JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Anna Wirth BSc

Submitted at
Institute of Business Informatics- Data and Knowledge Engineering

Supervisor
o. Univ.-Prof. Dr. Michael Schrefl

Co-Supervisor
Simon Staudinger MSc

March 2024

Performance and Scalability of Learned Index Structures in the Wild

Master Thesis
to obtain the academic degree of
Master of Science
in the Master's Program
Business Informatics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Abstract

This thesis aims to investigate the potential of Learned Index structures using the example of different datasets. These index structures surfaced in recent years, as a credible alternative for traditional index approaches, with several papers dedicated to the purpose of evaluating the possibilities enhancing indexes with machine learning models would bring. In this thesis, the previous work will be extended with the aspect of the performance comparison in different real-world datasets, more specifically on datasets from Dynatrace.

The comparison to traditional indexes and indexes commonly found in the industry is done through experiments examining metrics like build- and lookup time as well as other, model specific, parameters. Furthermore, the scalability of Learned Index structures is evaluated. These experiments are conducted on various datasets, both real world datasets from Dynatrace and synthetic ones. The main aim of the experiments was to investigate whether a performance gain would be possible, when implementing Learned Index structures on real world datasets that can be found in "the wild", meaning the industry aside from synthetic datasets. The results of the experiments conducted in this thesis lean to the side that a performance gain is in fact possible, as that Learned indexes can offer very fast lookups. These fast lookup time are due to the fact that Learned Index structures can utilize the Distributive Data Function to effectively find the wanted lookup keys. What must be considered is that the build times of Learned Index structures tend to be higher due to their complexity, which is a fact that has to be considered when using them in industrial use cases.

1 Introduction

In this upcoming section, an overview of the structure of the thesis is given. It includes a section about the initial motivation for choosing the scope of the topic of this thesis, followed by the outline and main objective. This section will also include the main research questions that the thesis is trying to find the answer to, followed by the outcome that is expected by conducting the research and a general outlook on the structure of the thesis.

1.1 Motivation

This thesis was done in cooperation with Dynatrace [5], a tech firm that is based in Waltham, Massachusetts, United States of America, distributing a software intelligence platform also named Dynatrace. The company currently operates worldwide, with more than 4100 employees in 50 different locations, as of 2023 and a revenue of over 1 billion $ for the year preceding June 2023. The company was originally founded in 2005 in Linz, Austria and is currently listed on the New York Stock Exchange, with its initial public offering on August 1st 2019.
The core product of Dynatrace is a software intelligence platform, aiming to utilize automation and AI-powered tools to be able to offer end-to-end observability.
Nowadays, a lot of companies will provide large amounts of data that have to be considered when trying to monitor its system landscape. Monitoring is known to be a highly data-intensive application in which query latency plays a critical role in allowing customers to derive insights from the data deluge. [1] This big amount of data means losing time when trying to set up monitoring dashboards, running queries on collected data and so on. In order to manage all those amounts of data that a company might encounter while going about their day-to-day business, Dynatrace is running a data lakehouse, named Grail, for observability, security and business analytics without boundaries. Meanwhile, Grail also considers the complexities of working with cloud-native architecture and multi-cloud environments. Grail strives to deliver fast answers to all kinds of queries that a user might want to get answers for.
For Grail and similar data management applications, it is important to consider all promising options for fast query latency. One of the research areas that has been up and coming in the last couple of years, for a variety of different use cases, is the idea of enhancing or replacing index structures with Machine Learning models. [14] Papers discussing several different alternative ideas, like Piecewise Geometric Model Index [6], Recursive Model Index [11] and RadixSpline [9]. These papers showed the potential of this revisit of the well-established field of index research already. The shown potential is why considering this new branch of research would be interesting to if there are performance gains to be made, especially for the industrial use case found at Dynatrace.

1.2 Objective and Research Questions

As already briefly introduced in the previous section, the main objective of this thesis is to evaluate the idea of Learned Index structures, that recently surfaced and picked up steam in the research community to the use case of data handled at Dynatrace. It should be evaluated whether the potential in performance and memory gains seen in previous works would also work in an environment, as distributed and fast-paced, as the one found within Dynatrace datasets. More precisely, the possible gains in performance and memory footprint of Learned Index structures, discovered in previous work, are to be evaluated. In the best case, an advantage from the industry-standard algorithms should be shown, deriving a use-case for Learned Index structures within Dynatrace. We also want to evaluate the performance of this new research area compared to structures that are currently used in industry. This leads us to the following main research questions, that should be answered within the course of this thesis and that are to be discussed in the following sections:

1. *Research Question 1:* What is the potential of Learned Index structures working with Dynatrace

data? Is there any performance gain to be made in comparison with traditional index structures on the example of Dynatrace data?

2. *Research Question 2:* How do the Learned Index structures perform in comparison to traditional index structures on the example of different synthetic datasets?

These questions will be methodically investigated throughout the thesis answered in the Section 4 and the implications of the results discussed at the very end in Section 5, in order to come up with clear and precise answers.

1.3 Expected Outcome

The outcome of this thesis can be one of two scenarios, with the positive evaluation as the preferred outcome of both scenarios described in this section:

1. Positive evaluation and outcome This would happen if the evaluation of the different Learned Index approaches would yield at least one, possibly even more than one, very performant index structure. In this case, the potential for performance gains using Learned Index structures on the given Dynatrace dataset are visible and significantly fast. Also, in this outcome those structures would not only be resulting in better times for lookup and storage metrics but generally working better than traditional indexing structures and/or indexing structures that are currently available. If that were the case, concrete numbers of the lookup performance of each evaluated Learned Index structure should be available, giving a clear picture of how each index performs in more detail. This outcome will most likely result in a plan for further work coming up with a prototype.

2. Negative evaluation and outcome Having a negative outcome of all the conducted experiments would mean that results show the performance gain of the analyzed index structures, do not improve lookup/query performance or, at the worst case, are even slower than traditional indexing methods/the methods currently in place.This would also be the case if the Learned Index structures show the same performance as traditional structures. For the case of worse or the same performance, a detailed analysis of failure for each approach is going to be conducted, taking into account the parameters in play for each approach. Furthermore, strategies on how to improve on the issues discovered is going to be proposed.

1.4 Structure of Thesis

To give a clear outlook on how this thesis will be structured, a quick outlook on the different sections is given as follows:

Starting off with, Section 2 will give an overview of how a Learned Index differs from traditional index structures, giving some context of how the idea developed and which the most important pieces of literature are for the topic. Furthermore, the theoretical and mathematical basics of each chosen index structures will be briefly explained, in order to highlight the important features of each index structure as well as emphasize on the differences in how they work and operate.
In Section 3,after understanding what makes each index structure unique, the background on the choices of each Learned Index approach are briefly mentioned, as well as going about the prerequisites that had to be considered. The same section also includes the description of the data used for the experiments and which data preparation steps needed to be taken in order for the experiments to be as comparable as possible. Section 3 will also cover the different setups of experiments to get a detailed picture of each indexes' performance.
Section 4 will report on the results of the experiments described before, providing the facts and figures

necessary to come to the correct conclusion. Section 5 discusses the research questions, on basis of the results gained before. The thesis attempts to discuss the questions from as many angles as possible, to get a complete picture of the impact the evaluation has on the previously defined research questions. Finally, the most important insights and takeaways from this thesis will be presented in a summarizing manner in Section 6, also including a paragraph about possible future work on the topic.

2 Theoretical Background

This section aims to explain the idea of Learned Index structures in general as well as cover the most important theoretical concepts of Learned Index structures, by introducing each of the chosen approaches in detail. The underlying structures and mathematical basics of each index structure will be laid out, as well as mentioning the previous work in each field. While there are a wide variety of Learned Index structures that have previously been introduced, this section will only go over three main ideas: *Recursive Model Index*, *Radixspline* and *Piecewise Geometric Model Index*.

General Structure

In the following sections, the comparison of performance and functionality will be compared using two main metrics: Build time and Lookup time. Whereas build time is very straightforward, measuring the time it takes to build a certain index structure on a dataset, the lookup time is split into two main parts that will be relevant for certain parts of the coming sections. While the lookup time as a whole measures the time it takes an index structure to come up with the precise index of a certain input key, we look at two main phases of this lookup process:

- Evaluation: This is the phase where the index first gives an estimate on the wanted input key, which will be further explained in Section 2.2. Whenever the evaluation time is mentioned, the time an index needs to complete this first evaluation is referenced.
- Last mile search: In this phase, the previously given evaluation is considered and a subset of data based on this estimation is searched. At times, this will also be called the search phase. The time it takes to complete this search on the subset of data will be referenced as the search time and further investigated in Sections 2.2 and 3.

These two phases make up the total lookup time with Evaluation time + Last mile search time = Lookup time.

2.1 The Evolution of Learned Index Structures

In today's world, solving problems with Machine Learning methods has been gradually on the rise. Even fields of research, which were assumed to be thoroughly analyzed before, showed potential for Machine Learning methods to improve results that were thought to be improved no more [13]. This trend has also not stopped before the much researched field of indexing data. One of the first, most notable works revisiting the idea of indexing data with the additional focus of using Machine Learning models were . In the flagship paper with the title "The Case for Learned Index Structures" they start off with the idea that essentially, traditional index structures for sorted data, like the B-Tree can essentially be seen as models themselves. They map a key to a position within this array, acting as a model to get the job done.
Having established the fact, that there is a similarity in the way Learned Index and traditional structures work, the paper states that it should basically be possible to replace those models in the middle with any other Machine Learning model, which produce the same results. That is, taking an input key and giving back the position as an estimate. With this approximate position and considering estimation errors, we get a result interval to be searched through for the exact result, much like the last mile search in a B-Tree works. This similarity of basic concepts between a B-Tree and any Learned Index structure, as explained in the original paper, can be seen in Figure 2.1.

The authors, like Tim Kraska and others show throughout the paper, that taking advantage of the distribution of the data, comes with several benefits both in terms of lookup time and memory footprint. Within the course of this paper the authors also come up with an implementation of their Learned Index

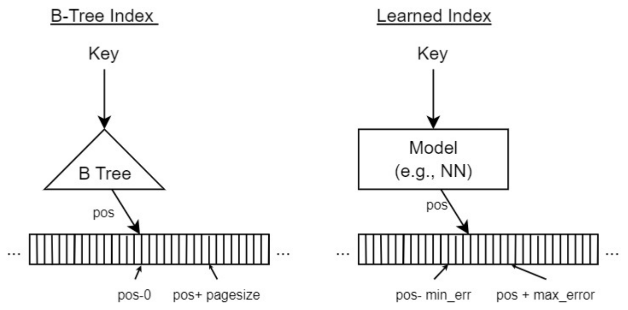


Figure 2.1: Concepts of a B-Tree vs. Learned Index structures [11]

theory, which experiments with several different Machine Learning models in a layered model, which will further be known as the "Recursive Model Index" short RMI [11]
This first implementation then prompted a response from other researchers and bloggers analyzing this new approach to an age-old topic, resulting in both critique [16], [15] and praise [6].

For a long time, one of the most important points of criticism was that there weren't any open source implementation available, resulting in researchers reimplementing the same algorithms over and over again. Furthermore, the "unfair" advantage of synthetic datasets with those newly introduced structures were criticized, as they are usually easier for the Machine Learning models to learn and therefore delivering better performance.

This would change once other authors introduced their alternatives to the RMI proposed by Tim Kraska, for example by Ferragina and Vinciguerra [6], introducing a Learned Index approach that is, unlike the RMI, built bottom up. Their ideas are publicly available on GitHub, and even integrated the Piecewise Geometric Model Index into Python, using their own library. [6]. Furthermore, the works of Marcus Kipf et al. [14] "Benchmarking Learned Indexes" aimed to solve the aforementioned problems of missing available open source code, providing a benchmark comparing publicly available open source implementations of different Learned Index approaches with each other. They work on real world datasets taking several performance parameters into account and analyzing the impact of each parameter on the performance of the Recursive Model Index (RMI), Piecewise Geometric Model Index (PGM) and RadixSpline (RS) approaches. The theoretical basics of these three Learned Index approaches will be evaluated in the next sections.

Since then, a various number of Learned Index approaches were proposed, mostly for very specific use cases, for example [3], [4], [17], [8], [18]. Most of those do not provide an open source implementation and are therefore not explained in further detail in this thesis.

2.2 Learned Index vs Traditional Indexing Approaches

Generally speaking, as briefly mentioned in the previous section, a Learned Index is an alternative approach to how data has been indexed previously. With the field of indexing being a widely researched area, a lot of thought and optimization had been invested in this topic. With Machine Learning techniques taking the world by storm in many different areas, it was only a matter of time before it reached the field of indexing as well. As diverse as the different Learned Index approaches that are going to be discussed later are, they all have the same idea in the background: to achieve the estimation of the position of data within an array of data, using an approximation of the Cumulative Distribution Function (short CDF) of the underlying data [14]. The CDF is the cumulative sum of the probabilities of a value occurring in a set of data, summing up to a total of one, meaning the probability of keys less than a

certain value x. In the context important for this thesis, we use the term CDF to mean "the function mapping keys to their relative position in an array" [14]. How an approximation of an example dataset could look like, considering the mapping to the respective position in the dataset, vs the real CDF of the data, can be seen in Figure 2.2. The approximation of the CDF can be done using all sorts of different Machine Learning techniques, technically ranging from neuronal nets to simple linear regression.
It has to be noted that Machine Learning techniques merely produce an estimation of the actual position of the input key x, which can by vary by a certain error. In the example shown in Figure 2.2 this error is seen as the distance between the green and blue line, which is biggest at input key 12 at the relative position 0.24 of the approximated position resulting of the linear regression vs the position 0.4 in the real dataset. This means that the maximum error in this function will be 0.16. Proceeding with this knowledge, we can establish that the approximated position will never be further away from the real position than the maximum error specified. [14]
This idea of a maximum error which defines the search can now be also applied to a B-Tree, one of the traditional index structures we would use on a sorted array like the one described in the previous example. We can view a B-Tree the same way, as a way of memorizing the CDF function for any data given: upon inserting every i^th key, the B-Tree can be viewed as an approximate index with the error bound set at $i-1$.

Another way to search through the sorted array would be using a binary search, starting in the middle to find the section in the array in which the key finally resides. This would be considered a simple search algorithm and not an index in its traditional sense [13].

2.2.1 Different Lookup Phases

Directly comparing traditional and learned indexes, when going back to Figure 2.1, the basic structure of Learned Index structures and traditional approaches like the B-Tree is similar. Figure 2.1 therefore shows two different phases when performing a lookup, which will be used as the two main parts that make up a lookup later on. Namely, they are Evaluation phase and the Last mile search or simply Search phase. Those phases differentiate on how they are done for the different index structures, as described in the following paragraph:

- Evaluation: when using Machine Learning models within the Learned Index structure, they produce a prediction within the data, by choosing the correct model to estimate the position. In a B-Tree this will be the traversing to the correct tree node.
- Last mile search: in the context of Learned Index structures, this is made up by the error correction. Given the position from the previous phase, the model will look in the area around the given number, in order to make up for eventual estimation errors made in the first half of the lookup. In traditional structures, this will be a simple search through the previously evaluated tree node.

2.3 Recursive Model Index

This index structure originates from the work of Kraska et al. [11], and has been analyzed, discussed and optimized by many follow-up papers since. How this index achieves the lookup of a given key, running through the two phases described in subsection 2.2.1, will be the subject of this section.

2.3.1 Purpose

The Recursive Model Index, short RMI, is a type of Learned Index structure that is able to work with multiple types of Machine Learning model types, error bounds and search strategies. All of these different

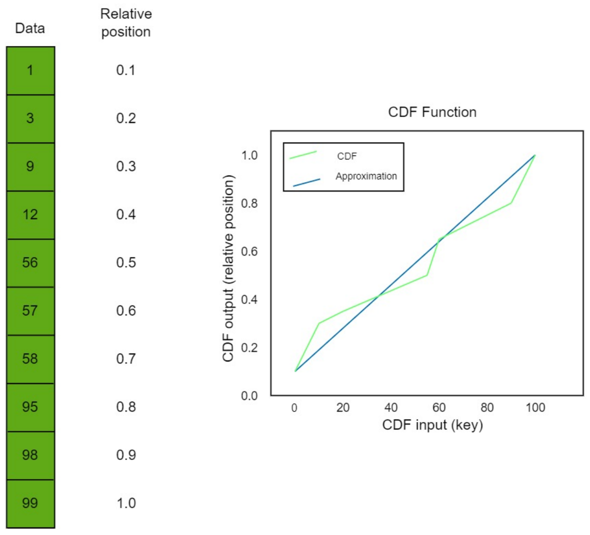


Figure 2.2: Approximation of example CDF vs. real Cumulative Distribution function [14]

parameters can be tuned, depending on the data they are representing.
Using a hierarchical structure, the RMI in its original form gets built on data that is sorted and there are no update operations done on the data. [11]

2.3.2 Internal Structure

Essentially, this index structure aims to estimate the Cumulative Distribution Function (CDF) through multi-stage models, which provide the possibility of combining different Machine Learning models. This is done through several layers of models, which are referenced by the previous layer, following a "divide and conquer" approach as seen in Figure 2.3. The general structure of this hierarchy consists of one model at the base or root of the hierarchy– this will be the root model f_0. The size of the other layers are more arbitrary.
All models making up a layer of this hierarchy will estimate a small part of the CDF. Together they will approximately map the complete CDF. [13]

2.3.3 Training

For the model $f(x)$ where x is the key to be looked for and $y \in [0, N)$ the position, the general assumption for the multi-layer model will be that there are M_l models, with l being the stage. First off, we will train the model at the root, meaning layer at stage 0, representing the entirety of the CDF: $f_0(x) \approx y$. Each model k in the following layers l, expressed by $f_l^{(k)}$ as seen in Figure 2.3 is trained with a loss which is defined in the following way:

$$L_l = \sum (f_l^{(\lfloor M_l f_{l-1}(x)/N \rfloor)}(x) - y)^2 \tag{2.1}$$

$$L_0 = \sum (f_0(x) - y)^2 \tag{2.2}$$

With $f_{l-1} = f_{l-1}^{(\lfloor M_l f_{l-2}(x)/N \rfloor)}$ which will be recursively executed. [11]

In order to build the whole model, we iteratively train each stage with the above given loss function L_l.

In order to minimize the error given by the prediction of the hierarchy models, the model is trained top-down, meaning that first the root model will be trained followed by the models in the first layer,

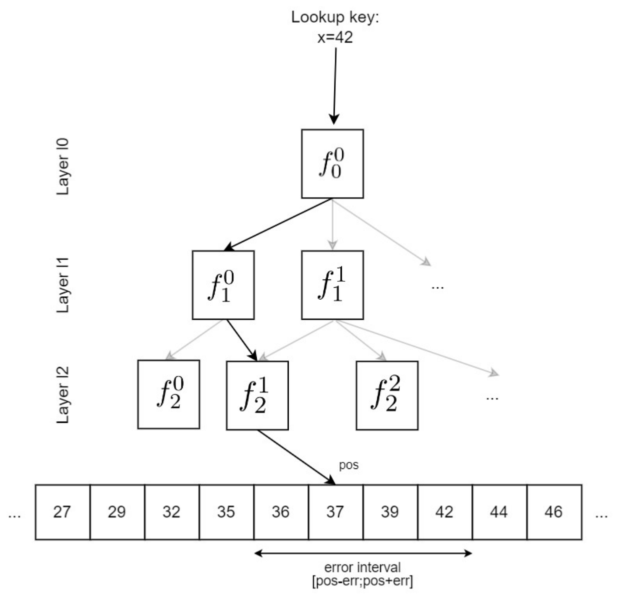


Figure 2.3: Graphical representation of the lookup for key 42 [13]

second layer, etc.

Models from a higher layer reference models in the next layer based on the input key they are passed on from the root model. In the very last layer, the estimated position of the wanted key will be calculated, giving way for the Search phase of the RMI. [11].

Two Layer RMI Technically the multistage model that the RMI is built on, could include several layers of models, though experiments showed that exceeding two layers is hardly ever required [14]. This means that, given that the root model is strictly always only one model, and referred to at the first layer will reference the models of the layer at stage one, which will be the second layer present in the model. This slimmer structure means that the second layer models are even more important to get an accurate prediction, as those are the ones pointing to the data directly. Generally the number of models B in the second layer is known as the "branching factor" of the RMI, which essentially means that the data in the second layer will be partitioned into this number of models representing the second layer, therefore making up the complete CDF. The number of models in the second layer will depend on parameters, like the desired amount of memory consumed by the index and of course the distribution of the data used. [14]

2.3.4 Lookup

As mentioned in Section 2.1 the lookup phase for Learned Index structures is divided in two main phases: Evaluation and Last mile search. The following paragraphs will explain the theoretical background on how a lookup on a Recursive Model Index works, split into the different phases of the lookup.

Prediction *Definition* Let R be a k-layer RMI on dataset D consisting of $n = |D|$ keys.

With the models trained in the fashion mentioned above, the prediction of a given key x of layer l_i is recursively defined as:

$$f_i(x) = \begin{cases} f_0^0(x) & i = 0 \\ \\ f_i^{\left\lfloor [\![|li| \cdot f_{i-1}(x)/n]\!]_0{}^{|l_i|-1} \right\rfloor}(x) & 0 \leq i \leq k \end{cases} \tag{2.3}$$

What Equation 2.3 assumes is that, intuitively, the estimate of model $f_{i-1}(x)$ of the previous layer is scaled to the size of the current layer l_i, in order to determine the correct model in the layer. As $f_{i-1}(x)$ can assume values smaller than 0 and bigger than n-1, it is necessary to introduce a restriction to stay within those boundaries. In order to end up with a valid index of the model, the results therefore need to stay with in the $[0, |l_i| - 1]$ boundaries. [13]
The estimated position for x is the output of layer l_{k-1}, which, considering that the estimate of the previous layer needs to be scaled to the current layer, is defined as the following:

$$R(x) = f_{k-1}(x) \tag{2.4}$$

Example When using these definitions on the small example data shown in Figure 2.2 this would mean that any lookup key x must be in $[0, 10)$. Assuming that there are three stages in this model resulting in a model in the first, second and third layer, the value of l will be at most 2, with a certain number of models $|l_i|$in each stage. With a similar structure to the hierarchy as seen in Figure 2.3 there is one root model which returns the approximation of the position (ranging from 0 to 9). In the second layer we will find two models and three models in the third layer at stage 2, which makes the model hierarchy in this example slimmer than the general representation seen in Figure 2.3 with a total of 6 models. Working with the approximation from the root model f_0^0, the prediction in the next layer (which is now $l = 1$) will work with the outcome and scale it to the number of keys available (in this particular case, 10) and the number of models in the current layer ($|l_1| = 2$). This scaling the result of the previous layer is recursively repeated until the last layer l_{k-1} is reached. In our example, we have a three layer model, the last layer, therefore, being layer 2. The estimate of the model from this layer, then predicts the estimated position of the real data which will then be used to perform the last mile search within given error bounds around this estimation.

