RESULTS reproduced

A Critical Analysis of Recursive Model Indexes

Marcel Maltry
Saarland University
Saarland Informatics Campus
marcel.maltry@bigdata.uni-saarland.de

Jens Dittrich
Saarland University
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

ABSTRACT

The *recursive model index* (RMI) has recently been introduced as a machine-learned replacement for traditional indexes over sorted data, achieving remarkably fast lookups. Follow-up work focused on explaining RMI's performance and automatically configuring RMIs through enumeration. Unfortunately, configuring RMIs involves setting several hyperparameters, the enumeration of which is often too time-consuming in practice. Therefore, in this work, we conduct the first inventor-independent broad analysis of RMIs with the goal of understanding the impact of each hyperparameter on performance. In particular, we show that in addition to model types and layer size, error bounds and search algorithms must be considered to achieve the best possible performance. Based on our findings, we develop a simple-to-follow guideline for configuring RMIs. We evaluate our guideline by comparing the resulting RMIs with a number of state-of-the-art indexes, both learned and traditional. We show that our simple guideline is sufficient to achieve competitive performance with other learned indexes and RMIs whose configuration was determined using an expensive enumeration procedure. In addition, while carefully reimplementing RMIs, we are able to improve the build time by 2.5x to 6.3x.

PVLDB Reference Format:
Marcel Maltry and Jens Dittrich. A Critical Analysis of Recursive Model Indexes. PVLDB, 15(5): 1079 - 1091, 2022.
doi:10.14778/3510397.3510405

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BigDataAnalyticsGroup/analysis-rmi.

1 INTRODUCTION

Machine learning and artificial intelligence are taking the world by storm. Research areas that were believed to have been researched to completion have been revisited with exciting new results, showing that considerable improvements are still possible *if* we factor in wisdom from the machine learning world. Notable examples include natural language processing and compute vision which were completely revolutionized in the past decade by variants of deep learning. In the database world, we witnessed a surge of similar re-exploration endeavors in the past five years. Notable recent examples of works in that space include cardinality estimation [11, 29], auto-tuning [1, 25], and indexing [7, 9, 10, 15, 18]. We believe that indexing is the most surprising result of these three areas because both cardinality estimation and auto-tuning are optimization problems and thus have a natural proximity to machine learning. The connection to indexing becomes evident when we examine a special case of indexing.

Problem Statement. Given a sorted, densely packed array A of keys and a query Q asking for a particular key x_i that may or may not exist in that array, return the array index i of that key x_i.

In other words, we are looking for a function that assigns to each key its position in the sorted array. Traditionally, this function is implemented by a suitable algorithm like binary search or a data structure like a B-tree. In contrast, Kraska et al. [18] observe that this function can be learned through regression, effectively making the indexing problem a machine learning task. Based on this observation, Kraska et al. [18] present the *recursive model index* (RMI) as first *learned index* with remarkable results in terms of lookup performance. We wanted to understand the performance benefits of RMIs early on and therefore tried to reproduce the results. However, we quickly encountered several issues.

Hyperparameter configuration. Configuring RMIs involves setting several hyperparameters. Unfortunately, the exact configurations with which the remarkable results were obtained were not reported and in some cases even described misleadingly. The use of neural networks is mentioned frequently throughout the experimental evaluation of the original paper. However, the low model evaluation times reported in Fig. 4 strongly suggest that none of the best-performing configurations actually uses neural networks. In personal communication with the first author in August 2019, we learned that linear models should be preferred over neural networks in most cases. In our experience, there is still a misconception in the community today that RMIs internally use neural networks. Subsequent studies [14, 22] involving inventors of the RMI investigated the performance benefits of learned indexes over traditional indexes. However, hyperparameter configurations for the reported results were obtained by a time-consuming enumeration process [23]. As a result, similar to the original paper [18], the studies neither show how the choice of hyperparameters affects performance, nor do they give advice for configuring RMIs in practice besides enumeration.

Closed source. The source code of the original paper was never made available. A so-called reference implementation [22] which differs from the descriptions in the original paper (see Section 3.2) was published in December 2019, two years after the preprint [17].

Goals: We pursue the following objectives with this paper.

(1) Conduct the first inventor-independent detailed analysis of RMIs to understand the impact of each hyperparameter on prediction accuracy, lookup time, and build time.

(2) Develop a clear and simple guideline for database architects on how to configure RMIs with good lookup performance.

(3) Provide a clean and easily extensible implementation of RMIs.

Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510405

Contributions: We make the following contributions to achieve these goals:

(1) Learned Tree-Structured Indexes. We revisit in detail recursive model indexes [18] and explain how they are trained and what hyperparameters to consider (Section 2). We provide a detailed overview on the design dimensions of learned indexes and the already large body of work in that space (Section 3).

