uniform dataset. ALEX performed worse on our uniform dataset with only 233 thousand writes/sec, although had a better performance on the logarithmic dataset with 5 million writes/sec. The reason for this is unclear. It could possibly have to do with the key-values that were generated in our uniform dataset, not yielding any clear patterns for ALEX to model. However, PGM-Index had roughly the same performance on both distributions.

7.5 Conclusion

Our tests confirms that ALEX and PGM performs very similar on our system as on tests conducted by the authors of PGM and ALEX on similar data distributions. ALEX had a greater read performance than PGM-index overall on all datasets, while PGM had a better write performance on the uniform data distribution. PGM showed consistent results between the two data distributions for all workloads. It is impossible to draw a conclusion to which learned index is better based of this benchmark, given the limitations and the scale of the tests. However, we did provide a novel side by side comparison, where both structures showed very similar results to those conducted by the original authors. The versatility both indexes showed by being able to handle both reads and writes with high throughput further indicates the possibility for learned index structures to be able to replace general purpose, traditional structures in modern database systems.

Bibliography

[1] URL: https://numpy.org/.

[2] *An in-depth look at Google's first tensor processing unit (TPU) — google cloud blog*. URL: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[3] Michael A. Bender and Haodong Hu. 'An Adaptive Packed-Memory Array'. In: 32.4 (Nov. 2007), 26–es. ISSN: 0362-5915. DOI: 10.1145/1292609.1292616. URL: https://doi.org/10.1145/1292609.1292616.

[4] Alex Beutel and Google Ed Chi. *Stanford Seminar - The Case for Learned Index Structures*. Stanford. 2018. URL: https://www.youtube.com/watch?v=NaqJO7rrXy0&t=850s&ab_channel=StanfordOnline.

[5] Chiranjeeb Buragohain, Nisheeth Shrivastava and Subhash Suri. 'Space Efficient Streaming Algorithms for the Maximum Error Histogram'. In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pp. 1026–1035. DOI: 10.1109/ICDE.2007.368961.

[6] Danny Z. Chen and Haitao Wang. 'Approximating Points by a Piecewise Linear Function: I'. In: *Proceedings of the 20th International Symposium on Algorithms and Computation*. ISAAC '09. Honolulu, Hawaii: Springer-Verlag, 2009, pp. 224–233. ISBN: 9783642106309. DOI: 10.1007/978-3-642-10631-6_24. URL: https://doi.org/10.1007/978-3-642-10631-6_24.

[7] Qiuxia Chen et al. ‘Indexable PLA for Efficient Similarity Search’. In: *VLDB*. 2007.

[8] *Convex hull using divide and conquer algorithm*. Sept. 2018. URL: https://www.geeksforgeeks.org/convex-hull-using-divide-and-conquer-algorithm/.

[9] Brian F. Cooper et al. ‘Benchmarking Cloud Serving Systems with YCSB’. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[10] Jialin Ding et al. ‘ALEX: An Updatable Adaptive Learned Index’. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 969–984. ISBN: 9781450367356. DOI: 10.1145/3318464.3389711. URL: https://doi.org/10.1145/3318464.3389711.

[11] Paolo Ferragina and Giorgio Vinciguerra. ‘Learned Data Structures’. In: Apr. 2020, pp. 5–41. ISBN: 978-3-030-43883-8. DOI: 10.1007/978-3-030-43883-8_2.

[12] Paolo Ferragina and Giorgio Vinciguerra. ‘The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds’. In: *PVLDB* 13.8 (2020), pp. 1162–1175. ISSN: 2150-8097. DOI: 10.14778/3389133.3389135. URL: https://pgm.di.unipi.it.

[13] Alex Galakatos et al. ‘FIT-ting tree’. In: *Proceedings of the 2019 International Conference on Management of Data* (June 2019). DOI: 10.1145/3299869.3319860. URL: http://dx.doi.org/10.1145/3299869.3319860.

[14] Goetz Graefe and Harumi Kuno. ‘Modern B-tree techniques’. In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 1370–1373. DOI: 10.1109/ICDE.2011.5767956.

[15] Gvinciguerra. *Gvinciguerra/PGM-index: state-of-the-art learned data structure that enables fast lookup, predecessor, range searches and updates in arrays of billions*

of items using orders of magnitude less space than traditional indexes. URL: https://github.com/gvinciguerra/PGM-index.

[16] Michael Jordan and Robert Jacobs. 'Hierarchical mixtures of experts and the'. In: *Neural computation* 6 (Jan. 1994), pp. 181–.

[17] Andreas Kipf et al. 'RadixSpline: a single-pass learned index'. In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*. 2020, 5:1–5:5. DOI: 10.1145/3401071.3401659. URL: https://doi.org/10.1145/3401071.3401659.

[18] Tim Kraska et al. 'The Case for Learned Index Structures'. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 489–504. ISBN: 9781450347037. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

[19] Pengfei Li et al. 'LISA: A Learned Index Structure for Spatial Data'. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2119–2133. ISBN: 9781450367356. DOI: 10.1145/3318464.3389703. URL: https://doi.org/10.1145/3318464.3389703.

[20] Baotong Lu et al. 'APEX'. In: *Proceedings of the VLDB Endowment* 15.3 (Nov. 2021), pp. 597–610. ISSN: 2150-8097. DOI: 10.14778/3494124.3494141. URL: http://dx.doi.org/10.14778/3494124.3494141.

[21] Ryan Marcus et al. *Benchmarking Learned Indexes*. 2020. arXiv: 2006.12804 [cs.DB].

[22] Microsoft. *Microsoft/Alex: A Library for building an in-memory, Adaptive Learned index*. URL: https://github.com/microsoft/ALEX.

[23] *ML: Underfitting and overfitting*. Oct. 2021. URL: https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/.

[24] ‘Novel Online Methods for Time Series Segmentation’. In: *IEEE Transactions on Knowledge and Data Engineering* 20.12 (2008), pp. 1616–1626. DOI: 10.1109/TKDE.2008.29.

[25] Patrick E. O’Neil et al. ‘The Log-Structured Merge-Tree (LSM-Tree).’ In: *Acta Inf.* 33.4 (1996), pp. 351–385. URL: http://dblp.uni-trier.de/db/journals/acta/acta33.html#ONeilCGO96.

[26] Daisuke Okanohara and Kunihiko Sadakane. *Practical Entropy-Compressed Rank/Select Dictionary*. 2006. DOI: 10.48550/ARXIV.CS/0610001. URL: https://arxiv.org/abs/cs/0610001.

[27] Noam Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017. DOI: 10.48550/ARXIV.1701.06538. URL: https://arxiv.org/abs/1701.06538.

[28] *STX B+ Tree C++ template classes*. URL: https://panthema.net/2007/stx-btree/.

[29] *Tensorflow*. URL: https://www.tensorflow.org/.

[30] Peter Van Sandt, Yannis Chronis and Jignesh M. Patel. ‘Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?’ In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 36–53. ISBN: 9781450356435. DOI: 10.1145/3299869.3300075. URL: https://doi.org/10.1145/3299869.3300075.

[31] Giorgio Vinciguerra, Paolo Ferragina and Michele Miccinesi. ‘Superseding traditional indexes by orchestrating learning and geometry’. In: (Mar. 2019).

[32] Qing Xie et al. ‘Maximum error-bounded Piecewise Linear Representation for online stream approximation’. In: *The VLDB Journal* 23 (Dec. 2014), pp. 915–937. DOI: 10.1007/s00778-014-0355-0.