Last Mile Search This phase of the lookup takes in the estimated position of x, obtained from the previous evaluation step and searches the area around this prediction in order to make up for any possible estimation errors. Keeping in mind that the array we search through is sorted, it has to be evaluated which direction from the estimated position to look for. With RMI, there are different possibilities to make this last mile search easier, with the help of stored error bounds. This could be, for example, the maximum error bound of the RMI. This would mean that the search bounds in the array would be $[R(x) - err; R(x) + err]$, which will have to be searched with the help of State-of-the-art search strategies, e.g. Binary Search. [13]. In the different implementations of the RMI, these search strategies and stored error bounds can differ, which will be showcased in more detail in later sections.

2.3.5 Hyperparameters

As mentioned in Section 2.3.3, there are quite a few possibilities how the Recursive Model Index can adapt to different conditions, making it very flexible. All these parameters will prove to be important for the experimental setup later on. The following five parameters are crucial to find the best possible performance of the index for the data: [13]

- *Model Types:* For each different layer of the model (in most cases 2) it is possible to choose different simple Machine Learning models. There has to be some thought given to the advantages a simpler model (like linear or cubic regression) might pose, as opposed to more complex structures (neural networks etc.).
- *Layer Count:* While most of the time only two layers are used, the possibility to create deeper RMIs still exists, with those models, while being able to distribute the keys more evenly, they are larger and possibly slower to evaluate.
- *Size of Layer:* The amount of models in the lower layers can also be set, depending on the requirements. Larger layer sizes come with more models, which might be able to make predictions more accurately, as the segments that each model is responsible for are smaller.
- *Error bounds:* Storing different kinds of error bound can significantly help to make the searches faster, as the intervals to be searched in the end become smaller.
- *Last mile search algorithm:* How the interval given by the estimation is searched can also be varied, depending on the previously chosen error bounds. Examples for this would be: Binary Search, Linear Search, Exponential Search.

2.4 RadixSpline

This section will cover the theoretical basis of the RadixSpline Index, a Learned Index structure which approximates the CDF using a linear spline. Additionally, those spline points are saved in a radix table that maps keys to the smallest spline point sharing the same prefix. [13] This combination of concepts inspired the compound name RadixSpline. This Learned Index structure was first introduced by Andreas Kipf, Ryan Marcus and Tim Kraska, among others, in 2020.

2.4.1 Purpose

The RadixSpline is configured to be an index that does not support updates or inserts, as it is made for an application where this is not necessary ("Write once-read many" use cases). Furthermore, the focus lies on delivering an index structure that can be built efficiently, meaning that it does not need several passes over the data to be able to build the structure. RadixSpline needs only one single pass over the data for building. Also, RadixSpline needs the underlying values must be ordered by the time the index gets built. [9].

In order to get to the finished index, there are two main phases that the data needs to run through, which can be done simultaneously for the sake of quicker building times:

1. Build the spline: In this step, considering certain error bounds, a linear spline is fit to the CDF.
2. Build the Radix Table: Taking into account the previously created spline points of the linear spline, those points are mapped in a radix table, as an intermediate index to those points.

2.4.2 Internal Structure

The index consists of two main components. First, the *Spline Points* and secondly the *Radix Table* as seen in Figure 2.4. The spline points are a subset of the indexed keys. They are distributed in a way that the interpolation between one spline point to the next, satisfies a certain error bound, that is set, as can be seen in Figure 2.5.

In order to keep track of all the spline points that approximate the whole CDF, a radix table is created, holding each spline point's position in order to quickly locate the according spline point for a lookup key. The radix table limits the range of spline points to look through by every possible b-length prefix of a

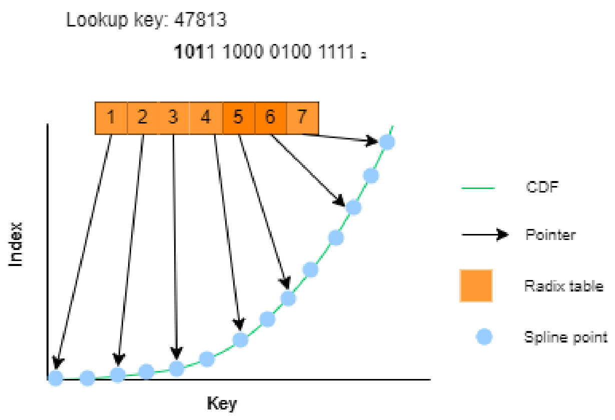


Figure 2.4: Connection between the Keys indexed in the Radix Table and the spline points, seen within an example lookup process. [9]

lookup key. This table is a flat array that maps the fix-length prefixes, called "radix bits", to the first spline point with that same prefix. The key prefixes are the offset into the radix table, where the spline points are represented as unsigned integer values. [9]

2.4.3 Training

Spline Let $D_i = (k_i, p_i)$ where D is the dataset, D_i represents i^{th} data point, k_i the key of the ith data point and p_i stands for the position (offset) of the ith data point. First, we build a spline model S, such that $S(k_i) = p_i \pm \epsilon$ where ϵ is a specified constant value. The spline points in this model can be seen as a "representative set of data points" and for any given lookup key the distance between the two closest points in $Knots(S)$ will not produce an estimate with an error bigger than ϵ. To evaluate $S(x)$ we end up with the following equation:

$$S(x) = p_{left} + (x - k_{left}) \times \frac{p_{right} - p_{left}}{k_{right} - k_{left}} \tag{2.5}$$

with (k_{left}, p_{left}) the knot with the greatest key, such that $k_{left} \leq x$ and letting (k_{right}, p_{right}) be the knot with the smallest key, such that $k_{right} > x$. [9] In the example used before from Figure 2.2 the value for k_{left} would be 99 with p_{left} being 9, and the minimum key and position k_{right} and p_{right} at 1 and 0.

Radix Table To build the table, an input parameter, specifying how many radix bits r are indexed, therefore also setting the size of the table at 2^r. After the correct size for the table has been allocated, we iterate over all the spline points, adding every new r-bit prefix b that arises, inserting the offset of the spline point into the respective slot at the offset b in the table. This is done from left to right, as the spline points themselves are already ordered. For optimization, common prefix bits that are shared by all keys are tossed.

2.4.4 Lookup

Now, when looking for a key, the built structures work together to find the position of the lookup key in the data. A sample can be seen in 2.4. In that case the lookup key x has the value of 47183, which is represented by the binary of 1011 1000 0100 1111. Depending on the size of the radix bits r, we extract a prefix b from the lookup key's binary representation. In this case, the value of r was set to 3, which translates to 101 in this example case. Using these extracted bits, an offset access into the radix table is made on the position of b and $b + 1$. As 101 would translate to 5, we would retrieve the pointers of the table stored at the position 5 and 6. This now results in the tight search bound for the wanted lookup key, as each number with the same prefix is stored in this exact segment of the data. Next, this search

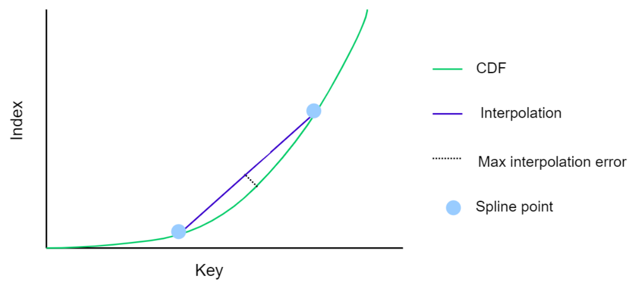


Figure 2.5: Close up of a spline segment maximum segment-the between the spline points making up a segment can only get up to a certain error bound. [9]

range, more specifically the spline segments between the indexed spline points of the radix table, are searched thoroughly in order to find the exact position. In order to get an estimated position p for the key in the search segment, the next step would be to perform linear interpolation between those spline points, the principle to be seen in 2.5. After estimating the value of p, a search within the search bounds ϵ, which depends on the preset value for this variable, is conducted, meaning the total search interval for the input key x will be $[p - \epsilon, p + \epsilon]$,. for these bounds, a Binary Search algorithm is used to find the first occurrence of the key in question. [9]

2.4.5 Hyperparameters

In order to properly train/fit this index to the data used, there are two main Hyperparameters that can be chosen upon building the model. These will influence the way the spline points are distributed in the first place, as well as have an effect on the way the prefix for the radix table is interpreted. Both are important to consider sufficiently when discussing the size and latency of the created index. These parameters are the following: [9]

- Spline error: This parameter determines the size of the maximum error bound, to be found between two spline points. This is especially important when first fitting the spline to the data. A larger error bound means the gap between the "knots" or spline points of a spline are further apart, meaning there are less spline points in overall, resulting in a smaller size model in total. Whereas a smaller error bound means the spline points are closer together, resulting in a bigger number of spline segments to be indexed and therefore a bigger overall size, with likely increased speed due to the fact that the last mile search is done on a smaller segment. [9] Instead of the spline error, one could also set the number of spline points to be constructed. In the following experiments, I stuck to differently sized spline errors, therefore this second possible hyperparameter will not be investigated further in this thesis.
- r number of bits: With this parameter, the length of the prefix for the indexed spline points is fixed. Here, a bigger value gives the opportunity of distinguishing the different spline segments more accurately, therefore inserting more prefix combinations in the radix table. Therefore, a bigger size of prefix bits increases the index size as opposed to smaller sizes, meaning less variation in the prefixed spline point values.

2.5 Piecewise Geometric Model Index

Another approach to using the idea of Learned Index structures was introduced in 2020 by Paolo Ferragina and Giorgio Vinciguerra: "PGM Index: a fully-dynamic compressed Learned Index structure with provable worst-case bounds". In this section, the basics of the Piecewise Geometric Model Index, a Learned Index structure approximating a data's CDF by piecewise linear approximation, are covered. [6]. The internal structure, as well as the process how a lookup is performed, are included in the next sections.

2.5.1 Purpose

The Piecewise Geometric Model Index, short PGM, is another example of a Learned Index structure, that attempts to efficiently index data by approximating the CDF. One of the main features of this type of index is its dynamic choice of segments in the Piecewise Linear Approximation Model, as the model is optimized to the number of segments of underlying data. This data is to be sorted, as in the previous approaches discussed.

This aims to speed up pointwise queries, building upon previous work done regarding
the RMI [11], and another Learned Index structure, similar to the PGM Index,
the FITing Tree, [7] a structure which will not be elaborated on further in this work.

Another important characteristic of the PGM model is its multi-level structure, with each level holding a set of linear regression model, each fitted on a small subset of data, depending on a set error bound. [14]. The whole structure is built bottom up, meaning the creation of each model starts from the data upwards, unlike the way it is done with the RMI. [6]. When building this index structure, a suitable error bound is chosen as a first step, giving instruction to the model how big each of the data segments, upon which a Piecewise Linear Model is build, should be. This is repeated until the number of resulting models is smaller than a certain threshold. [14]. This will produce a structure which will look very similar to the example shown in 2.6

This indexing approach developed to also be able to handle inserts into the data, as well as handle a certain workload of a query.

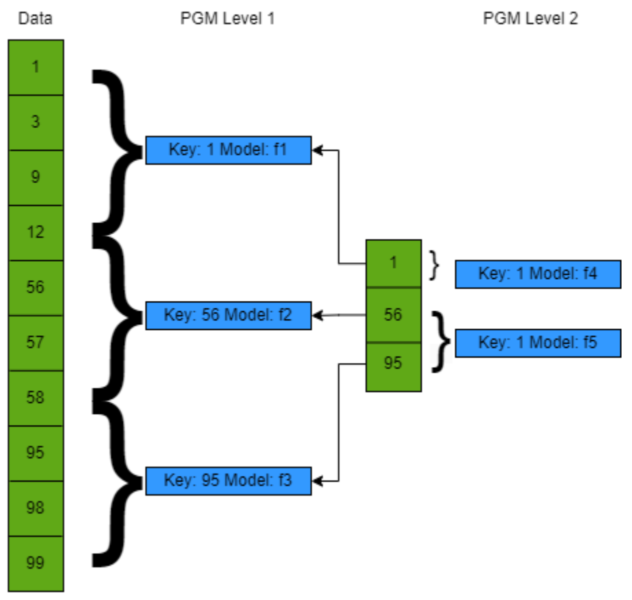


Figure 2.6: Exemplary structure of a Piecewise Geometric Model Index [14]

2.5.2 Internal Structure

Let A be a sorted array storing keys of the multiset S of the Universe U, with S possibly containing duplicate keys. Within this structure, we aim to learn a mapping that returns the estimated position for the key $k \in U$ in the array A, which can at most be ϵ away from the correct position in A. [6] The most important building blocks of a PGM Index are the different Piecewise Linear Regression Models built upon the data. On each level, depending on the shape of the data distribution, a number of models are constructed. These linear models are referred to as segments, which are themselves are represented by the triple $(key, slope, intercept)$. Such a segment indexes a range of U through the function $f_s(k) = k \times slope + intercept$. *Definition 1*: Let A be a sorted array of n keys from a Universe, U and let $\epsilon \geq 1$ be an integer. A segment $s = (key, slope, intercept)$ is said to provide an $\epsilon - approximate$ indexing of the range of all keys in $[k_i, k_{i+r}]$, for some $k_i, k_{i+r} \in A$ if $|f_s(x) - rank(x)| \leq \epsilon$ for all $x \in U$ such that $k_i \leq x \leq k_{i+r}$. [6]

When we once again consider the example of Figure 2.2, A would be the data given there ranging from 1 to 99 with $n = 10$ keys, without duplicates. The Universe U being integer values. An example of key k would be 12, which is found at position 4. In the example seen in Figure 2.6, the three segments are constructed in a way that every key in that segment is at most ϵ positions away from the initial estimation, which we can see from the relationship of the true rank of a lookup key x, for example 12, with the estimation from the segment model. The difference between the estimation of the position of 12 and the rank, which is 4 in the example case, can be at most ϵ values apart. The models are therefore constructed that this relationship holds true for every model in each layer.

Given the above definition, we see that a segment can be seen as a predecessor search structure for the data range that it covers. Covering the entirety of all keys in A might not be enough to guarantee the $\epsilon - approximation$ of every key. Therefore, it is suggested to provide a sequence of segments to cover every key in A.

Definition 2: Given $\epsilon \geq 1$ the piecewise linear *ϵ-approximation problem* consists of computing the PLA model which minimizes the number of segments $\{s_0, ..., s_{m-1}\}$, provided that each segment s_j is ϵ-approximate for its covered range of keys in S. These ranges are disjoint and together cover the entire

Universe U.

Which, translated to the above-mentioned example would mean that the models in one layer, spanning on the data from 1 to 99 is constructed in a way that the amount of models covering the whole data range is not larger than it has to be. In our case, we would have 3 segments covering the entirety of $A\{1, 3, 9, 12, 56, 57, 58, 95, 98, 99\}$.

2.5.3 Training

The next step will be to find the optimal PLA model for the array of data A, with the solution heavily relying on similar research that has been done for lossy compression and similarity search in time series. [6]. For the context of the PGM index, the answer to the number of keys in a segment, based on previous work, lies at 2ϵ meaning any segment of the optimal PLA-model covers at least 2ϵ keys.

Now that the optimal amount of models has been found, the next step would be to index the resulting segments. The output of the first steps being a sequence $M = [s_0, ..., s_{m-1}]$, with the ϵ-approximate segments, with the rightmost segment such that $s_j.key \leq k$. In order to achieve fast lookup throughout those segments of data, taking into consideration the advantage gained from knowing the data distribution, the next step recursively repeats the steps from the first layer. The keys needed for this operation will be derived from the segments that were previously constructed. Starting with the sequence M constructed over the whole array A of input keys, we extract each key of A covered of any segment in M. This step is to be repeated until the number of models in a layer is small enough, typically consisting of only one segment at the very top. [6]

For the above example, this translates to:

$$M = \{(1, slope_1, intercept_1), (56, slope_2, intercept_2), (95, slope_3, intercept_3)\}$$

therefore extracting each key which results in a new dataset consisting of $1, 56, 95$ as can be seen in Figure 2.6.

2.5.4 Lookup

Given the now-finished structure and trained model on the given dataset, running a lookup on the PGM Index would work as follows: Starting from the smallest layer on the top we use the previously built segments to find an approximation of the wanted key k on the following layer. With this estimate, the real position in the following layer is obtained, running a Binary Search in the range of 2ϵ around the estimated position. This repeated until the models do not further point to segments but to the actual data. Using the estimation of the last layer of models, the prediction is once again used as the middle ground to form the search interval, in the matter previously encountered. [6].

2.5.5 Hyperparameters

The PGM Index is by default auto-tuned, and does not need any previous hyperparameter tuning. [6] The only way that the model can be influenced is by choosing a different value of the error term ϵ, which impacts how many segments the optimized PLA creates and therefore having an effect on the model size and lookup time.

3 Experimental Setup

In this section, the data preparation that was done on the raw data is described, along with the prerequisites that result from harmonizing the usage of every open source implementation as well as from the indexes theoretical boundaries. Furthermore, the different datasets used to perform the experiments on are described along with their data distribution. The following subsections also include the description of the framework that was used to conduct the experiments and make them comparable to previous work, as well as each of the experiments and their setup in more detail.

3.1 Prerequisites and Data Preparation

For running the experiments, a couple of prerequisites had to be considered in order to satisfy the assumptions made by each of the Learned Index structures, which were briefly mentioned in Section 2.

- *Numeric values:* The open source implementations of the chosen Learned Index structures are currently only implemented for values that contain numeric values only, but not for strings, characters etc. Therefore, it has to be ensured that the data only includes values that are numeric.
- *Certain data format:* Additionally to the data being numeric only, the numbers have to be unsigned 64 or 32-bit integers, as the algorithms to build the index and to run lookups on it are currently only implemented to work with those two data formats.
- *Sorted values:* As previous research in the field of learned indices has pointed out, the numeric values in each dataset have to be in a sorted, ascending order. This will be critical to approximate the distribution of the data efficiently, and to build the index the way it was initially intended.
- *Read only data:* With the exception of the PGM Index, no index structure does yet support inserting/updating operations upon the data it gets built on. At this point in time, any operation that attempts to change the already indexed data, results in a re-build of the entire index structure. In order to make the experiments comparable, we will refrain from using the variation of the PGM index that actually supports inserting operations. This means that the data we work with has to be read only at the time that the index gets built.
- *Size of keys included:* Another minor detail for the harmonization of the build and lookup of each open source implementation, the size of the lookup keys contained in the dataset, is set as the very first value in the array of numbers e.g. 1 million for the smaller datasets and about 50 million for the bigger dataset.

Derived from the requirements on the data mentioned above, I conducted several data preparation steps before the datasets were ready to be used for the experiments. Foremost, the datasets, which came in several different input shapes and forms, had to be harmonized in several steps: It had to be ensured that only numeric columns are contained in each of the datasets, Especially for the Dynatrace datasets, I omitted some other columns, as they were in a non-numeric format, which is not relevant for the following experiments. Next up, the required data format of unsigned integers had to be achieved, therefore transforming the given numbers into the correct format of unsigned integers. As a next step, I checked that the data was sorted in ascending order, which simultaneously made sure that no missing numbers would be included in the resulting datasets.
Lastly, I added the size of keys to the dataset as the very first element in the array.

3.2 Datasets

For all experiments described in one of the upcoming sections, different datasets were used. Those datasets can be roughly grouped into 3 different groups:

1. Datasets which were already part of previous benchmarking work
2. Dynatrace Datasets
3. Synthetic Datasets

The datasets in more detail along with the reasons why exactly those datasets were chosen will be explained in the next section.

3.2.1 Benchmarking Datasets

In this part I used one of the datasets presented in the "Search on Sorted Data Benchmark" paper [14] namely the "books" dataset. It was included to get a feeling for the different index structures and at the same time, to sanity check the results, with the ones already given from the original paper [14]

The "books" dataset consists of 200 million unsigned 64-bit integer keys, where each key uniquely identifies a particular book, while also adding a payload to the keys, consisting of the popularity of said book. As the results for this dataset were already studied in previous work, see [13] and [14], the explanation of setup and results will be omitted in the further sections of this thesis.

3.2.2 Dynatrace Datasets

The datasets used here are the centerpiece of this thesis. They are used to exemplary judge the performance of each of the Learned Index approaches upon Dynatrace data. For this purpose, there are 2 different sets of data considered: a smaller one consisting of about 1 million records and an aggregation of several smaller ones into a bigger dataset, containing about 50 million data points. The datasets were altered to only contain the numeric columns of the previous datasets, as the index can only be generated on columns that hold numeric values. The distribution of the keys which are going to be relevant can be seen in Figure 3.1 and the keys that occur the most can be found in Table 3.1.

As the range of keys for both datasets are relatively big, a chunk plot over the dataset meaning groups of consecutive values had to be built, with the chunk size for the smaller dataset being set to 200 and with the bigger dataset set to 1000. Therefore, the gray area of the covers all counts that occur within this group of 200 (or 1000) values. This means, that if a certain chunk contains many duplicates, the collective count for this chunk is going to cause a visible spike. To additionally showcase this behavior, the mean count of each group or chunk was plotted as the blue line. This gives a better picture of how large the amount of duplicates for each chunk really is. Additionally, both graphs had their y-axis adjusted to show the logarithmic value of their respective counts. This was done to better showcase the difference between the values in one dataset, which would otherwise be too great to see properly.
In addition to the chunk plots, the CDF for both datasets were also constructed and confirm the findings of large amounts of duplicates in certain areas, as seen in Figure 4.8, which lists the keys which occur the most in the dataset, stating the amount of time one key can be found in the dataset in the "Count" column,

If we now analyze the two graphs separately, there are a couple differences to be spotted along with overall trends. Starting with the smaller dataset, it is visible that there are roughly three areas which contain a lot of duplicates, which Table 3.1 shows. Therefore, the area around 2800 contains several values with lots of duplicates with the most found at 2857 with over 83 000 duplicates. The other main areas showing a great amount of duplicates are single numbers like 4 and 12 641, which have a count of 67 728 and 69 232, respectively.
On the bigger dataset, there are significantly more groups that have many duplicate values. With the number of duplicates spiking in the lower values areas, there are comparably less duplicate values in

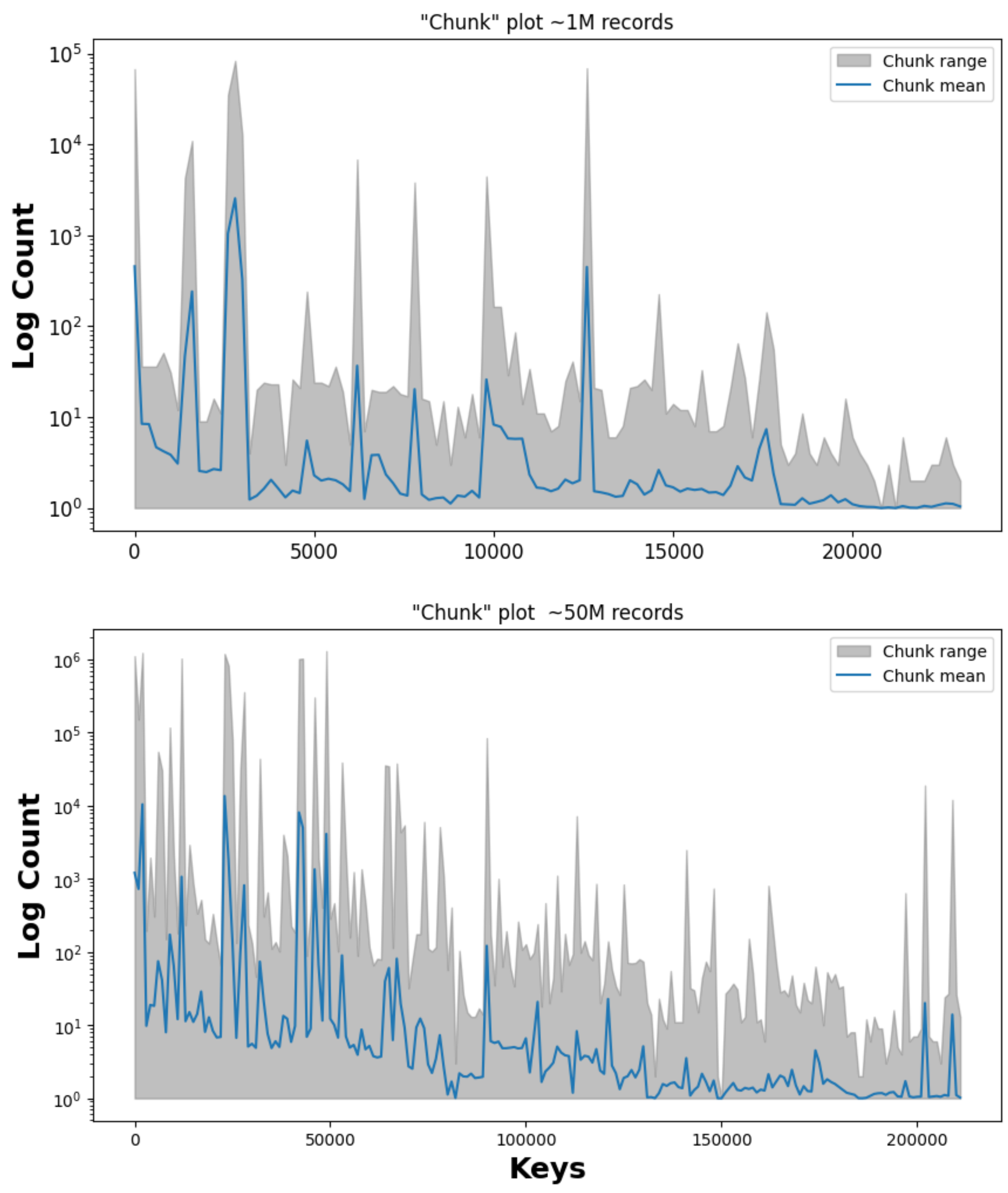


Figure 3.1: Chunk-plot of both Dynatrace datasets, showing the distribution of each dataset

Key	Count
2857	83518
12641	69232
4	67728
2868	65137
2973	56900

Table 3.1: Top 5 most common values in the smaller dataset

Key	Count
49542	1301164
2815	1227840
23149	1190134
3	1109698
49520	1040893

Table 3.2: Top 5 most common values in the bigger dataset

the second half of the dataset. This observation can be confirmed when looking at the 5 values with the most duplicates in Table 3.2. All of those values are smaller than 50 000, which falls into the first quarter of the data, considering the largest key to be more than 200 000. For this dataset the values which occurs most often is 49 542 with 1 301 164 counts of this particular value. The other number occurring in this table can also be spotted when looking at Figure 3.1, with value 3 occurring 1 109 698 times and value 2815 1 227 840 times. As the graph also spikes in these areas, they are once again validated.

What can be said in both cases is that the data that the experiments get run on, is that they are really skewed. This is because they contain many duplicates in different areas of the dataset, though due to the size of the dataset the amount of extreme duplicates varies a lot between those two datasets. When looking at the CDF of each dataset, this picture of extreme skew is supported by Figure 3.2.

3.2.3 Synthetic Datasets

As a third group of datasets, a couple of different synthetic datasets were created and included in the experiments. These also build some sort of control group, for the effort to confirm the trends that can be seen in experiments with the Dynatrace datasets. All of those datasets were created to contain 1 million data points. It is important to mention at this point once more, that these datasets, that follow some sort of sophisticated data distribution are very more likely to push the performance of Machine Learning models. Therefore, the results for these datasets are solely used as a means to argue why Learned Index structures might perform better in some scenarios. Generally, as some of the datasets described below are designed for continuous variables and the Learned Index structures can only work with integer values, the values for these datasets were rounded to the next whole number. For these purposes five different synthetic sets of data, which follow a certain data distributions, are included in the experiments:

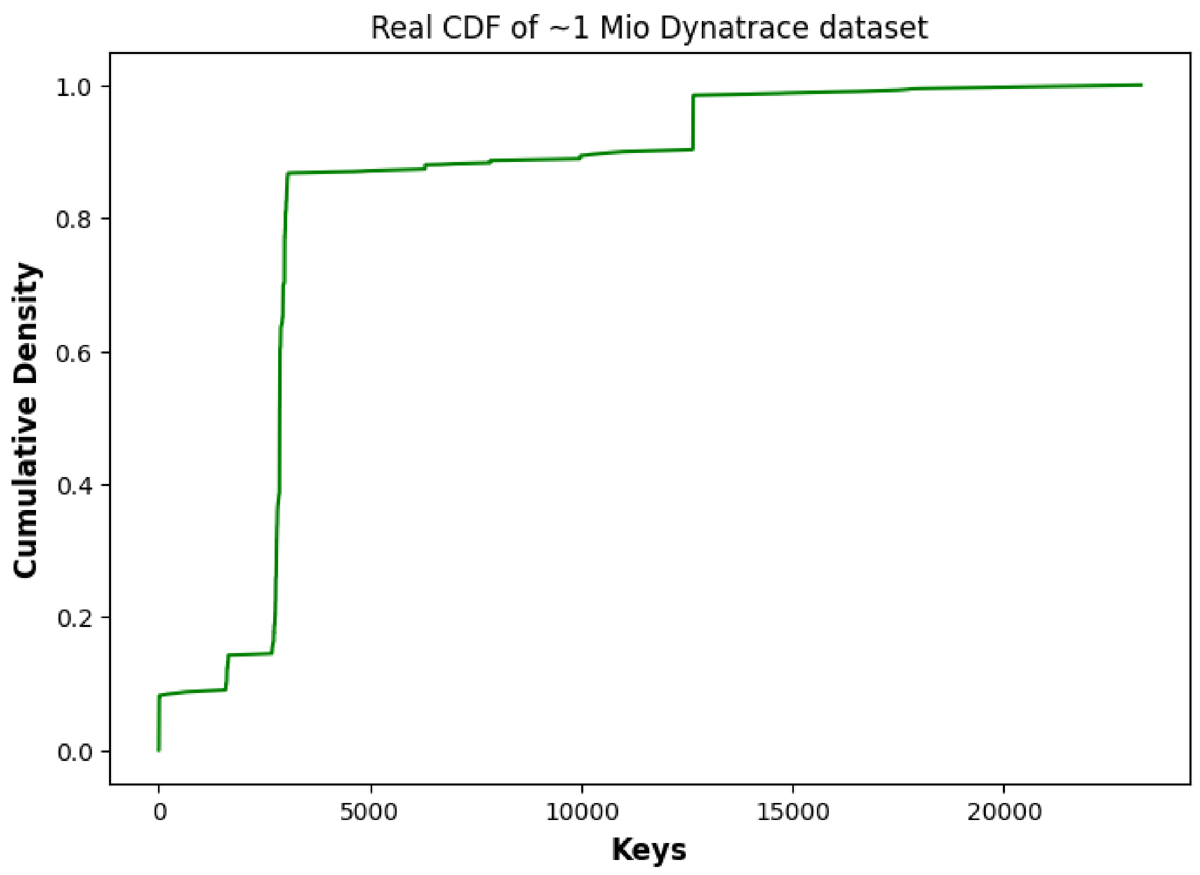
- Zipf distributed data

  The dataset used here follow the Zipf Distribution. It is a discrete distribution often used to model a variety of different phenomenons, named after the German linguist George Kingsley Zipf. This distribution basically states that the size of its i-th occurrence is inversely proportional to the rank it has in the distribution. [19] Using the standard Python function `numpy.random.zipf`, with a distribution parameter of 5, in order to make the distribution less extreme and easier to plot. This produces a dataset with data distribution that can be seen in Figure 3.3.

  The majority of keys in this sample size of 1 000 000 have the value of 1 with rapidly decreasing probability for any of the following keys, as can be seen in Table 3.3.

- Binomial distributed data

  This next dataset follows a binomial distribution, one of the most common data distribution, developed by Carl Friedrich Gauss. It is characterized by the mean and the standard distribution, influencing the form of the distinctive bell shaped curve. In our example, and in order to be somewhat



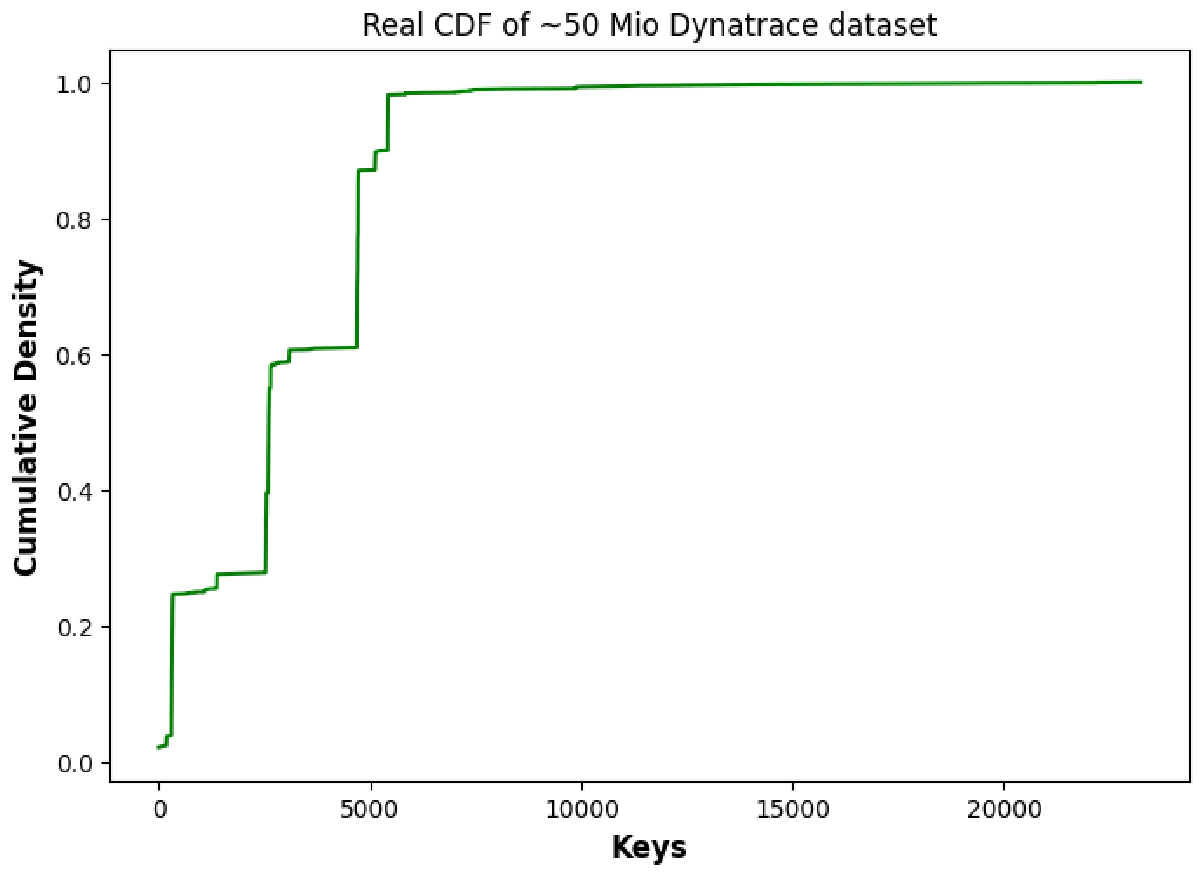


Figure 3.2: Actual CDF of both Dynatrace datasets

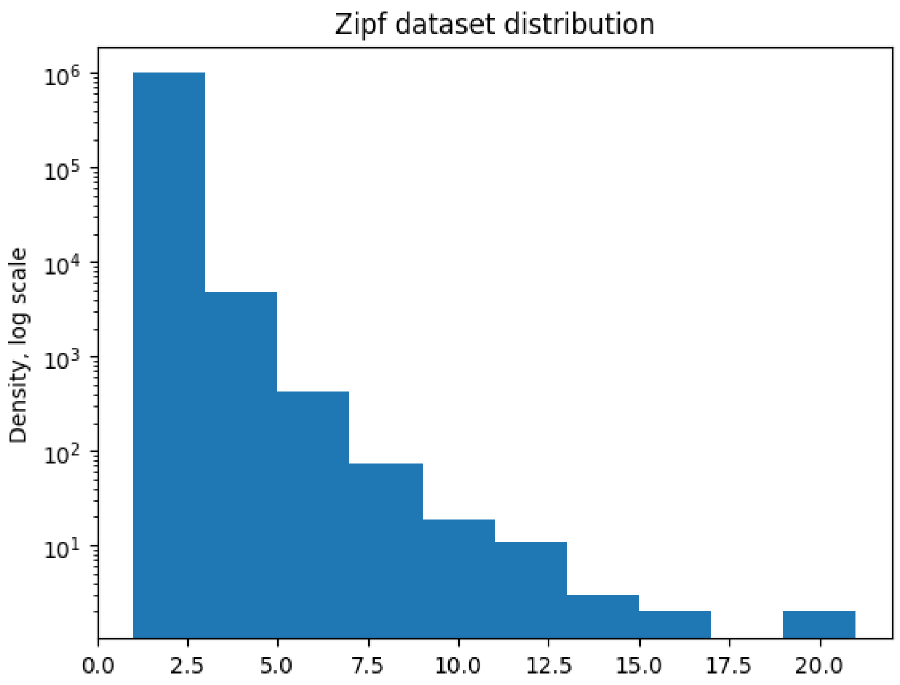


Figure 3.3: Zipf distributed dataset with a=5

Key	**Values**
1	964797
2	29816
3	3942
4	911
5	279
6	143
7	59
8	16
9	14
10	5
11	4
12	7
14	3
15	1
16	1
21	2

Table 3.3: Table of exact data distribution of keys present in sample Zipf distribution

similar to the values range that the Dynatrace datasets cover, we set the mean to be at 25000 and the standard deviation to 111.75. Using these parameters with the `numpy.random.binomial` function, a sample size of 1000 000 and with a p of 0.5 would produce the output as to be seen in Figure 3.4.
A fair share of the keys are grouped around the mean value of 25000, the detail of the distribution can be taken from Table 3.4.

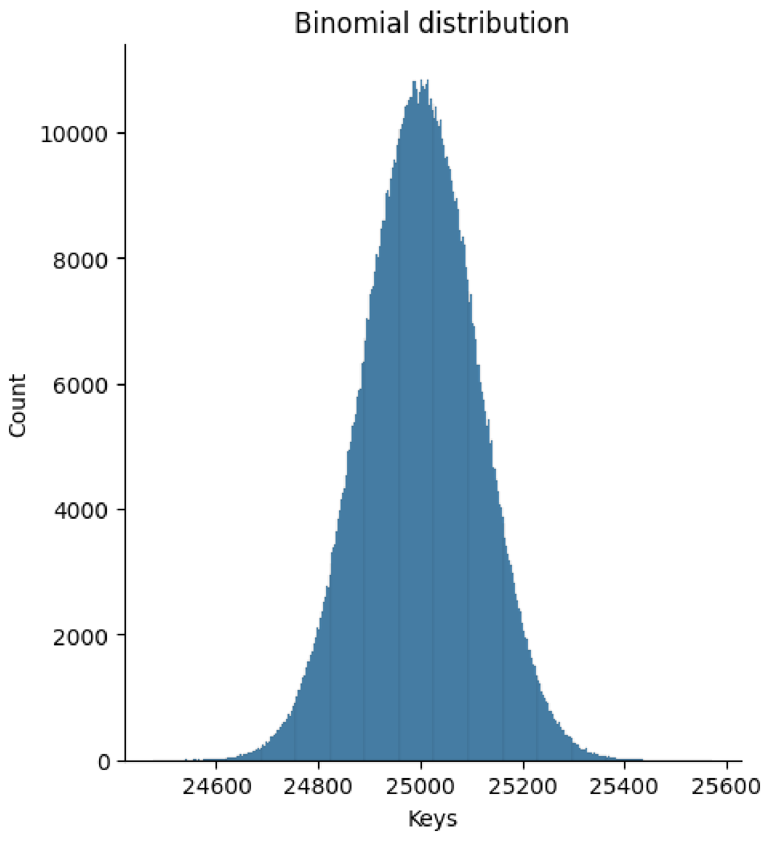


Figure 3.4: Distribution graph of binomial distributed values

count	1000000
mean	25000.02
std	111.75
min	24441
25%	24925
50%	25000
75%	25075
max	25599

Table 3.4: Summary of binomial distributed data sample

- Uniformly distributed dataset

  This dataset follows the uniform distribution, where all outcomes or values in the specified interval are equally likely to occur [12]. In order to get a sample of integers of the required size of 1000

000 using `numpy.random.randint` an interval had to be specified, along with the sample size. In this case the interval size ranges from 1 (inclusively) to 50000 (exclusively). The resulting distribution can be seen in Figure 3.5

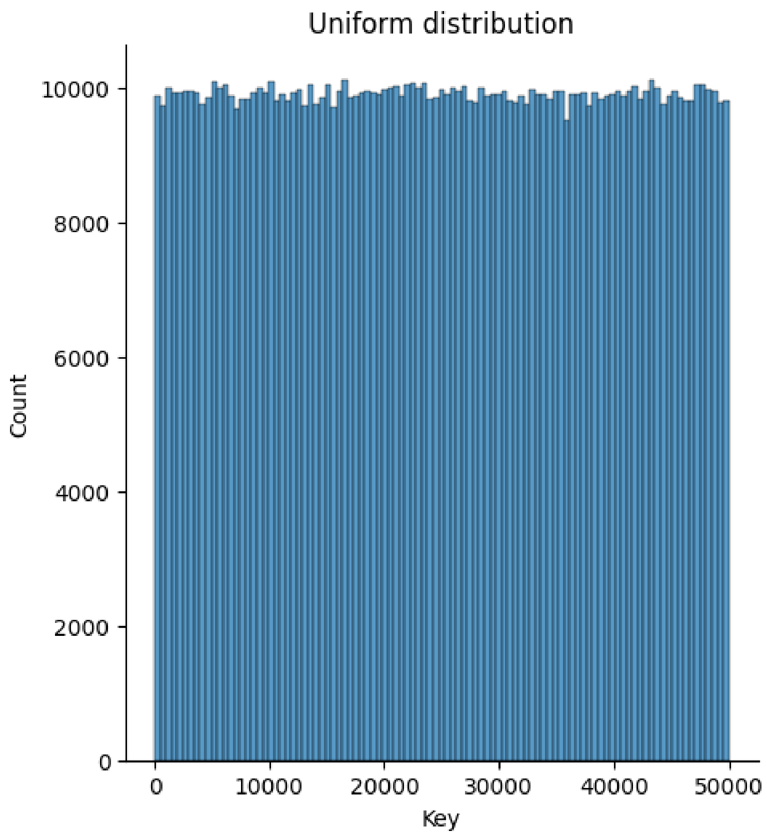


Figure 3.5: Uniform distribution within the interval of [1,50000)

The figure shows neatly how all the values in between the specified interval boundaries occur more or less evenly. This also become visible in the data summary at Table 3.5. The quantiles are spread out evenly, and the mean also lies just about in the middle ground between the set interval boundaries.

count	1000000
mean	24994.18
std	14430.80
min	1
25%	12496.75
50%	25012
75%	37478
max	49999

Table 3.5: Summary of uniformly distributed data sample

- Exponentially distributed dataset

  This dataset contains a highly skewed dataset following an exponential data distribution. It is also one of the more common continuous distributions that can be found. It takes in a parameter λ that is also referred to as the rate parameter. In this case this parameter was chosen as a large number, 250 in order to cover more keys within the distribution. This shapes the distribution for this purpose in the following way that is depicted in Figure3.6.

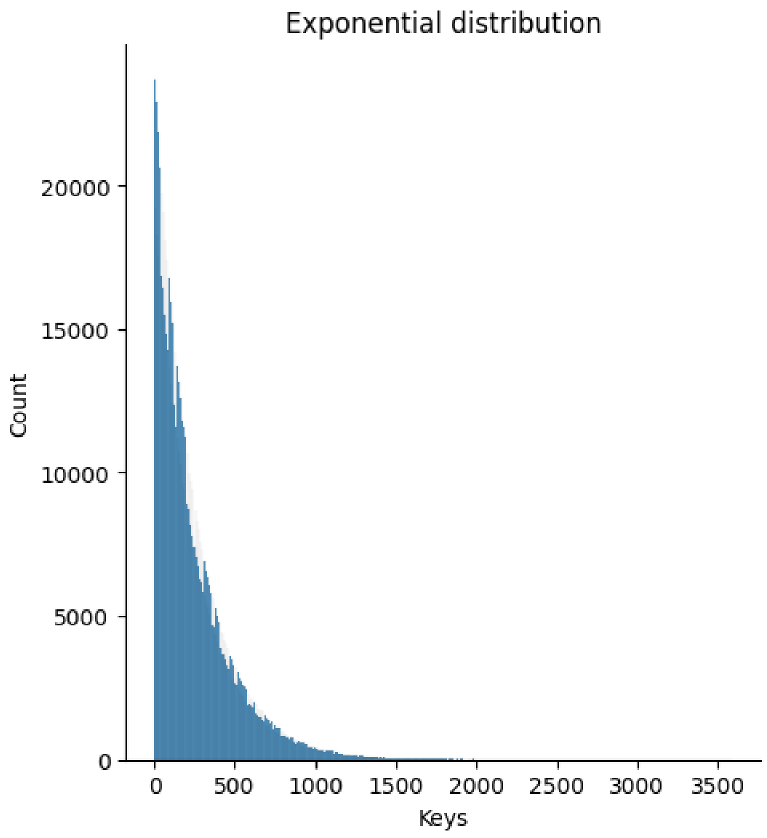


Figure 3.6: Exponentially distributed data with a scale of 250

count	1000000.00
mean	249.76
std	250.44
min	0
25%	72
50%	173
75%	346
max	3588

Table 3.6: Summary of exponentially distributed data sample

Table 3.6 shows that the range of the distribution going from 0 going up to a maximum of 3588. The mean of around 250 and the first 2 quartiles also confirm that the data is heavily skewed, with most half of the values smaller than 173.

- Data with many outliers

The last dataset is basically another equally distributed dataset with the only difference that there are several outliers with some distance to the rest of the data. This can be seen in Figure 3.7. The y-axis is shown with a log scale, in order to show the few outlying data points in comparison to the majority of the data. The main block of data here consists of 1000 000 data points, around the value of 2000 the 10 outliers are located. This distribution can also be seen in Table 3.7. We can see that the smallest key present in the dataset is 2073.29 and the biggest one being 27928.24.

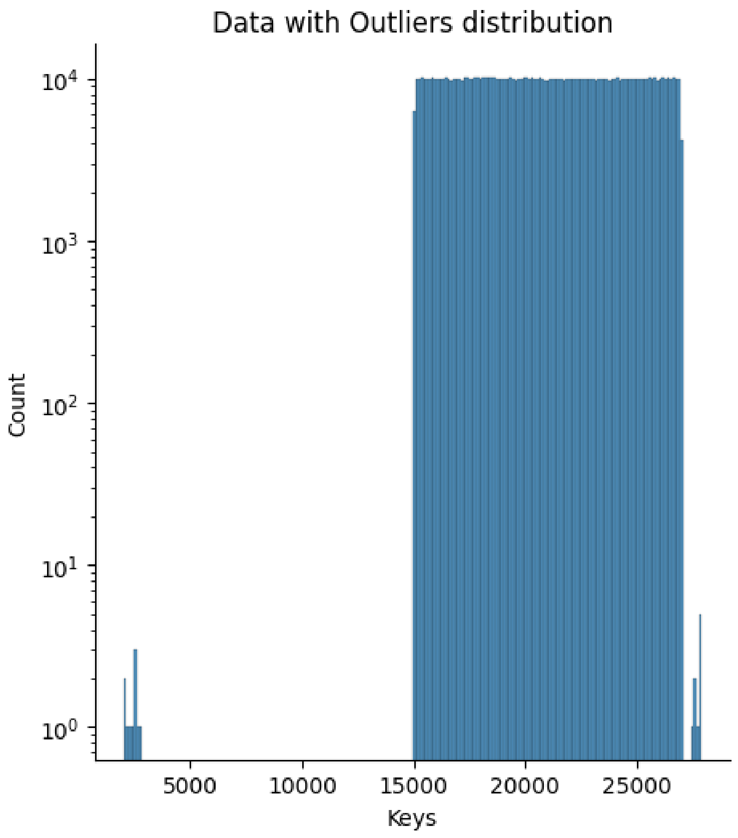


Figure 3.7: Uniformly distributed datasets with several outliers

count	1000010
mean	21003.90
std	3465.93
min	2209
25%	18004
50%	21001
75%	24009
max	27991

Table 3.7: Summary of uniformly distributed data sample with several outlying data points

3.3 Framework for Experiments

The requirements for the framework of running the experiments were primarily that different types of Learned Index can easily be build and analyzed back to back. Additionally, the experiments should be similar to the ones conducted in "Search on Sorted Data Benchmark" [14]. As this paper provides very detailed analysis that would exceed the scope of this thesis, a subset of experiments (which will be covered in Section 3.4), will be provided, additionally enhanced with custom experiments for answering the research questions.
For this purpose, the setup provided in [13], can be considered ideal, as it introduces a simple structure to run different experiments, as well as containing ways to create insightful summaries over the performance of the different Learned Index types on different datasets. This code is the basis of the paper "A Critical Analysis of Recursive Model Indexes" published in the "Proceedings of the VLDB Endowment" in 2022 by Marcel Maltry and Jens Dittrich of Saarland University.

3.4 Experiments

This section will briefly describe the experimental setup of each experiment concluded, each of them run on all the previously mentioned datasets. These experiments can be roughly grouped into several different categories, each focusing on a slightly different aspect of the whole picture this thesis aims to paint. For the experiments about lookup and build time, the described experiments were run for the above-mentioned Learned Index approaches as well as some traditional index approaches. This ensures that the results are put into perspective. The initial implementation of [13] was altered slightly to exclude the index approaches that cannot work with duplicate keys, as the main datasets included many duplicate values to be indexed. Also, other Learned Index approaches that were not included in the "Search on Sorted Data" benchmark were also excluded from the list. The pool of indexes that were considered for those experiments are therefore the following:

(a) Recursive Model Index

(b) RadixSpline

(c) Piecewise Geometric Model Index

(d) Binary Search

(e) B-Tree

Additionally, as for the above listed index structures do not have a "one size fits all" configuration, due to the fact that there are several parameters to consider when constructing the index. I evaluated and compared different combinations of parameters (which were mentioned in Section 2) in order to find the configuration that works best in the given scenario. Also, this is supposed to help to get an overall idea of what influence the different configurations might have on the performance. As those vary for each index structure, a short overview of the used configurations per index is given:

(a) *Recursive Model Index:* Size of second model layer. Ranging from 126 to 67108862.

(b) *RadixSpline:* Different amount of radix bits used to build the radix table: 8, 10, 12, 14, 16, 20, 22, 24, 26, 28. Additionally, the maximum error used to build the spline points: 1, 2, 4, 8, 16, 32, 64, 128, 256.The experiments build all possible combinations of those parameters, which will be further investigated in another set of experiments.

(c) *Piecewise Geometric Model Index:* Maximum error rate ϵ. Ranges from 1 to 8192.

(d) *B-Tree:* Sparsity of B-Tree, influencing its size. Increases by a factor of 2, starting from 1 going up to $2^{14} = 16384$.

(e) *Binary Search:* No parameters provided, therefore only simple Binary Search conducted, serving as a baseline for the other index approaches.

For these five different types of indexes it is important to note that the size of the index, will vary throughout some experiments. The experiments will work with constant dataset size and only the size of the index, which is influenced by the set parameters, varies. The result this change in index size has on the overall result will be discussed in detail in Section 4.

Below, each of the experiments is explained in further detail:

1. Lookup comparison experiments

   In this series of experiments, the main focus is on the comparison of the different datasets and index structures regarding how much time it takes to find a lookup key in the array of data given.To efficiently measure the lookup performance of different lookup keys in the data, a sample of 1 000 000 random samples from the data's distribution were drawn. As the lookup of a key that does not exist, would produce an error, including such numbers in the samples set was refrained from. In the experiment that followed this basic setup, the same pattern was followed throughout: the previously drawn samples would be used as lookup keys. These values would then be searched for in the indexed data. I then measured the time that it took to find every key from the drawn samples. The time that was needed there would be averaged, to make up for outliers of extremely fast and extremely slow lookup keys. This allows for a more robust estimation of the lookup time of each index. I then repeated this experimental setup a hundred times each, recording the total time one iteration would need. These results, along with several other basic setup information, I saved in a results file. Afterward, it would be possible to compare the results for each of the iterations, comparing the mean lookup times for each iteration. The experiments need to be robust against any other activity in the background and prevent (if possible) that segments that are stored in the cache and therefore falsify the results. Repeating each experiment with the same setup multiple times and additionally averaging the lookup time for the set of samples in each iteration aims to tackle those requirements.
   As stated previously, those experiments were repeated in the same manner for each of the implemented index structures and their respective range of possible tuning parameter. This yields several different outcomes for each dataset and index. These results will be analyzed and brought into context in Section 4.

2. Build time comparison experiments

   This series of experiments takes advantage of the structure already in place from the lookup time experiments, and therefore is also a part of the general comparison structure. Intuitively, before any of the lookup experiments on the different datasets, indexes and configurations can be done, the index has to be build first. For each iteration, meaning the combination of an index with its respective parameters, the time it takes to build the index (logically only once for the series of lookup keys) is measured and saved alongside the mean lookup time for each iteration done.

3. Detailed comparison of different configurations of the RMI

   The previous experiments will uncover the general performance of the different index configurations in regard to lookup- and build time. Another interesting part will be to find out what influence each configuration will have on this performance. As previously mentioned, the layer count itself will not be discussed, as previous experiments showed that two layers perform adequately, and adding the possibility of more layers had been omitted, therefore running all the experiments of deeper RMI understanding on the given fact that it is a two layer RMI. As a first step, the ideal combination of model types was analyzed in order to run the other hyperparameter tests on based of this important basis structure. For the purpose of covering all the important configurations, the different possible combination of hyperparameters for the RMI implementation of Maltry and Dittrich [13] will be looked at in more detail, exploring which combination of last miles searches and error bounds would perform in which way. Additionally, the size of the second layer and its influence on the performance of the index is analyzed, represented by the size of a model has.

4. Detailed comparison of different configurations of the RadixSpline

   For the RadixSpline, we also attempt to understand what influences the performance of this index structure in more detail. For this, the lookup and build time for each combination of the two parameters (radix size & error bounds) was analyzed in order to construct a heatmap with the best and worst performing pairs of configurations. In this experiment, we try to establish if there is a trend to be seen across the datasets, which might give us a rough guideline which parameters values will most likely give us a good performance of the index structure.

   In addition to analyzing the build time and lookup performance, an attempt to understand the data structure in the background is made, analyzing the distribution of the spline points and also the values present in the radixtable.

5. Detailed comparison of different configurations of the PGM Index

   This series of experiments analyses the performance of the PGM index and how the index structure works in the background. For this, the influence of ϵ on the model's performance was evaluated along with the deconstructing of the data structure in the back, trying to understand which part of the performance might be due to the way the models are represented.

6. Comparison to the current standard

   As one of the research questions is how adequate these new index structures might prove to be for the Dynatrace data, a bit of perspective will have to be established. As a first step of course the comparison to well known index structures/search algorithms, as done in the first set of experiments, serves well as a baseline to compare the Learned Index approaches to. Additionally, and to further put it into perspective for the systems that are established within the industry, the performance of an indexer that is widely used, Apache Lucene will be also analyzed and put into direct comparison.

All experiments mentioned before were run on a sample size of 1 million randomly sampled keys from the respective distribution of the dataset, repeating each experiment 100 times in order to make the results as robust as possible with the running times still at a decent scale. Furthermore all data and index structures used in the experiments are built and stored in the memory at runtime, which eliminates the effect a cold start could have on the results. Reading from the disk during any of the experiments is therefore not necessary and can not have an impact on the lookup of the search keys.

The setup used for all experiments was the same, running on an 11th Gen Intel Core i9-11950H @ 2.60GHz with 32 GB of RAM. The operating system was Windows 11.

The B-Tree implementation that was used is provided within the tlx library that can be found on GitHub and that is "A Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers" [2]. The B-Tree used in this library is self-balancing and therefore optimized for reads as well as supporting efficient write operations.

4 Results

This section is dedicated to describing the results from each of the experiments described in Section 3. Each of the experiments introduced will be handled and results put into context with the other experiments conducted to draw a complete and thorough picture of the impact of Learned Index structures. First of, the general structure that the results came in will be described, followed by the presentation of results for lookup and build time experiments as well as the detailed analysis of each Learned Index structure. Due to the complexity and size of the experiments that are done on a Dynatrace dataset, the detailed analysis will only be done for the results of the smaller dataset.

4.1 Result File Structure

Each of the experiments mentioned in Section 3.4 saved the results of each run in a .csv file, which stored different configurations and results as an output of the experiment, depending on the area of interest in each run. The basic structure of the results, which were analyzed in the next step, is described below:

4.1.1 Lookup Comparison Experiments & Build Time Comparison Experiments

As these experiments are very similar and run in the same script, measuring both the build time and lookup time for the different configurations, the analysis of the performance utilized the same results file. The basic structure of this file can be seen in Table 4.1.

dataset	n_keys	index	config	size_in_bytes	rep	n_samples	build_time	eval_time	lookup_time

Table 4.1: General structure of output for the comparison of lookup and build times for different index structures

Output variables: The file contains the following fields:

- *dataset:* A string holding the name of the dataset that the experiment was run on
- *n_keys:* The number of keys present in the dataset, referring to the total length of the dataset, not the unique values
- *index:* A string specifying the index structure used which is one of the index structures mentioned in Section 3
- *config:* A string specifying which parameters configurations native to each index structure was used
- *size_in_bytes:* Number of bytes occupied by this index structure with the specified configuration
- *rep:* A number indicating which repetition of the experiment the run belongs to: holding a number from 0 to 99, as the number of repetitions was set to be 100
- *n_samples:* Number of samples set to be looked for during the course of one run. In this case always set to 1_000_ 000
- *build_time:* Time in nanoseconds, that the building process of the specific index structure took in total
- *eval_time:* Time in nanoseconds, that it took the index to evaluate the given models and estimate an interval to search through, for each of the 1 mio lookup keys summed up. Part of the lookup time as well

- *lookup_time:* Time in nanoseconds, that the lookup of the sampled 1 mio lookup keys took in total. Summed up for the entire lookup process

Calculated variables After the experiments themselves were finished other columns were added, to make the analysis more readable (when dealing with very large numbers), and to put several things in perspective (e.g. the summed up lookup time for the entirety of the lookup samples). These additional columns were calculated based on values that the initial result set included. These columns added afterward included the following:

- *size_in_MB:* Derived from the size_in_bytes, this value converts the often very large and hard to read number of the size in a unit that is easier to comprehend and plot
- *build_in_s:* This column is also derived from previously existing values, in order to make them more readable, converting the previously given time in nanoseconds to seconds
- *eval_in_ns:* Measures the average evaluation time of the samples. Created by dividing the evaluation_time by the number of samples in that experiments, which is always 1_000_000
- *lookup_in_ns:* Measures the average lookup time in ns. Result of dividing the total lookup time by 1 000 000, as this is the total number of lookups in each run
- *search_in_ns:* Time spent (per lookup) to finding the value through last mile search. Difference of lookuptime_in_ns and eval_in_ns

4.1.2 Detailed Comparison of Different Configurations of the RMI

For this analysis of all the different hyperparameters for the Recursive Model Index, apart from layer size, similar experiments than those for the lookup and build time were conducted. In order to measure the impact of each hyperparameter, the model structure, last mile search algorithm and error bounds and size of the second layer model were part of the output. This produces an output with the general result structure shown in Table 4.2, for which the further analysis was done.

dataset	n_keys	layer1	layer2	n_models	bounds	search	size_in_bytes	rep	n_samples	lookup_time

Table 4.2: General structure of output for the detailed analysis of RMI parameters

Each of the columns holding values that describe different aspects of the index, which will be explained in more detailed in the following.

Output variables: The file contains the following fields:

- *dataset:* A string holding the name of the dataset that the experiment was run on
- *n_keys:* The number of keys present in the dataset, referring to the total length of the dataset, not the unique values
- *layer1:* A string specifying which model class was used for building the first layer of models, also known as the root model
- *layer2:* A string describing which model type the second layer is using
- *n_models:* The number of models making up the second layer of the model hierarchy
- *bounds:* A string indicating which error bounds strategy was used here

- *search:* A string describing which last mile search strategy was used to find the values based off of the estimation given by the second layer model and the error bounds calculated
- *size_in_bytes:* Number of bytes occupied by this index structure with the specified configuration
- *rep:* A number indicating which repetition of the experiment the run belongs to: holding a number from 0 to 99, as the number of repetitions was set to be 100
- *n_samples:* Number of samples set to be looked for during the course of one run. In this case always set to 1_000_ 000
- *build_time:* Time in nanoseconds, that the building process of the specific index structure took in total
- *eval_time:* Time in nanoseconds, that it took the index to evaluate the given models and estimate an interval to search through, for each of the 1 mio lookup keys summed up. Part of the lookup time as well
- *lookup_time:* Time in nanoseconds, that the lookup of the sampled 1 mio lookup keys took in total. Summed up for the entire lookup process

4.1.3 Detailed Comparison of Different Configurations of the RadixSpline

In order to analyze the performance of the RadixSpline, two different aspects of this index structure were saved in an individual output file.

Firstly, we saved the location of each spline point that was created, in order to approximate the Cumulative Distribution Function, in a result file. This file was a lot simpler than previous experiments, as *only the x and y coordinates of each point* needed to be written into a file. This resulted in a two column results file, listing all spline points in the best performing RadixSpline configuration.

The other half of this series of experiments, comparing the impact of the different parameters, similarly to the detailed analysis for the RMI index, saves the exact values of each parameter alongside the respective times for both lookup and build time. This results in a table with the following structure to be found in table 4.3.

dataset	n_keys	max_error	radixbits	size_in_bytes	rep	n_samples	build_time	build_time	lookup_time

Table 4.3: General structure of output for the detailed analysis of RadixSpline parameters

Each of the columns holding values that describe different aspects of the index, which will be explained in more detailed in the following.

Output variables: The file contains the following fields:

- *dataset:* A string holding the name of the dataset that the experiment was run on
- *n_keys:* The number of keys present in the dataset, referring to the total length of the dataset, not the unique values
- *max_error:* This value states the maximum spline error used in the described setup
- *radixbits:* The amount of bits indexed in the radixtable.
- *size_in_bytes:* Number of bytes occupied by this index structure with the specified configuration

- *rep:* A number indicating which repetition of the experiment the run belongs to: holding a number from 0 to 99, as the number of repetitions was set to be 100
- *n_samples:* Number of samples set to be looked for during the course of one run. In this case always set to 1_000_ 000
- *build_time:* Time in nanoseconds, that the building process of the specific index structure took in total
- *eval_time:* Time in nanoseconds, that it took the index to evaluate the given models and estimate an interval to search through, for each of the 1 mio lookup keys summed up. Part of the lookup time as well
- *lookup_time:* Time in nanoseconds, that the lookup of the sampled 1 mio lookup keys took in total. Summed up for the entire lookup process

The fields in this series of experiments are once more following a similar structure to those mentioned before, with a couple columns that capture the parameters used for a proper RadixSpline setup:

- *max_error*: This values states the maximum spline error used in the described setup
- *radixbits*: The amount of bits indexed in the radixtable

4.1.4 Detailed Comparison of Different Configurations of the PGM Index

Analyzing the performance of the Piecewise Geometric index comes in two different parts. First off, deconstructing the underlying structure of linear regression down to the shape and size of all layers. This was done, saving all the triples that represent one of the models used in the index, per layer. For this, the *key, slope and intercept of each model* in each of the layers was saved in order to construct the skeleton of indexed points later on and to be able to revisit the way the CDF is estimated.The output file has the following structure:

Key	Slope	Intercept

Table 4.4: General structure of output for the detailed analysis of PGM structure

Output variables: The file contains the following fields:

- *Key:* The smallest key in the data segment represented by this triple
- *Slope:* A number specifying the value of the slope which is an important metric for the linear model of the current segment of data
- *Intercept:* A number specifying the value of the intercept of the triple representing the data, which is an important metric for the linear model of the current segment of data

The next half of the results was merely derived from the already existing results file from the index comparison experiments, as can be seen in Table 4.1. As this index structure only needs one type of parameter as an input, this value was extracted from the "config" column of the bigger dataset. The output file for these experiments therefore looks identical to Table 4.1

4.1.5 Comparison to the Current Standard

For evaluating the difference to the Apache Lucene performance, the result files for the index comparison were extended to also include results building and running an index with Lucene. This way the results can be directly compared side to side, taking advantage of the existing structures given.

4.2 Results Index Comparison Dynatrace Datasets

In this section, the results of the previously described experiments comparing the different index structures with each other will be presented, as described in Section 3.4 as experiments 1 and 2. All experiments in this section were done on both Dynatrace dataset and the synthetic datasets. This section will cover the results for the two Dynatrace datasets, with the results for the synthetic data, to be found in a later subsection of this section.

To start off with, the results of the experiments measuring build and lookup time are presented, as described in Section 3.4 experiments 1 and 2. Figure 4.1 holds the results for the build performance of each previously described index structure applied on both Dynatrace datasets. These graphs exclude Binary Search, as this algorithm does not need to be built beforehand. For better readability of the graphs, the y-axis and the x-axis are shown as log scaled.
The general trends that can be seen on the smaller dataset already is that the Learned Index approaches, such as RMI, PGM and RadixSpline generally have longer build times than the B-Tree. RadixSpline and PGM seem to have very similar build times, which makes sense, considering that one of the characteristics of those approaches is that for building the data just has to be passed over once. Building the recursive model index on the other hand is slower, no matter how big the index gets, even significantly increases once the size of the index passed 10 MB. RadixSpline and PGM stay on a constant level, and also do not show a large size range. Building the non-Learned Index approach, the relationship between index size and the building time seem to be somewhat linear, meaning that a steady increase of index size also results in the same increase of build time. This difference in build time speed can also be seen when calculating the median of all build times for each index, which can be seen in Table 4.5. This also verifies that the similarity of the RadixSpline and PGM build times. With a build time of 0.09 ms the B-Tree is by far the fastest, which is due to the fact that a lot of the configurations in the smaller size category (implying that there the numbers of leaves of the tree is very little, thus making each leaves responsible for a lot of different values).

index	build in s
B-Tree	0.00009
PGM-index	0.01383
RMI	0.03758
RadixSpline	0.01358

Table 4.5: Median of build times for the small Dynatrace dataset

When comparing the build performance of the bigger Dynatrace datasets, holding about 50 million values, it is visible from the start that the general trends do not change significantly, as B-Tree still seems to have the smallest build time, RadixSpline and PGM performing similarly and RMI performing slightly better, with a wide range of size. A noticeable difference is that with bigger index sizes of the RMI the build time seems to stay comparably stable this time. Possible reasons for this behavior will be analyzed in Section 5. What is also remarkable is that the build times of all mentioned index structures do not increase at the same rate the values increased.

When looking at the results for different index sizes (therefore configurations of each index) for the lookup times, as seen in Figure 4.2, the picture looks quite different than for the build time experiments as seen in Figure 4.1. The graphs show a side-by-side comparison of the lookup performance (measured in nanoseconds), for each of the three Learned Index structures, B-Tree and shows the performance of

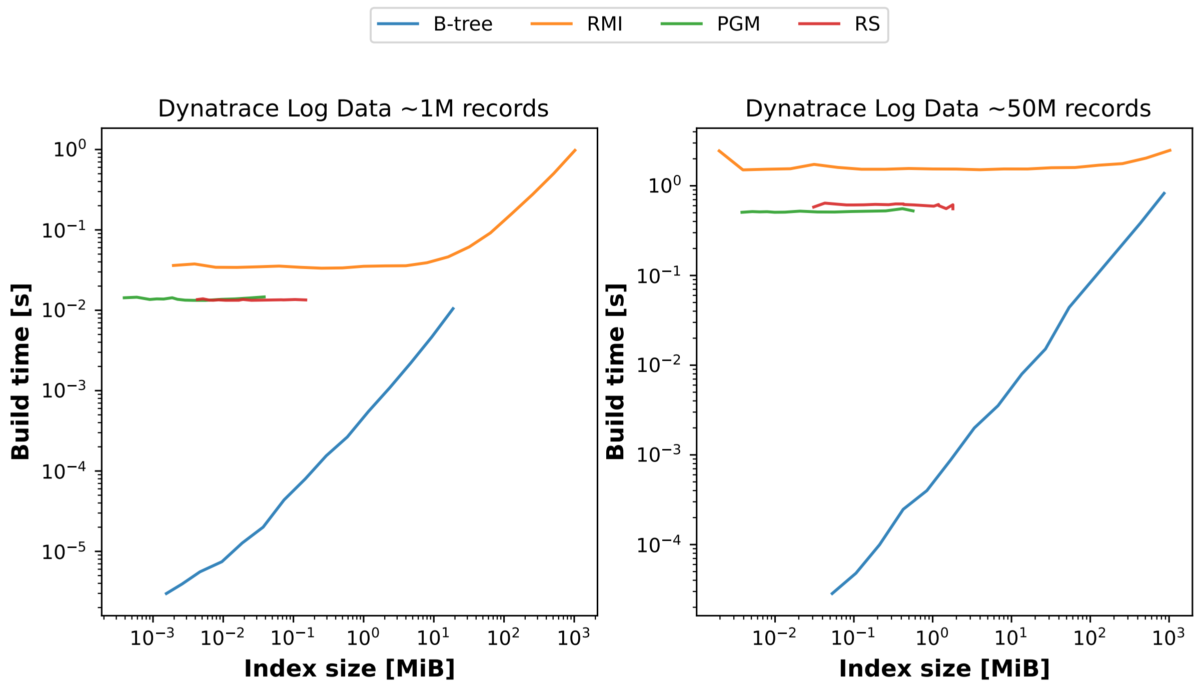


Figure 4.1: Comparison build time of different index approaches

the Binary Search to find the desired lookup key as the black dashed line, to serve as a baseline. The x-axis is shown as log scaled.

Starting with the analysis of the dataset with 1 million records, it is clear that not all of the Learned Index approaches perform better than their non-learned competitors. The PGM index is obviously an outlier in this picture, with a performance on average around 373 ns (see Table 4.5). With the Binary Search baseline at 213 ms, the B-Tree only does better at a certain size, whereas the rest of the Learned Index approaches, like RMI and RS are outperforming both non-Learned Index structures, independent of the size of the index structure. A general trend is that the Learned Index structures offer fast lookup times with increasing index sizes (at least up to a certain point), given that the unterlying data retains the same size. meaning that the index of the same size on a bigger or smaller dataset will not yield the same results. This can be attributed to the fact, that building more and therefore more accurate models, decreases the error of the Learned Index structures, resulting in a better prediction in the Evaluation stage and smaller search intervals in the last mile search stage. Important to highlight is, that the size of the search intervals are always relative to the datasize. A search interval that is considered big on a smaller dataset, might be rather small for a bigger dataset, that's why it is always important to also take the size of the dataset that the index is built on, into consideration.

Such a trend can also be seen in the B-Tree as the increased size of the index is tied to a more dense kind of index, with the nodes holding a smaller number of records, therefore needing more nodes to accurately depict the size of the data. The trend for the B-Tree does point downwards as well, though the difference is not as big as observed within the Learned Index structures. Especially with a bigger index size of the B-Tree, meaning a bigger number of nodes making up the B-Tree to traverse through, the lookup performance worsens visibly, which can be attributed to the fact that the evaluation(searching through all the small nodes in the tree) increases at a faster rate than the search time decreases.

With a lot more configuration freedom, the RMI again shows a broader range of size with the performance increasing, meaning faster lookup times up until a size of around 0.1 MB where the performance does not increase a lot, when increasing the size, therefore building more and smaller/more accurate models. RadixSpline does not have an equally big size range, but nevertheless competing with RMI for the fastest index structure, with better lookup performance the bigger the index gets, which is due to the fact that more spline points form smaller search intervals but also more memory usage. Solely judging by the median lookup times in Table 4.6, RMI seem to do slightly better with 79 nanoseconds, as opposed to RadixSpline with 108 nanoseconds on average.

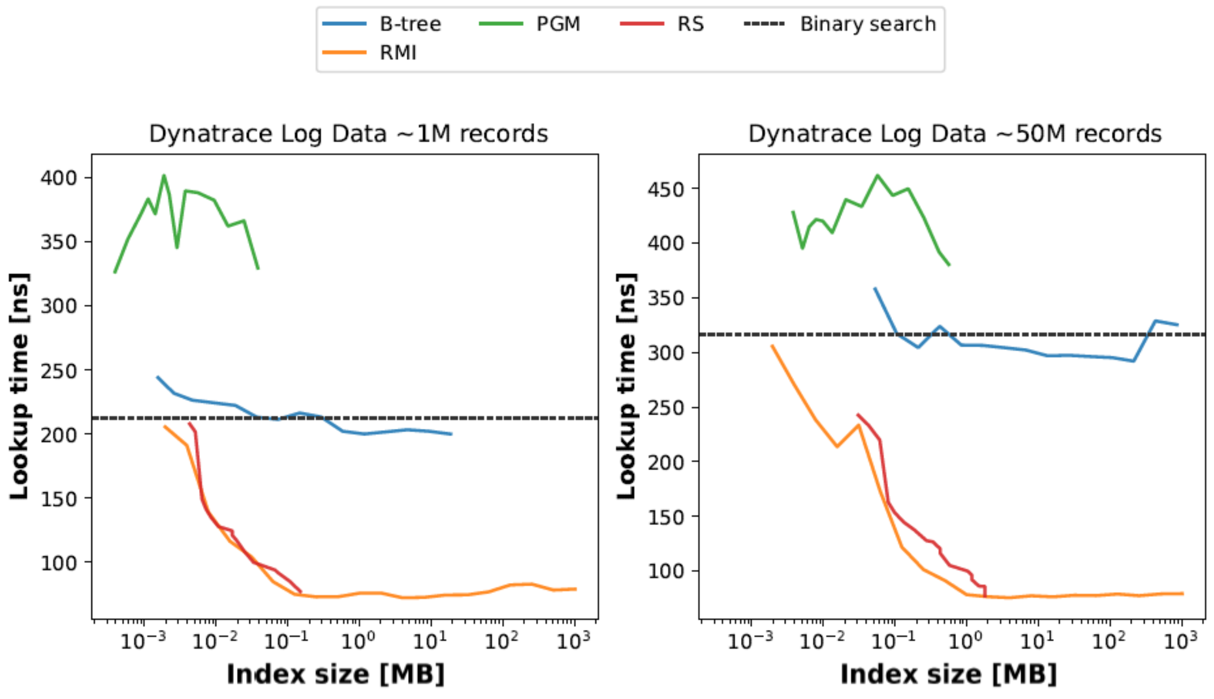


Figure 4.2: Comparison lookup time of different index approaches

	lookup in ns
B-Tree	213.39
Binary Search	212.79
PGM-index	373.64
RMI	79.450
RadixSpline	108.00

Table 4.6: Median of lookup times for the small Dynatrace dataset

When comparing the graph lookup performance of the bigger Dynatrace datasets in Figure 4.2 the trends seem to be very similar to the ones previously observed. PGM again lands on the place of the worst performing Learned Index approach, B-Tree seems to be comparable or only slightly better than the baseline lookup time of just over 300 ns. RS and RMI are performing somewhat similarly again, with RMI having the more visible better performance this time, meaning it manages to approximate the CDF of the bigger dataset better in general. It also seems that the "sweet-spot" of performance and optimal size seems to not be the largest models possible, but to lay in the middle, around 1 MB. A fact that is interesting to see is that the lookup times for the significantly bigger dataset does not seem to differ by more than 100 ns for each index. compared to the lookup times found in the smaller dataset, seen on the left of Figure 4.2

Putting all of the above-mentioned graphs and numbers in the thesis up to this point into perspective, a comparison of each best performing configuration is needed. This can be seen in Figure 4.3. This figure separates the lookup time into the shares that evaluation and search have on the total lookup time. The previously mentioned lookup time is therefore the sum of the time the index needs to evaluate the approximate location in the data (evaluation time), and the time it needs to look through the search interval and find the record matching the lookup key (search time).
This confirms the picture seen in the lookup time experiments from Section 3.4, with RMI and RS providing the index structures with the fastest lookup times, outperforming the best B-Tree version by a factor of approximately 3. An interesting insight is visible through the differently colored shares of lookup time. Therefore, for both RMI and RS, a big part of the total lookup time is spent searching through the interval, that is the result of the previous evaluation step, analyzing the given models. Whereas, the relatively high lookup time of the PGM seems to be due to the fact that the evaluation of layered models seems to take longer than if a simple Binary Search was just applied to the data.

An almost identical picture to the results from the smaller Dynatrace dataset can be drawn from looking at the results for the bigger dataset. The shares regarding the lookup time stay approximately the same, with RMI and RS competing for the fastest lookup again, both delivering their fastest results in just under 100 ns. PGM on the other hand takes more than 350 ns to get to the same conclusion, therefore taking longer than both B-Tree and Binary Search, both taking around 300 ns to find the wanted key.

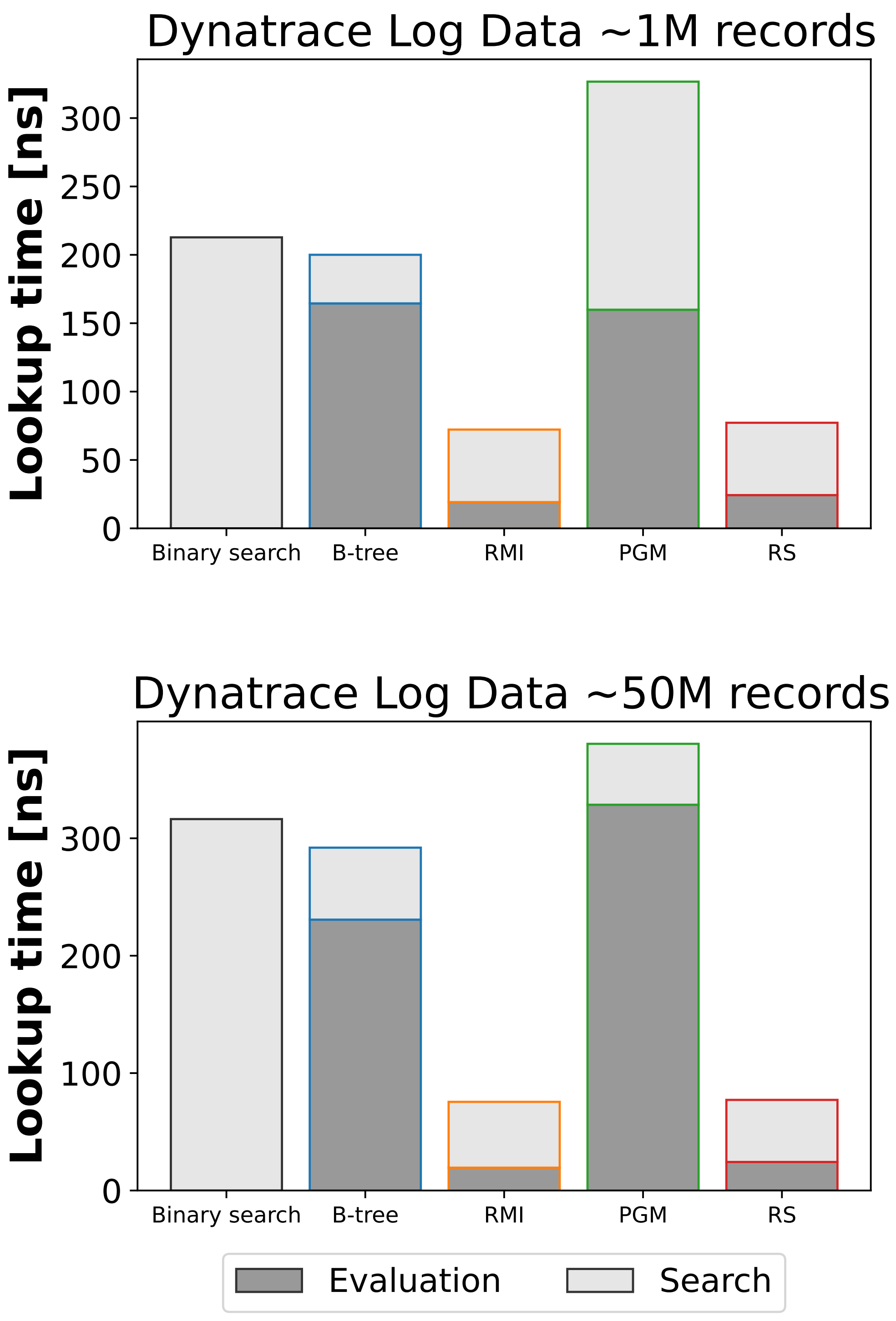


Figure 4.3: Comparison lookup time of best performing configurations

While those are only the results of the best performing configuration with the deconstructed lookup time, a full analysis on all available configurations (similar to the graph seen in Figure 4.2) can be found in the Appendix. There it can be seen that the relationship between search and evaluation time basically stays the same throughout the different size ranges and is not just a characteristic of the best performing combinations.

4.3 Results Detailed Analysis Recursive Model Index

Up until now it had been established how the Recursive Model Index works, which possibilities exist to tune the index exactly to the data that is available. In order to illustrate how different the outcome would be if any of the parameters were chosen differently to the way they are chosen to find the best configuration, this section will highlight some of the most important combinations of search strategy and error bounds.

4.3.1 Influence of Different Model Types, Search Strategies and Error Bounds

In order to wrap the maximum amount of insights into the results graphs of the experiments described in 3.4 and also to showcase the tight relationship search strategies, error bounds and model types the influence of them is plotted together. Figure 4.4 shows the results for both Dynatrace datasets for the combination of a linear regression model in the first layer and a linear spline model as the main second layer model. The differently colored lines are each representative of a different combination of search strategies and error bounds and showing the behavior of each of those for a certain index size (which can roughly be translated to: bigger size results in bigger second layer model count) and the lookup time in nanoseconds.

RMI Search Strategies: The search strategies considered for this graph are the following, which are the abbreviations found in the "Search" column of the RMI output file found in Table 4.2:

- Binary Search (Bin): Does not take into account the estimated positions as given as an output by the second layer models
- Model-biased Binary Search (MBin): Binary Search taking into account the estimated position as a starting position for the Binary Search instead of the middle of the entire data array
- Model-biased Linear Search (MLin): Linear Search done on the data, starting from the estimated position as an output of the second layer models
- Model-biased Exponential Search(MExp): Exponential search tweaked so that the search considers the estimated position of the data

RMI Error Bounds: Apart from a variety of search strategies that can be used to find the exact location of a record in the data, the model also saves error bounds, which can fall in any of the categories described below. These are the abbreviations that can be found in the "Bounds" column of the RMI output file found in Table 4.2

- Local Individual (LInd): Storing the maximum negative and positive error on an individual model level. Makes the error bounds generally more robust against outliers that are not present in the current model
- Local Absolute (LAbs): This only stores the maximum absolute error, also on an individual model level
- Global Individual (GInd): Memorizes both the maximum positive and negative error over the whole RMI structure
- Global Absolute (GABs): Stores the maximum absolute global bound over the whole RMI. More space efficient but more sensitive to outliers
- No bounds: No bounds are stored

Now, when considering the results shown in Figure 4.4, it becomes clear, that the combination of Model-biased Exponential search and No Bounds does the best out of all the other combination, for both datasets. While it is a more obvious for the Dynatrace Dataset, especially for the Linear Spline on Linear Regression model with a million records, where the lookup time rapidly decreases when increasing the layer count and therefore the size of the model and then stays at a relatively stable level when going beyond 0.05 MB until the largest configuration of a little less than 1000 MB. This trend of reaching the sweet spot in the middle size range also holds true to the bigger dataset. However, the other 6 combination of search strategies and error bounds do not show such a stable behavior, with several spikes showing in the other curves, in addition to the fact that the lookup time is significantly higher in all size classes.

While the structure and logic of those results is established, it has to be noted that this exemplary snippet is only one of several possible combinations of models. A wide variety of different models was compared, with every possible combination for two layers with different models from the following types:

- Cubic Spline (CS)
- Linear Spline (LS)
- Linear Regression (LR)
- Radix (RX)

The full results with the 16 different versions of the above-mentioned results graph can be found in the Appendix.
One important snippet to analyze, as this was the fastest combination coming out of this analysis and also the tuning for all further experiments when we refer to the RMI, is the one seen before in Figure 4.4.
The same combinations of error bounds and search strategies can be found here, in the first pair of graphs there is a linear model as the root model and Linear Spline models in the second layers, while the second pair of graphs shows the opposite case, with Linear Spline in the first layer and Linear Regression in the second layer models, both for the two Dynatrace Log datasets. Considering that the y-axis show the same unit, we can see that the best performing combination are very similar whether Linear Regression and Spline are switched around or not, though the combination with the Linear Spline in the first and Linear Regression in the second seems to be slightly faster, regarding the lookup time, especially in the larger dataset. For both datasets and both combinations of model types, the best performance comes from the variant without stored error bounds and a model-biased exponential search. The best performance here can be found at a model size of 1 MB, which is in the very middle of the size field and an increase in models in the seconds layer (therefore a bigger memory footprint) does not seem to automatically result in better lookup times.
What is interesting, is, that when switching around the model types in each layer, the other combinations of models which previously are very competitive, take a while longer when using Linear Regression in the second layer, therefore making in clearer that LS->LR are the best performing model types.

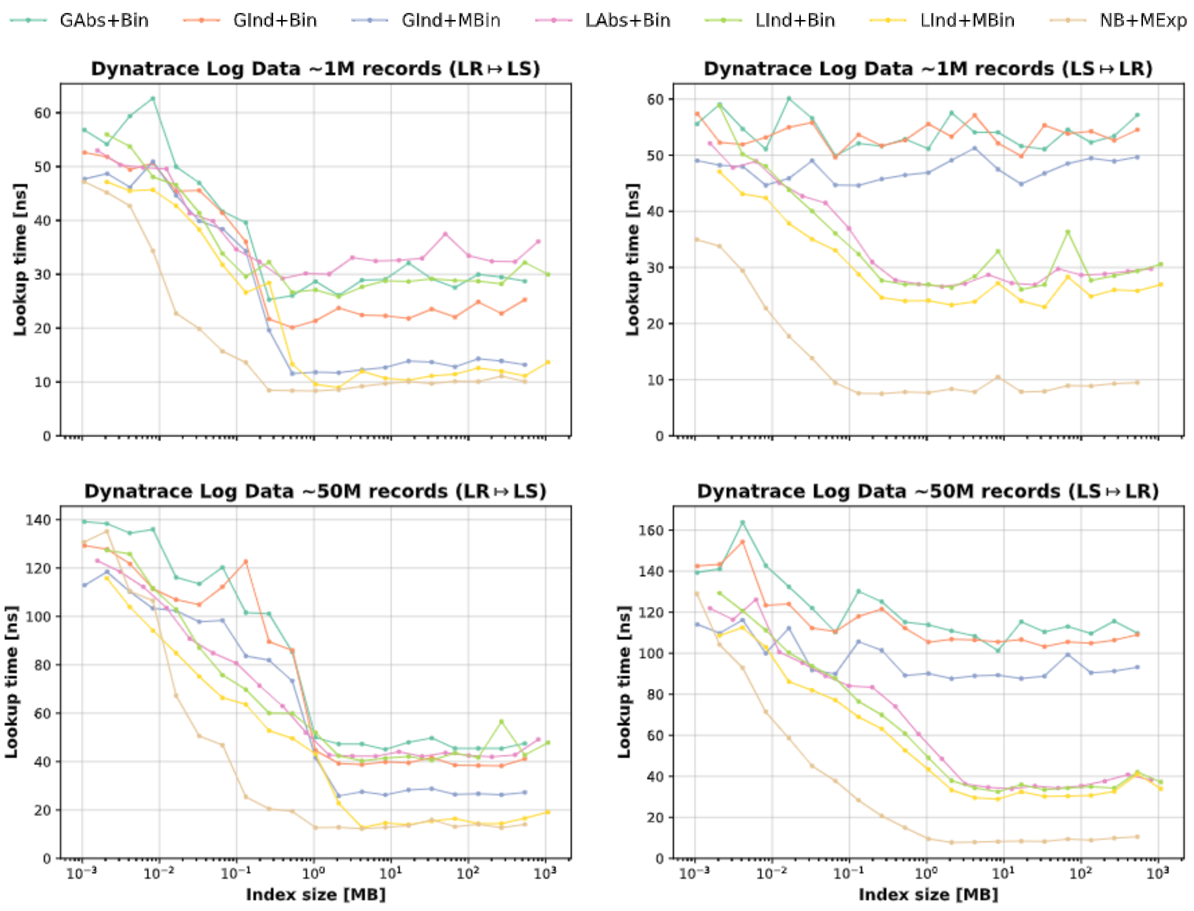


Figure 4.4: Results for different search strategies and error bound of an RMI with the combination of Linear Regression and Linear Spline

4.4 Results Detailed Analysis RadixSpline

In Section 4.2, it was established that the RadixSpline approach of indexing data is among the best when it comes to lookup performance, while also being relatively quick and straightforward to build. This section tries to establish what relationship the parameters of the RadixSpline model have to its performance, both on building and lookup time. Additionally, the way the index works in the background is illustrated.

Figure 4.5 shows the relationship between the two parameters that we can influence when building the index: Number of significant radix bits and spline error. The colors of the heatmap ranging from lighter colors indicating slower building times of the current combination of parameters to a darker green, with the darkest spots indicating the fastest configurations. The range of build times, measured in ms, is using different scales for the two different datasets. For build times of the index for the smaller Dynatrace dataset, it is rather hard to identify a clear trend, as there are darker/faster combinations all throughout the figure. This can be due to the data distribution, or because the dataset is still small enough to make up for small performance differences in the long run. The fastest configuration, though, consists of 22 relevant radix bits and a spline error of 128, followed closely by a combination with significantly less radix bits. Overall, the building times seem to not differ throughout the whole dataset as the maximum range of building times is only around 4 ms.That fact is also backed by the almost straight line when comparing the building times of the different index structures, as seen in Figure 4.1.

When comparing that to the result of the bigger Dynatrace dataset on the right-hand side of Figure 4.5, the better configurations are easier to identify. In general, there seem to not be that many combinations of parameters performing similarly well. Generally, the combinations which seem to do especially well are located in the middle of the range of radix bits, from 14 to 22, in combination with the whole range of spline errors. The best configurations with about 640 ms to build can be found when the number of relevant radix bits is set at 14 and a spline error of 64. Also, a way smaller spline error, specifically an error of 1, in combination with more relevant radix bits, 22, achieves a similar result.

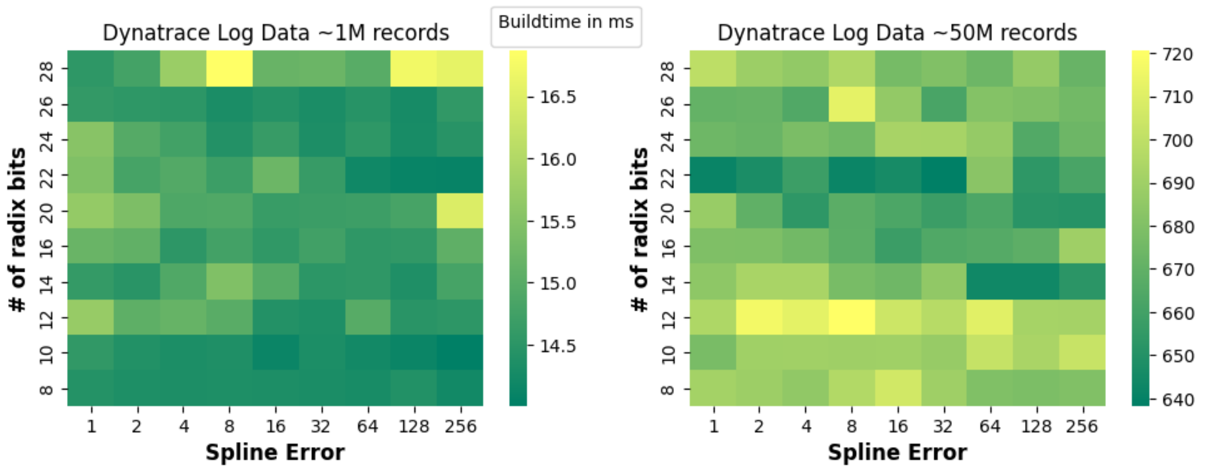


Figure 4.5: Relationship of RadixSpline parameter to build time performance

The same approach of comparing the different possible combinations of parameters was used to compare the influence on the lookup time, in Figure 4.6 with the heatmap showing the difference in speed of lookup times, measured in nanoseconds(ns). This time, the relationship between the two model parameters, spline error and radixbits is clearer: the faster dark and slower yellow patches are clearly distinguished and do exceptionally well in certain areas. For the dataset with around 1 million records, the lookup performance seems to be especially slow when the number of radix bits is chosen to be 8, which means that the radix table itself is smaller, therefore indexing less relevant spline points and leaving bigger areas of spline points to be searched through. The slowest configuration, therefore can be found when the spline error is also chosen to be very high, needing 220 nanoseconds for finding the relevant lookup key in the data. The other side of the spectrum, holding the combination with the fastest lookup times, finding the result in under 100 ns, are achieved when the number of radix bits is very high (26 or 28 bits), while at the same time satisfying a very small spline error of 1 or 2.

A similar result than the one from the small Dynatrace dataset can be found on the bigger Dynatrace datasets, holding just over 50 million records. Interestingly, the lookup times are almost the same as when searching through the smaller dataset, with the fastest combinations taking just 100 ns and the slowest configurations taking a little more than 275 ns. The trends in which configurations are more successful are also almost identical to the smaller dataset, deeming a combination with a smaller number of relevant bits and at the same time big error to be the least effective, while on the other hand large quantities of radix bits and a simultaneously small error seem to guarantee faster lookup times. Generally, choosing the error too big, seems to also have implications on the lookup time, slowing the lookup down to some extent, whereas the wrong choice of radix bits, does more damage overall.

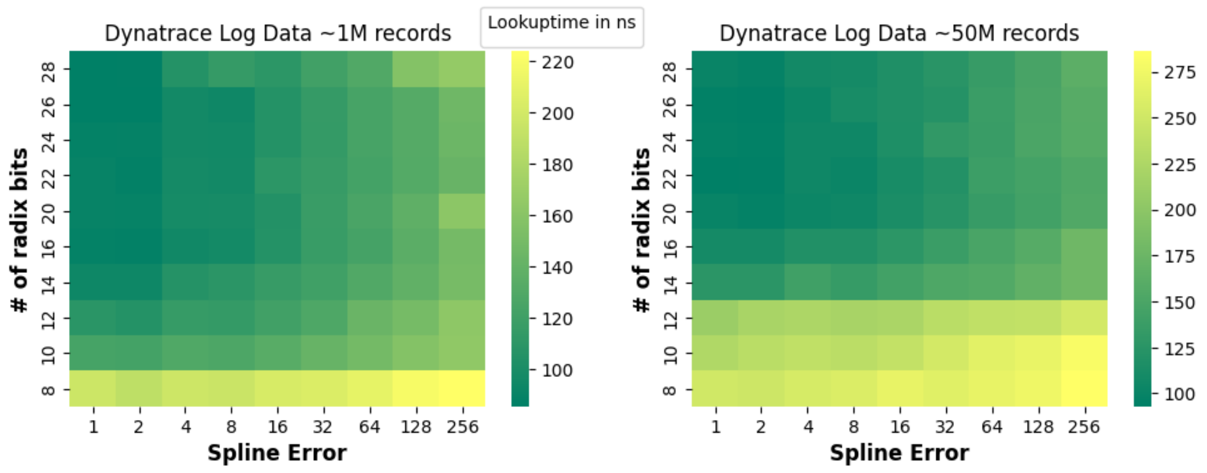


Figure 4.6: Relationship of RadixSpline parameter to lookup time performance

The way the RadixSpline algorithm works has been extensively discussed in Section 2, but to actually illustrate how the index tries to approximate, Figure 4.7 was created on the smaller Dynatrace dataset. It covers the distribution of the spline points of one of the fastest combinations of parameters, according to the previous analysis, with a given spline error of 1 and the number of relevant bits to be indexed in the radix table as 26. The x and y-axis of the graph, are representing the value of the key and the respective value of the Cumulative Distribution function, eventually adding up to 1. 6567 spline points were built on the initial dataset, building the estimated CDF, with the spline point population more dense, where large amounts of data are found and leaving some gaps in areas where the maximum spline error covers the difference between the respective values. All together, the representation of the real CDF of the data is done really well, when comparing the position of the spline points to Figure 4.8, showing the actual CDF of all data points.

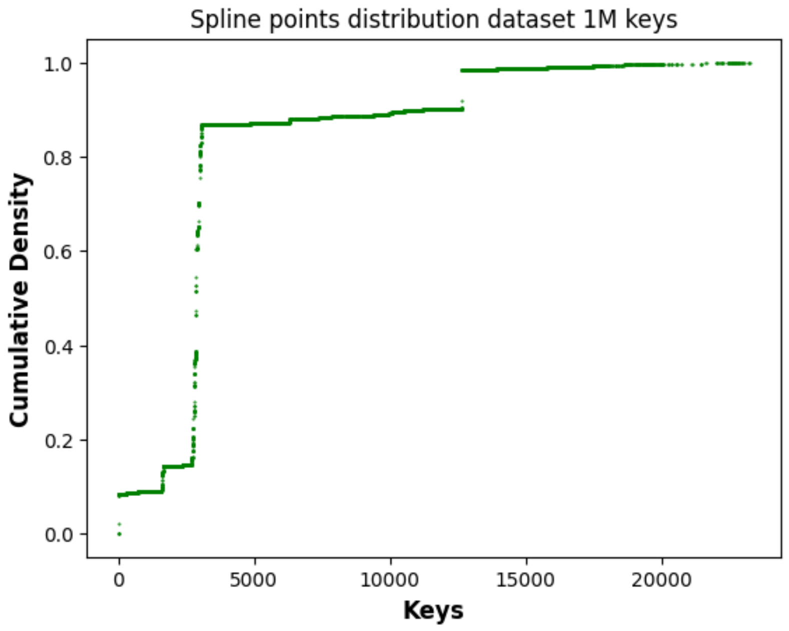


Figure 4.7: Distribution of Spline Points for RadixSpline Index

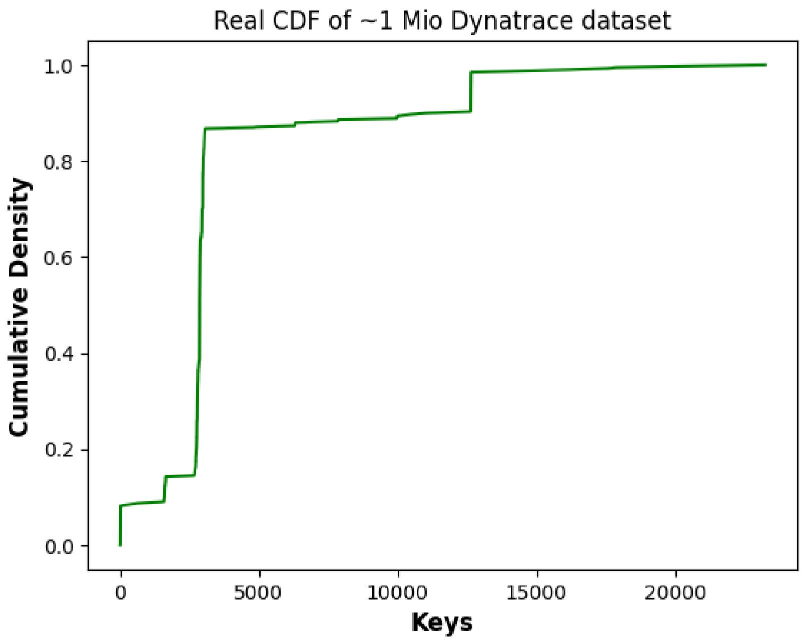


Figure 4.8: Actual CDF of the small Dynatrace dataset

4.5 Results Detailed Analysis Piecewise Geometric Model Index

As previously shown, the performance of the Piecewise Geometric Model Index does not seem to hold up to the performance of the other Learned Index approaches. In this section, the influence of the parameters used in the index is analyzed and the internal structure of the index is visualized.

In a first step, the influence of the only parameter needed to tune this index, the size of the error term

of each piecewise linear model, is calculated, in regard to the building time. Figure 4.9 shows how the build time develops when the error size is changed. In order to see the differences better, areas, where there are a lot of changes in the build time are looked at separately. Therefore, the first row gives an overview on how the building time (shown in seconds) develops when the error size moves up from 1 to its maximum value, 8192 which is an arbitrary number given by the decision to choose the error term as an exponentiation of 2 capped at 13.
In the first row, the results for the 50 mio Dynatrace dataset (on the left) and the 1 mio dataset (on the right) are shown. For the first one, it is clearly visible that the largest building times can be found among the smallest error sizes, which, considering that small error terms, mean build more models making up the index structures, seems very intuitive. This phenomenon can obviously be found in both datasets. While for the bigger dataset the building time is significantly larger than the next slowest configuration found when using an error term of 128, the difference for the smaller dataset is a lot smaller and also occurring at a larger value of error term. Furthermore, the building times for the larger dataset seem to also be decreasing faster when considering bigger error bounds as opposed to the smaller dataset, where with bigger error terms the building time does not go down as consistently.
Regarding the configuration with the smallest building times, the trend seems to be quite similar for both datasets, as they seem to occur on relatively small error bound considering the range that is offered. As can be seen in the third row of Figure 4.9, the dataset holding more records seem to have it optimal building time when choosing an error term of 32, while the smaller dataset has its fastest building time using just half of those maximum error bounds, with their optimal value at just 16. That is interesting, knowing that bigger error sizes mean that a smaller number of models need to be created, but obviously take longer to build altogether.

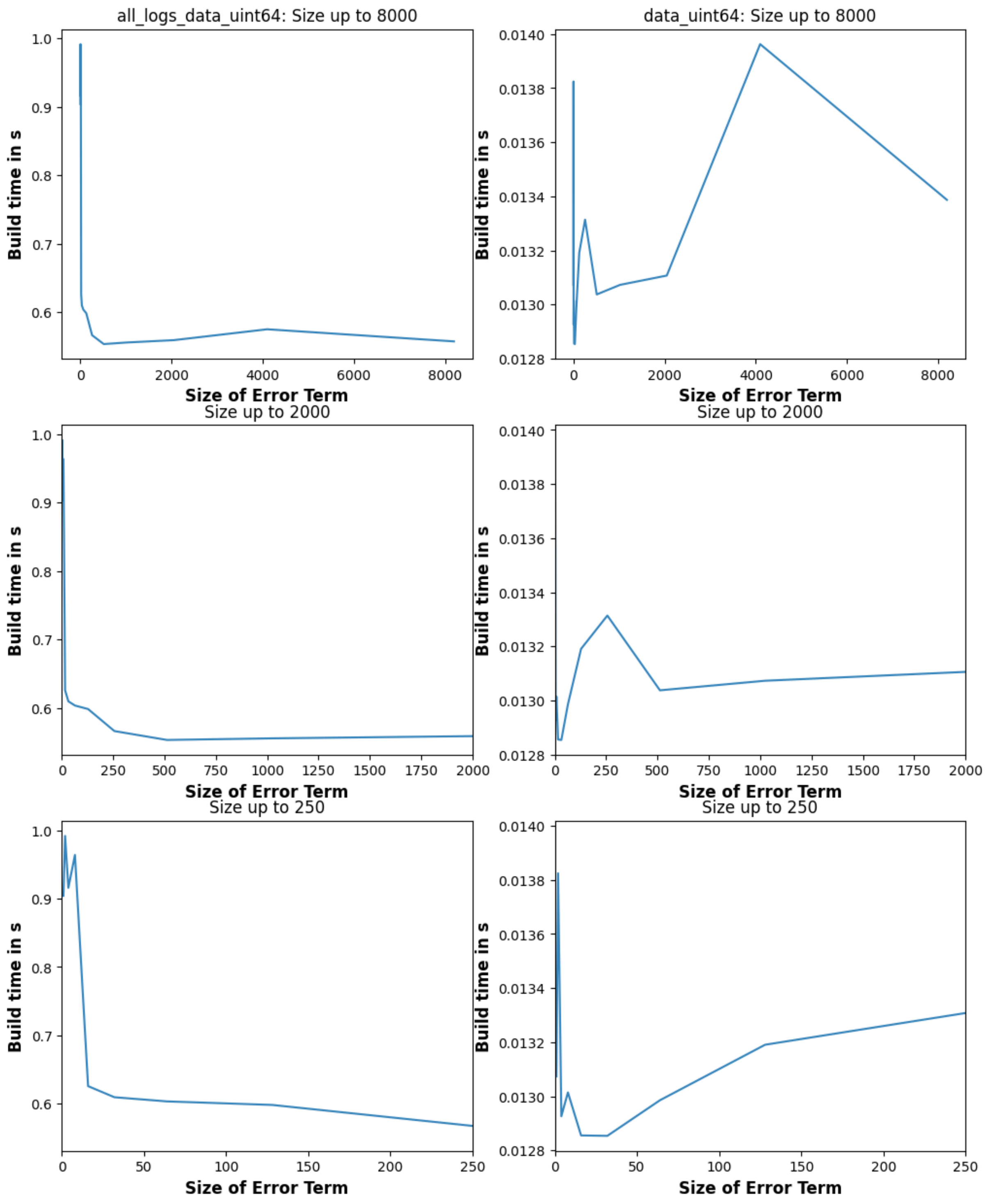


Figure 4.9: Influence of PGM error term on build performance

The same analysis was done, just instead of measuring the impact of the error term on the build time, the lookup time was considered in Figure 4.10, where the two Dynatrace datasets are analyzed on the influence the different choice of error term has on the lookup time, measured in nanoseconds. Here the fluctuations seem to be even closer together, that's why a closer look at certain parts of the graph covering the whole range of possible error terms is even more important.

Interesting about the both datasets, is that the smallest possible lookup times are really far away from each other. In the bigger dataset, which also offers a wider range of different keys, a small choice of error term does not yield the best results, but rather seems to perform best when the error term is 4096, with just under 450 ns of lookup time (443 ns), slightly outperforming the next best configuration at 250, taking just about 450 ns.The fact that the error term is chosen to be so big can also be seen in the previously discussed share of both evaluation time and search time. As the interval, which has to be searched through according to the initial evaluation is significantly bigger, the search time also makes up a bigger part of the overall lookup performance as can be seen in Figure 4.3. Whereas looking at the results for the smaller dataset on the right in Figure 4.10, obviously the best choice for the error term yielding the best results is by choosing the smallest value possible, 1. Even though when increasing the error size the performance at the end also dramatically increases again, the best lookup is achieved when choosing a maximum error of one, resulting in an average time for the lookup of 306 ns.

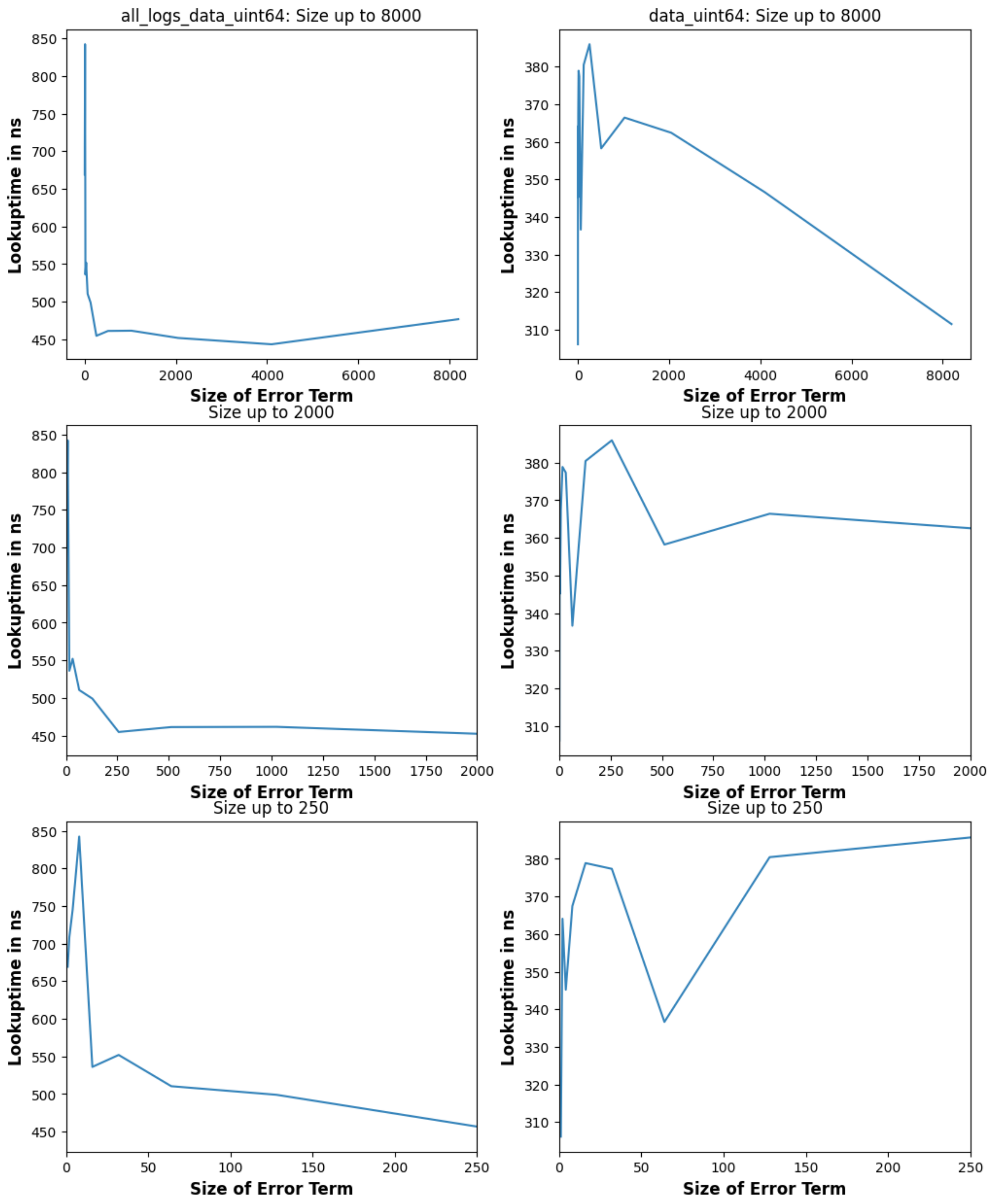


Figure 4.10: Influence of PGM error term on lookup performance

Now knowing the differences between the different configurations, a closer look at the internal processes responsible for building the index and therefore also the performance is necessary. As explained in Section 2, the internal structure of the PGM consists of different layers containing a decreasing amount of models until a small enough number of models is reached in the last layer constructed. This can result in varying numbers of models, depending on the data. For the lookup, those layers of models have to be traversed through, until the model responsible for the correct area in the data is found and

can be searched through. In order to illustrate these different models and layers, the best performing configuration of the PGM index of the dataset holding 1 million data points was analyzed in more detail, which will be laid out in the following paragraphs.
In the example of the small Dynatrace dataset, the choice of a maximum error term of 1, resulted in 2508 different models in 3 different layers. The bottom layer, which is the layer considered "closest" to the actual data, holds the most models, covering the whole range of data points and consists of 2495 different models, which is clearly the majority of all models in total. The second layer consists of 123 different models representing the bottom layer models, respectively, while the last layer holds only a single model, therefore building the root of this layer structure, which will be evaluated first.
Looking at the visual representations of each level, using each models' key and intercept in the place of the triple it stands for, the first level looks like the following Figure 4.11: The intercept in this graph was normalized to highlight the similarity to the original CDF graph seen before (Figure 4.8). Here the data points resemble the actual cumulative distribution pretty well, which makes sense considering that the error term in this model was chosen to be the smallest value possible, therefore making the space between the minimal keys also small.

The second level of the model abstracts the data distribution a lot more already, as seen in Figure 4.12. The data points here still span across the most of the range of keys, but flatten out most of the characteristic shapes of the CDF. The points seem to be more tightly coupled in areas where many duplicates are found (between the key values of 2500 and 5000) and more spaced out in all other areas. Eventually, these dots are again references by a single model, trying to represent all the keys in the second level. This single model would then be creating level 3 of the PGM Index.

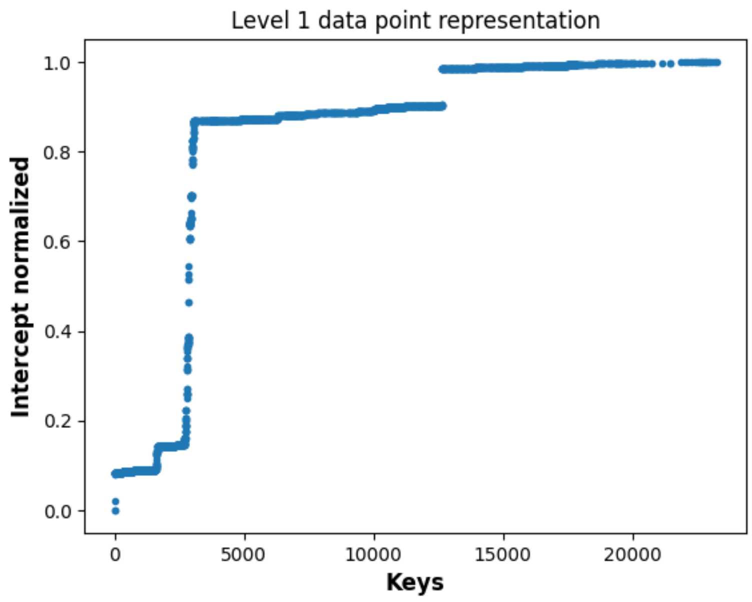


Figure 4.11: Representation of PGM triples in level 1

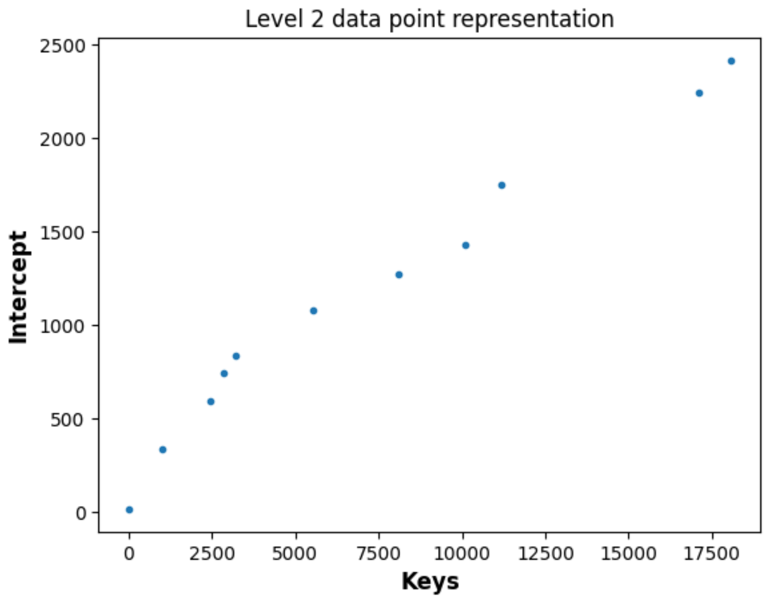


Figure 4.12: Representation of PGM triples in level 2

4.6 Results Synthetic Datasets

As previously mentioned, all experiments to compare lookup and build time were additionally run on synthetic datasets, which follow a known data distribution. This following section passes over the results, highlighting the most significant insights. As the most important results can be found in the lookup and build time experiments described in 3.4, which were already done for both Dynatrace datasets, the same experiments were also run on the synthetic datasets. The results regarding build time and lookup time is what the following paragraphs on this section will be focusing on.

4.6.1 Lookup Comparison Experiments & Build Time Comparison Experiments

Regarding building time, the results can be found in Figure 4.13. The trends for the 6 datasets are very much the same and also very similar to the results obtained for the Dynatrace dataset in the previous Section 4.2.

- The Recursive Model Index generally has the highest building time, due to its more complicated internal structure. For the configuration with the smaller memory footprint (around 10 MB), the building time seems to be very stable, followed by an upwards trend for models that go above this threshold of 10 MB. This can be seen in every one of the synthetic datasets used here. The maximum build time of the RMI therefore lays at roughly 1s, with most configurations landing at about half that value.
- Looking at the development of the B-Tree with an increasing index size, it is also visible that for every single dataset a bigger amount of memory used leads to an increased building time, ranging from 0.00005 to 0.1 s for building. The relationship between a larger model and the building time is stable, meaning that the configuration with the smallest memory footprint also provides the fastest build time.
- Comparing the RadixSpline building performance, it becomes clear, that the build time does not change for most datasets. Ranging around 0.001 s for building, it only provides configurations that perform significantly worse, when trying to fit a larger RadixSpline index to the Zipf dataset. Also, when fitting the RS index on the exponential and Right skews distribution, the range of build times across the different is not very large, meaning that even when trying to fit models with bigger error bounds, the model seems to end up with a similar amount of spline points.
- A similar trend can be seen for the PGM index, though it stays constant on all the different datasets, around the same time that the RS needs for building the models, around 0.001 s. On the datasets like exponential distribution and right skew, the range of PGM also does not seem to change a lot, which tells us, that at a certain size of the error term, the model does not produce significantly more models in the respective layers.

With the results of the building time reviewed, the next step will be to try and distinguish the performance of each of the index structures by the time it takes to find a random lookup key in the given data, taken on average after 1 000 000 lookups had been made. The results for that can be found in Figure 4.14, where the lookup time – measures in nanoseconds – is compared for the different index structures and their possible parameter configurations, represented by the different range sizes, all compared to the "baseline", which is the performance of a simple Binary Search on the data, to be seen as a black dashed line in each graph.

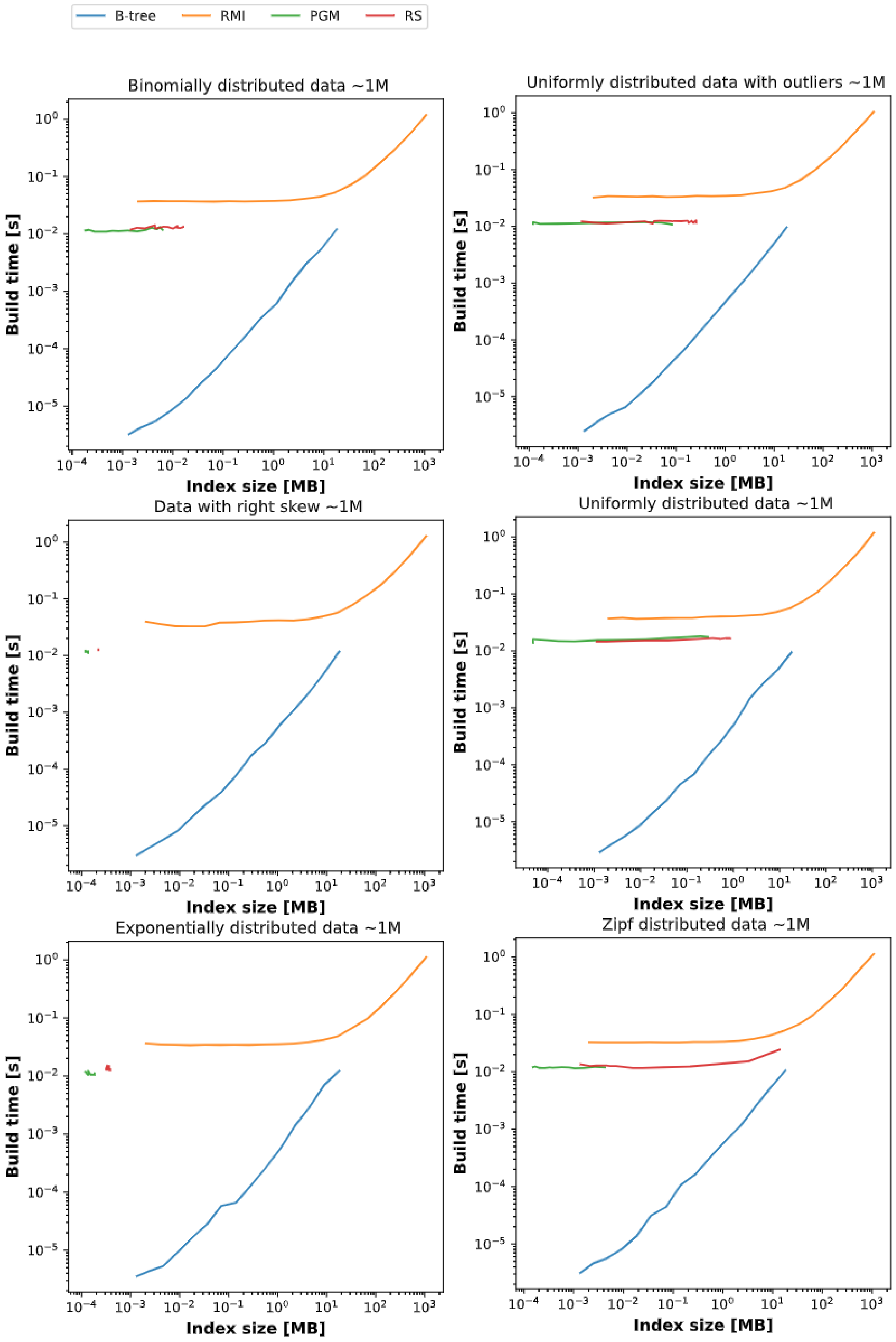


Figure 4.13: Comparison of build time for synthetic datasets

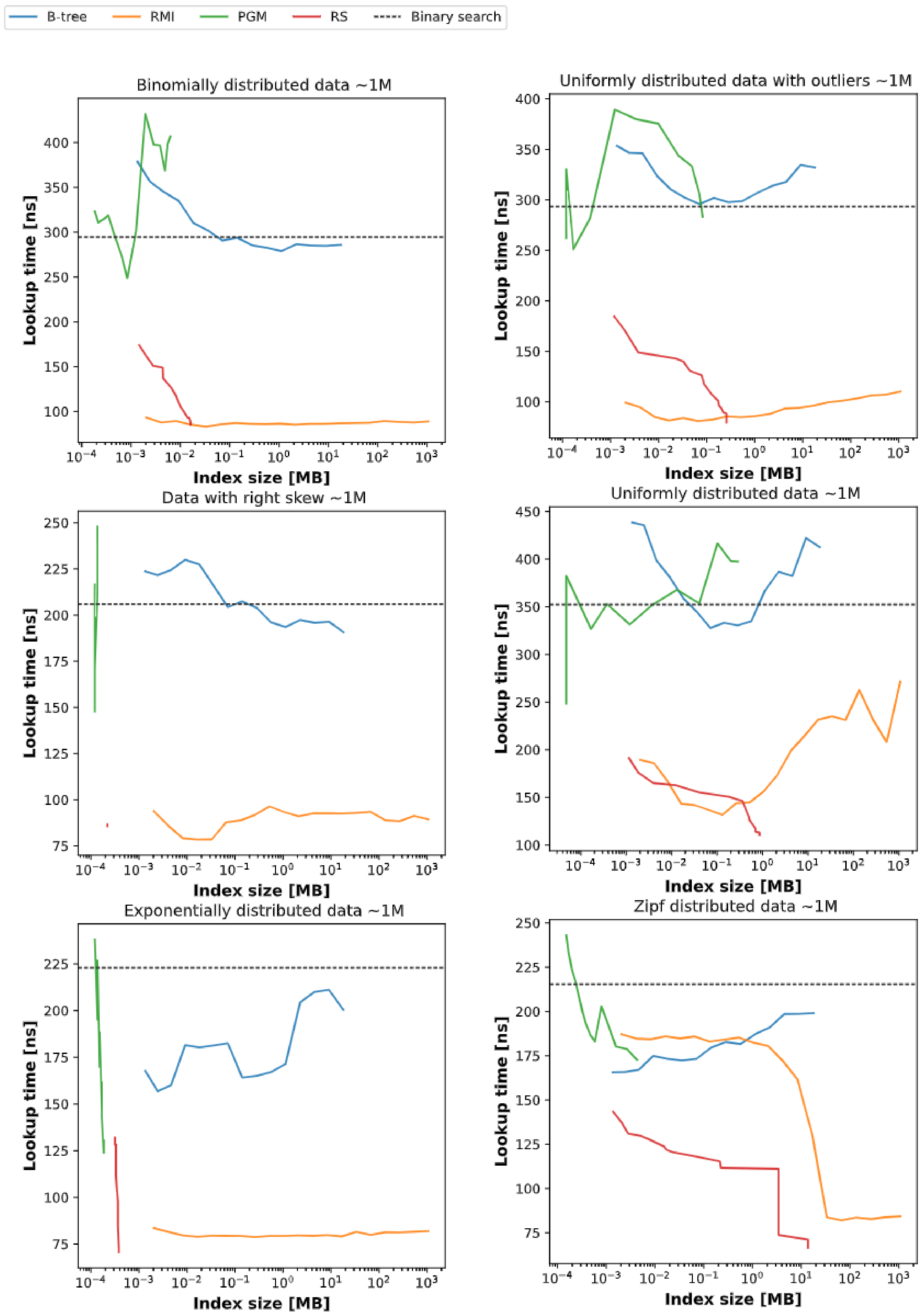


Figure 4.14: Comparison of lookup time for synthetic datasets

First off, a rough trend can be seen in all the datasets, though in detail they are all very different from each other. The performance of PGM and B-Tree is among the slowest in every dataset, with some variations of those two index structures even above a simple Binary Search (especially the smaller and therefore less accurate model sizes). The results in detail for each given dataset are explained in the next paragraphs.

- *Binomial distribution:* For this distribution, the cut between the better and worse performing index structures is very clear: RMI and RS are clearly the fastest among all different sizes, with RMI taking on average less than 100 ns, without it increasing significantly with different combinations of configuration. Second best seem to be the RadixSpline index, which gets a lot faster nearing the optimal size of 0.001 MB, where the performance is comparable to that of the RMI. For the PGM and B-Tree the slower lookup speed is clearly visible with most configurations taking around 400 ns, which is over 4 times slower than the faster Learned Index structures. Though the B-Tree seems to slightly improve its lookup performance when increasing the granularity and therefore size, the best cases are still taking around 3 times that of the fastest lookup performances.
- *Data with right skew:* Identically to what was seen in the graph for the building time, the size ranges of RS and PGM are a lot smaller than previously seen. The biggest differences between those two index structures here is that they seem to make up both ends of the spectrum when it comes to lookup time. While the RS gives a result within 85 ns, the fastest configuration of the PGM takes about twice as long, with the other variations performing a lot worse. Similarly to before, the RMI seems to provide very quick answers as well, although the size ranges are slightly over that of RadixSpline. The sweet spot for the best performance for this dataset is not in the bigger ranges of this index, but can be found at a mere size of 0.05 MB. The trend for the B-Tree also follows the one seen before, with a smaller lookup time of 186 ns per lookup, found in the variation using the biggest memory footprint.
- *Exponential distribution:* This distribution actually delivers some interesting results, Firstly, it offers a scenario that with similar sizes, the PGM and RS index seem to perform somewhat similar, with the PGM doing even slightly better. Secondly, the B-Tree shows behavior, where the lookup time actually increases, when using a larger model, which is the opposite of the trends seen before. Nevertheless, RMI, which again performs at a stable level around 80 ns, without a significant decrease or increase when choosing different hyperparameters. RadixSpline, while not offering a broad range of sizes this time around, shows a dramatic decrease of lookup time when the size is just slightly increased, providing us with the best performing index for this dataset.
- *Uniformly distributed data with outliers:* This dataset also shows the trend of the B-Tree and PGM index performing a lot worse than RMI and RS on the same dataset. While we find that the RS seems to have a slightly bigger memory footprint than in the other dataset, the increased size is responsible for the index performing as well as RMI, which is the best index structures over long stretches, even though the lookup time increases for the bigger variations of the RMI. Nevertheless, both RMI and RS can find a lookup key in this dataset in on average 90 ns, while B-Tree and PGM are mostly even slower than a lookup using Binary Search.
- *Uniform distribution:* When fitting the index structures on the relatively straightforward uniform distribution, we see the biggest performance difference yet. While being relatively calm for the bigger part of the previous synthetic datasets, using RMI on the uniformly distributed dataset, gives one of the slowest lookup times for the RMI, with a significant increase, the bigger the models seem to get. RadixSpline on the other hand, seems to not have as many problems with the data distribution, though the best performing configuration is also slightly more than previously seen, with 120 ns on average. The performances of PGM and B-Tree are very similar, with PGM providing a better performance in the smaller size range, though still significantly worse than RMI

or RS.

- *Zipf distribution:* Last but not least, the Zipf distribution and the respective performances were analyzed. Here, the unique case that PGM, B-Tree and RMI perform similarly for a while, whereas the RMI improves greatly when increasing the index size to more than 80 MB, when the lookup time drops from previously 180 ns on average to around 85 ns. RadixSpline handles this data distribution the best of each index structure, where a larger variety of the index performs the best overall, while clearly outperforming all the other indexes. B-Tree's performance decreases when the size of the index increases, which is also an interesting fact, though somewhat related to the performance on the exponential distribution.

To wrap up the result section for the synthetic datasets, a side by side comparison of each the fastest configuration of each index structure for each of the datasets is done, while simultaneously analyzing which parts of the lookup (evaluation of the models or last mile search) contributes the most time to the general lookup. The results of this analysis can be seen in Figure 4.15.

- *Binomial distribution:* The comparison of the fastest configurations basically confirm the trends already shown in Figure 4.14. RMI and RS are by far the fastest index structures, with the best lookup times just under 100ns. When evaluating the time spent for evaluation for each index structure (with Binary search having no evaluation time at all, as it is, as the name already gives away, a search only structure, the lookup time in total for RMI and RS are even below the evaluation time of both PGM and B-Tree. This means that the Recursive Model Index and the RadixSpline Index are finished with the lookup before PGM or B-Tree even evaluated the correct segment of data to look through. Additionally, we can see that, in the better performing index structures, have a rather small share of their lookup time dedicated to evaluating the models and most of the time is spent on the last mile search.
- *Data with right skew:* On the dataset with heavy right skew, the result looks similar to before, with RMI and RS performing the best, with a big portion of their lookup dedicated to looking through the last segment of data, that was evaluated by the model before. For this dataset, the difference between the best performing index structures and PGM and B-Tree is slightly smaller than previously seen, even though the ratio of search and evaluation time is still somewhat unchanged.
- *Exponential distribution:* For the exponentially distributed data, a similar trend can be seen, as PGM and B-Tree are also within a lookup time of 150 ns, with the share of the evaluation time being at 75 ns which is about the total lookup time for RS and RMI.
- *Uniformly distributed data with outliers:* When analyzing the dataset with outliers, the difference between RMI, RS and PGM as those are the three Learned Index structures is again very visible, with the best configuration of the PGM taking almost three times the amount of the best variants of RMI and RS. Solely 150 ns of the lookup time of the PGM can be attributed to the last mile search which, considering that the best performing PGM configuration is rather small (as can be seen in Figure 4.14), means that the models each have to be responsible for a bigger chunk of data,
- *Uniform distribution:* What is remarkable, comparing the best performing configurations of each index on a uniform distribution is that, even though the performance of RMI and RS is still unmatched, the evaluation time is low for each of the Learned Index structures, whereas it remains unchanged in the B-Tree, compared to other synthetic datasets. Additionally, the search time in RMI and RS is slightly higher than previously, which also increases the lookup time in total, with the best combination taking 125 ns for an average lookup.
- *Zipf distribution:* The results for the rather complex Zipf Distribution, looks almost identical to

previously seen results, with the Recursive Model Index and the RadixSpline being quick to evaluate their models, with RadixSpline seemingly handling the distribution best. B-Tree and PGM taking significantly longer for the lookup, is also in alignment with the previous results, even though the search time on the PGM configuration slightly shorter and therefore the evaluation time longer than in the other datasets. Taking into account the knowledge from Figure 4.14, we can see that the fastest PGM configuration is situated not in the smaller size ranges, as it is the case in most other datasets, but the fastest lookup time achieved with the largest PGM variation. As this implies that there are more models built, and possibly more layer to be traversed through in order to find the right segment of data, it makes sense that the evaluation time takes up a bigger share.

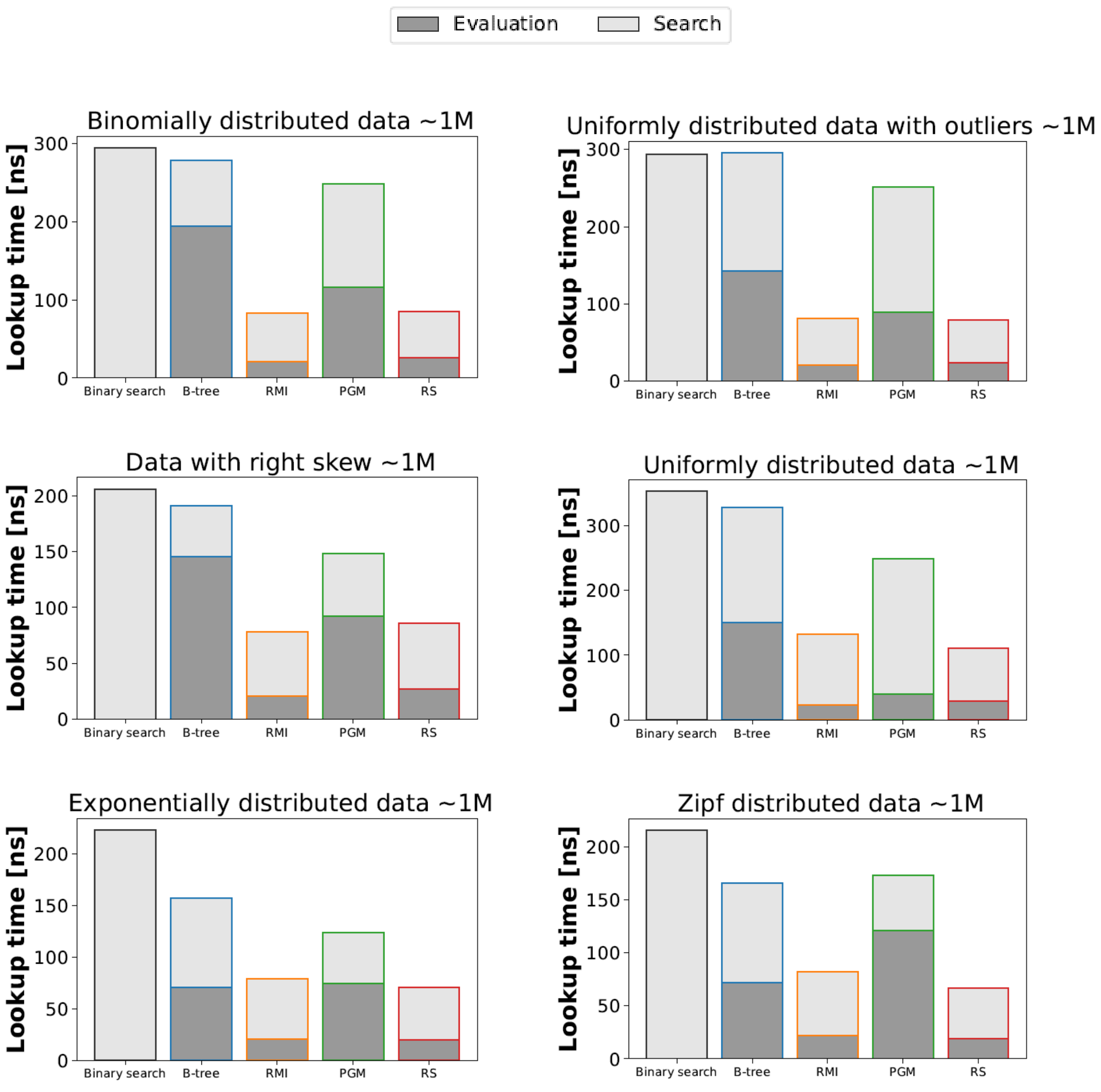


Figure 4.15: Comparison of best performing configuration for each index structure for synthetic datasets

4.7 Results Comparison to the Industry Standard

When comparing the results for lookups to systems that can generally be found in the industry, Apache Lucene, it soon became clear that using Learned Index approaches, speeds things up significantly. The performance of Apache Lucene, an open source Java library providing efficient indexing and search features. This system is not explicitly built for handling numeric lookups.

index	lookup in ms
B-tree	0.21339
Binary search	0.21279
Lucene	98700
PGM-index	0.37364
RMI-ours	0.07945
RadixSpline	0.10799

Table 4.7: Median of lookup times for the small Dynatrace dataset compared to Lucene

The results can be observed in Figure 4.16 and Figure 4.17 and clearly show that for both building and lookup the performance of Lucene is far worse compared to all the previously analyzed index structures. Building the Apache index on the given data already takes around 100 times longer than, though the performance stays within a tight bound, which is due to the fact that there are not any parameters which can be tunes to vary the build of the index.
When it comes to the performance of the lookup on the data, after having build the structures it soon becomes clear that, even though the range of the different lookup times is very large, Apache Lucene takes several hundred times longer in returning the position. This vast difference in lookup times can also be seen in Table 4.7, which confirms that, taken the median of the lookup times, Lucene can in no way compare to the efficient Learned Index structures.
These results come as a surprise, given that Lucene is per se able to deal with numbers as well, providing numeric field for these cases. The vast difference in performance however, can be explained in the way Lucene stores and accessed data. In Lucene values are stored as documents, which is a collection of fields, one of the fields in a document being a numeric field for the integer values to be indexed. With Lucene being an inverted index it stores which field values can be found in which document. During the lookup the index returns all documents, containing the wanted lookup key, in this case all records with the wanted lookup key. It is not possible to efficiently return only the first occurrence of a value, like we do with Learned index structures. This fact can explain why the lookup times for Lucene are generally much worse, as Lucene basically wants to do a more thorough job in searching for the wanted lookup key.

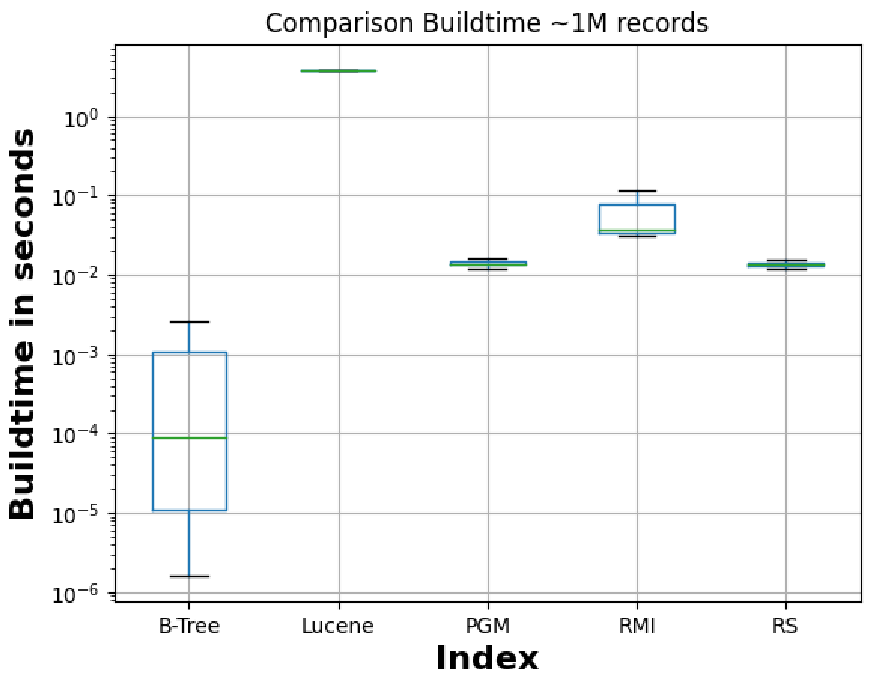


Figure 4.16: Performance of build times of Index structures compared to Lucene on the smaller Dynatrace dataset

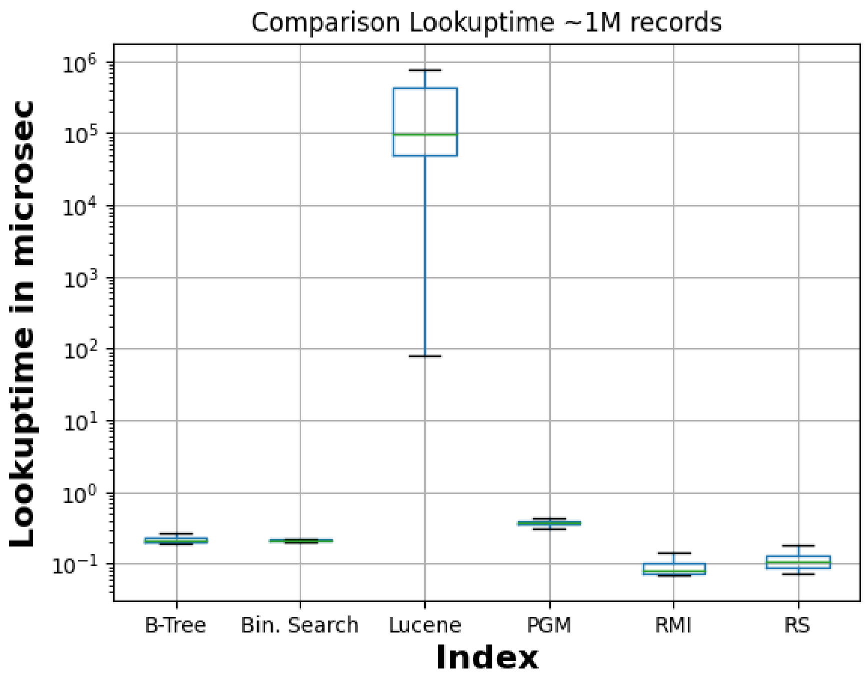


Figure 4.17: Performance of lookup times of Index structures compared to Lucene on the smaller Dynatrace dataset

5 Discussion

The goal of this thesis was to investigate what the potential of Learned Index structures compared to traditional index structures could be, based on a possible use case with Dynatrace. The thesis especially focused on three different approaches that all have been used in previous benchmarking work and therefore tested on real life data already. The main objective lay on the analysis of two different datasets that represented data used within Dynatrace, along with synthetic datasets. This way, the functionality of each index approach could thoroughly be investigated and possible trends in the Dynatrace dataset, validated by further experiments. These experiments consisted of an in-depth analysis of the lookup and build performance of various index structures, of both learned and traditional indexes and search algorithms. As each of those index approaches can be tuned to some degree, several different variations of the same index structure were applied to the same datasets to showcase the differences between different data distributions.

In a technical environment that is increasingly data-intensive, providing index structures to tame the sheer endless amounts of data available is a must. Depending on the requirements of the business case, there are different index approaches that are widely used nowadays. [10] This doesn't mean though that the index structures that have been around for a long time (like B-Trees) dominate structures that only now start to emerge. This is also one of the motivations for this thesis, that should be evaluated with this study. In order to fully understand what makes the idea of using Learned Index structures currently interesting, a broad theoretical foundation was established throughout this work. This was followed by a detailed description of the experimental setup and its limitations.

In their core, each of the Learned Index structures follow the same basic principle: utilize the Cumulative Distribution Function to efficiently estimate the position of a certain lookup key. The way this is done, differs between the different structures. The main Learned Indexes used for answering the research questions defined in Section 1.4 were 1. Recursive Model Index, a top down built index utilizing different Machine Learning models in a hierarchical structure to estimate the cumulative Distribution Function of a dataset; 2. RadixSpline, an index which can be built in a single pass across the data, creating and indexing spline points of said data; and 3. Piecewise Geometric Model Index, which, as the name suggests, uses exclusively piecewise linear regression to estimate the data to index.
While the RMI and PGM seem to be somewhat similar from a first look, the biggest difference between them (and ultimately between the results as well), is that while PGM gets built bottom up, without a fixed number of layers to adequately represent the data given data distribution, RMI is more restricted. With a top-down approach, previous work for the RMI has shown, that more than two layers are not solving the problem more efficiently and that the granularity of two layers is the sweet spot of the ideal lookup time. Ultimately, the RMI offers more freedom when choosing which models should represent the given data, even though the usage of simple models (like linear regression), comes with a significant advantage in lookup time. The experiments conducted in this thesis showed that for the Dynatrace datasets especially, but also used on the more regular synthetic datasets, the more lightweight structure of the RMI finds the needed key faster that its direct competitor PGM. As seen in Figure A.1 especially, all configurations of the RMI find a result, including the last mile search, before the PGM even found the estimated position in the data (excluding looking through the last stretch of the data, based on the found model).
Evaluating all results around the PGM Index, it is obvious that PGM struggles to keep up with the performance of the other Learned Index structures, which is visible in Figure 4.14 and Figure 4.2. In every instance, the PGM takes significantly longer than its learned counterparts, even under controlled conditions, while mostly keeping up with the performance of a B-Tree or Binary Search. In a more in-

depth analysis of the background of the PGM structure on a Dynatrace dataset with the size of around 1 million, which is irregularly distributed and with many outliers (see Figure 3.1), a possible reason can be identified: the PGM bottom-up- builds 3 layers of Piecewise linear regression referencing each other, meaning that during a lookup all those layers have to be evaluated, which intuitively takes longer than going through two-layers only (as can be seen when building the RMI). Additionally, the choice of error within a model of the PGM can also drastically influence the lookup time, as seen in Figure 4.10. A smaller model error, which means building more models to cover the same range of data than when choosing a bigger error value, yields results that take almost twice as long as the best performance. Compared to its fellow Learned Index approaches, its optimal lookup performance is still around three times as slow.

When discussing the top performers, lookup-wise, it quickly becomes apparent that the performance of RadixSpline and RMI are similar on a lot of different datasets. The more lightweight and adaptable structures of those two indexes, capture the given data distribution very good, as the lookup times are well below 100ns for most datasets, as can be seen in Figure 4.3 and Figure 4.15. Those two Learned Index structures clearly perform better as both PGM and B-Tree except for the Zipf dataset, where only the RadixSpline showing a clearly faster performance. Much of this performance plus comes at the hands of a faster evaluation step, which is what the findings in the previously-mentioned dataset suggest. Therefore, most of the total lookup time for RMI and RadixSpline can be attributed to the last mile search after the initial evaluation points to the correct part of the data. The RadixSpline achieves this fast performance with the right choice of error bounds between the spline points (relatively small) and the most efficient way of indexing the spline points themselves (with a high distinctiveness achieved by indexing a lot of significant bits in the radixtable). As these results not only hold up for both Dynatrace datasets, but also in the synthetic datasets, the findings strongly suggest that it is very realistic that using Learned Indexes, especially RMI and RadixSpline, can speed up the lookup performance when used on Dynatrace data.

The results for the lookup times has to be seen in context with the build times though. Experiments focusing on the time it takes to build a Learned Index and traditional ones on a given dataset, show that the building times for the more complex Learned Indexes are generally higher than their non-learned counterparts (see Figure 4.1 and Figure 4.13.) Seen in the context of where each best configuration lies, the difference is not as large as the first look might suggest, but still will have to be considered when comparing the two metrics.

Another fact that has to be considered when implementing the principles of the index structures mentioned in this thesis is the several limitations that come with each index. The fact that each index, as used in this context, does not support updates in the data, means that the data has to be read only upon starting to build the index. Also, the limitation of only using positive integer values is a significant downfall, which will have to be addressed in further work, as numeric columns in a database often contain a wider range of numbers and formats. As mentioned in Section 3 the data has to be sorted first in order to construct any Learned Index over it, another requirement that is often not given in a natural setting.

6 Conclusion and Future Work

While being a relatively new field of research, the usage of Machine Learning models within the field of indexing yielded promising results in the first pieces of research that surfaced. As with most other new research areas, multiple ways to incorporate those models in indexes followed the initial ideal published in "A Case of Learned Index Structures". Having multiple approaches to choose from, this thesis compares structures that have previously been tested and used in benchmarking work, such as the Recursive Model Index, RadixSpline index and the Piecewise Geometric Model Index.

In direct comparison with each other, this thesis explored the subtle differences of these Learned Indexes using the example of data used within Dynatrace, that is potentially interesting for indexing. It did so, experimenting with the different parameters that each index approach comes with as well as test and double check trends using synthetically produced datasets, that follow a certain data distribution. The two different Dynatrace datasets used are distributed in a manner that can not be described with any known data distribution, as it contains a great number of duplicates. As all the used index structures come with a certain amount of limitations and requirements to the data that they are built on, a great deal of data preparation was necessary in order to properly compare the given data.

Eventually, the results of the different available parameters and tuning possibilities, suggest that there can be significant differences regarding lookup- and build time when using Learned Index structures as opposed to more traditional approaches. It was shown that when using the same index size, Learned Index structures, though a bit slower to build due to their complexity, could provide faster lookup results. When randomly sampling 1 000 000 keys from the distribution of the given data and then looking for them using the index structures, especially RMI and RadixSpline provided results in a fraction of the time that other indexes might need. This holds true for both Dynatrace datasets and most synthetic datasets used. Especially remarkable is the fact, that they are not only faster for the best configuration possible but for most other parameter combinations, which suggests that for the same storage used, Learned Indexes like RadixSpline and RMI provide a more efficient and faster lookup performance. The lookup time of PGM index though, did not hold up to the standards set by the other two used structures. This was mostly due to the fact, that the internal structure of the PGM uses a larger hierarchy tree that would have to be traversed through in order to find the correct key.
Comparing these results to the findings of Dittrich and Maltry [13] as well as the other previous work in the area of Learned Index structures, the previously made findings are once again validated. In the example studied here, the considerably faster lookup time of Learned Index structures come with higher build time in every scenario investigated here. Also, this work once more underlines the potential Learned Index structures have, which is also in line with previous research work done.

This preliminary result sounds already very promising in a protected experimental setup, still there will have to be more tests conducted to make sure that this performance can also be achieved outside the experimental setup found in this thesis. Especially the limitation that the data has to be read only will most probably not hold up in a real productive environment. Therefore, an update to the indexes to make them updatable will be necessary for further work, a feature that now only exists for the PGM Index.
Another problem to solve in the future to make the indexes able to deal with data as it appears in the real world better, would be to look into the requirement that the data has to be sorted. With these problems solved, the Learned Index structures are hopefully truly able to take on data as it appears in the wild.

References

[1] Fatima Abdullah, Limei Peng, and Byungchul Tak. "A Survey of IoT Stream Query Execution Latency Optimization within Edge and Cloud". In: *Wireless Communications and Mobile Computing* 2021 (Nov. 2021), p. 4811018. ISSN: 1530-8669. URL: `https://doi.org/10.1155/2021/4811018`.

[2] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. `https://panthema.net/tlx`, retrieved Oct. 10, 2022. 2018.

[3] Andrew Crotty. "Hist-Tree: Those Who Ignore It Are Doomed to Learn". In: *Conference on Innovative Data Systems Research*. 2021.

[4] Jialin Ding et al. "ALEX: An Updatable Adaptive Learned Index". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2019).

[5] Dynatrace. *Dynatrace | Modern cloud done right*. `https://www.dynatrace.com/`. Accessed: 2023-12-12. 2023.

[6] 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. URL: `https://pgm.di.unipi.it`.

[7] Alex Galakatos et al. "FITing-Tree: A Data-aware Index Structure". In: *Proceedings of the 2019 International Conference on Management of Data* (2018).

[8] Ali Hadian and Thomas Heinis. "Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction". In: *International Conference on Extending Database Technology*. 2021.

[9] 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. URL: `https://doi.org/10.1145/3401071.3401659`.

[10] Martin Kleppmann. *Designing Data-Intensive Applications*. Beijing: O'Reilly, 2017. ISBN: 978-1-4493-7332-0. URL: `https://www.safaribooksonline.com/library/view/designing-data-intensive-applications/9781491903063/`.

[11] Tim Kraska et al. "The Case for Learned Index Structures". In: *CoRR* abs/1712.01208 (2017). arXiv: `1712.01208`. URL: `http://arxiv.org/abs/1712.01208`.

[12] L. Kuipers and H. Niederreiter. *Uniform Distribution of Sequences*. en. Google-Books-ID: mnY8LpyXHM0C. Courier Corporation, May 2012. ISBN: 978-0-486-14999-8.

[13] Marcel Maltry and Jens Dittrich. *A Critical Analysis of Recursive Model Indexes*. 2021. arXiv: `2106.16166 [cs.DB]`.

[14] Ryan Marcus et al. "Benchmarking learned indexes". In: *Proceedings of the VLDB Endowment* 14.1 (Sept. 2020), pp. 1–13. URL: `https://doi.org/10.14778%2F3421424.3421425`.

[15] Thomas Neumann. *Database Architects: The Case for B-Tree Index Structures*. Dec. 2017. URL: `http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html` (visited on May 23, 2023).

[16] Pratiksha Thaker Peter Bailis Kai Sheng Tai and Matei Zaharia. *Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes · Stanford DAWN*. en. Jan. 2018. URL: `https://dawn.cs.stanford.edu/2018/01/11/index-baselines/` (visited on May 23, 2023).

[17] Jiacheng Wu et al. “Updatable Learned Index with Precise Positions”. In: *ArXiv* abs/2104.05520 (2021).

[18] Jiaoyi Zhang and Yihan Gao. “CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm”. In: *Proc. VLDB Endow.* 15 (2021), pp. 2679–2691.

[19] George Kingsley Zipf. “Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology”. In: *Science* 110.2868 (1949), pp. 669–669. eprint: `https://www.science.org/doi/pdf/10.1126/science.110.2868.669`. URL: `https://www.science.org/doi/abs/10.1126/science.110.2868.669`.

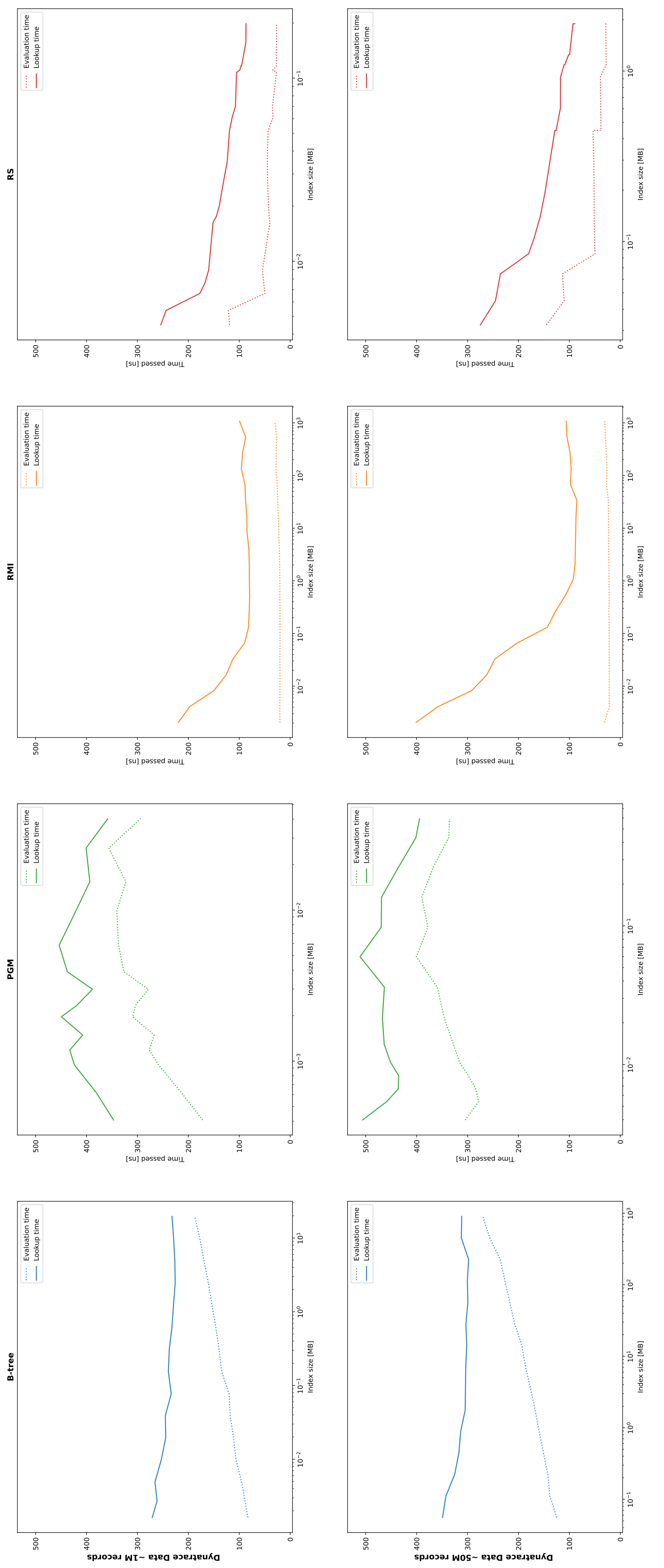


Figure A.1: Shares of evaluation time and search time on all configurations

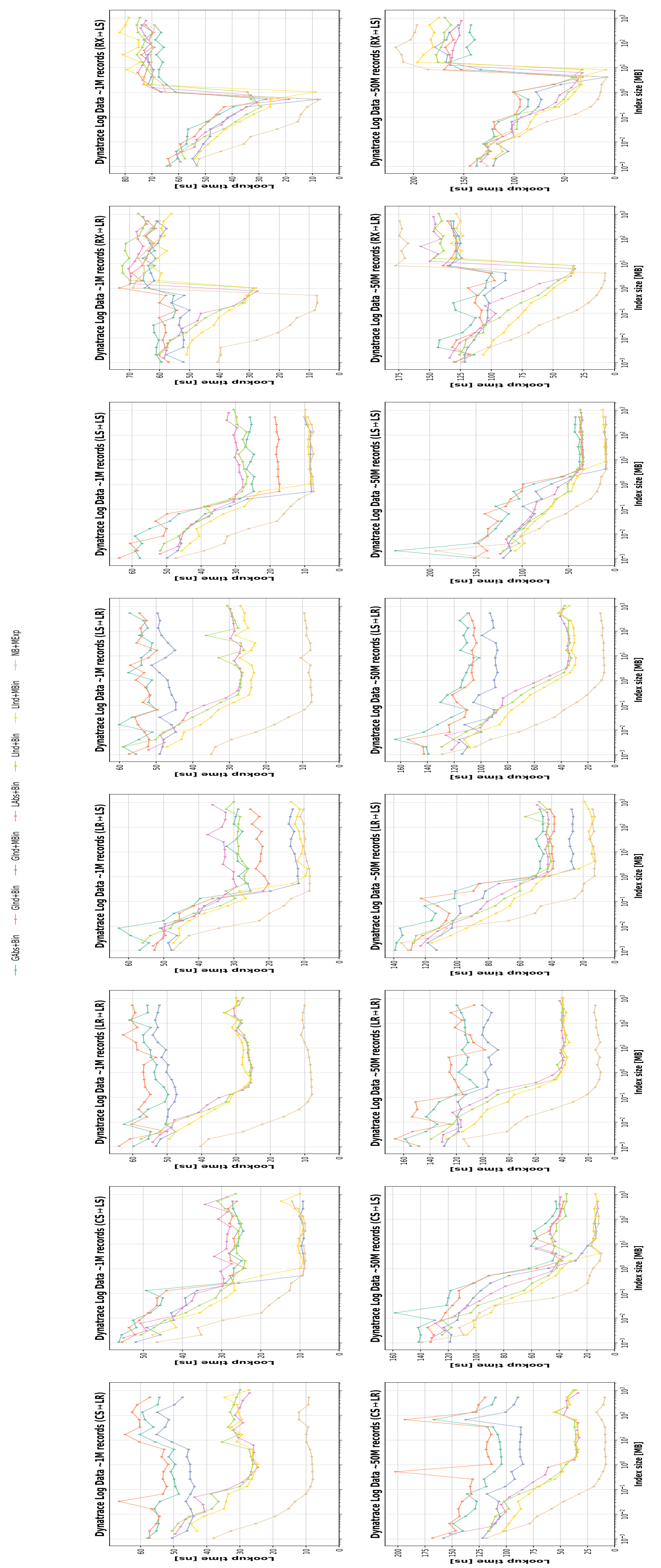


Figure A.2: Effect of different combinations of RMI hyperparameters of lookup performance- full picture