(2) Hyperparameter Analysis. We present our experimental setup (Section 4) and conduct a set of extensive experiments to analyze the impact of each hyperparameter on predictive accuracy and search interval size (Section 5), lookup performance (Section 6), and build time (Section 7).

(3) Configuration Guideline. Based on our findings, we develop a simple guideline to configure RMIs in practice (Section 8).

(4) Comparison with Other Indexes. We compare the RMIs resulting from our guideline in terms of lookup time and build time with a number of learned indexes like ALEX [7], PGM-index [9], RadixSpline [15], and the reference implementation of RMIs [23], as well as state-of-the-art traditional indexes like B-tree [3], ART [19], and Hist-Tree [5] (Section 9).

2 RECURSIVE MODEL INDEXES

In this section, we recap recursive model indexes, how to perform a lookup, how they are trained, and what their hyperparameters are.

2.1 Core Idea

RMIs are based on the observation that the position of a key in a sorted array can be computed using the *cumulative distribution function* (CDF) of the data. Let D be a dataset consisting of $n = |D|$ keys. Further, let X be a random variable that takes each key's value with equal probability and let F_X be the CDF of X. Then, the position i of each key $x_i \in D$ in the sorted array is computed as:

$$i = F_X(x_i) \cdot n = P(X \leq x_i) \cdot n \tag{1}$$

Note that in the context of learned indexes, the term CDF is frequently used synonymously for a mapping from key to position in the sorted array instead of its statistical definition of a mapping from key to the probability that a random variable will take a value less than or equal to that key. In the following, we submit to the former interpretation.

The core idea of an RMI is to *approximate* the CDF of a dataset by means of a hierarchical, multi-layer model. Consider Figure 1 for an example three-layer RMI. Each model in an RMI approximates a segment of the CDF, all models of a layer together approximate the entire CDF. An RMI is a *directed acyclic graph* (DAG), i.e., in contrast to a tree, a node (or model) in an RMI may have multiple direct predecessors. We denote the i-th layer of a k-layered RMI by l_i where $0 \leq i \leq k-1$ and refer to the j-th model of the i-th layer by f_i^j. The first layer l_0 of an RMI always consists of a single *root model* f_0^0. Each subsequent layer may consist of an arbitrary number of models. The number of models of a layer l_i is denoted by $|l_i|$ and called the size of the layer.

2.2 Index Lookup

A lookup is performed in two steps: (1) *Prediction*: We evaluate the RMI on a given key yielding a position estimate. (2) *Error correction*: We search the key in the area around the estimated position in the sorted array to compensate for estimation errors. We discuss both steps in more detail below.

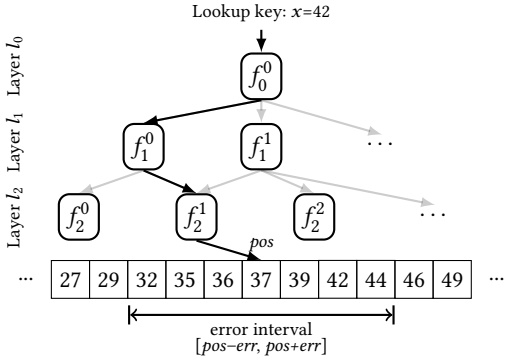


Figure 1: A three-layer RMI that is evaluated on key 42 yielding estimated position *pos* (prediction). Based on *pos*, the sorted array is searched for the key (error correction).

Prediction. Consider again Figure 1 that shows an index lookup for key 42. We start by evaluating the root model f_0^0 on key 42 yielding a position estimate. Based on this estimate, model f_1^0 in the next layer l_1 is chosen for evaluation. This iterative process is continued until the position estimate *pos* of the last layer is obtained.

Definition (Prediction): Let R be a k-layer RMI trained on dataset D consisting of $n = |D|$ keys. Let us denote the value p restricted to the interval $[a, b]$ by

$$[\![p]\!]_a^b := \max\big(a, \min(p, b)\big). \tag{2}$$

The predicted position for key x of layer l_i is recursively defined as

$$f_i(x) = \begin{cases} f_0^0(x) & i = 0 \\ f_i^{\left\lfloor [\![|l_i| \cdot f_{i-1}(x)/n]\!]_0^{|l_i|-1} \right\rfloor}(x) & 0 < i < k \end{cases} \tag{3}$$

Intuitively, to determine the model in layer l_i that is evaluated on key x, the estimate $f_{i-1}(x)$ of the previous layer is scaled to the size of the current layer. Note that $f_{i-1}(x)$ might be less than 0 or greater than $n-1$. Thus, the result is restricted to $[0, |l_i| - 1]$ to evaluate to a valid model index. The predicted position for key x of RMI R is the output of layer l_{k-1}:

$$R(x) = f_{k-1}(x). \tag{4}$$

Error correction. Based on the estimate $R(x)$ obtained by evaluating the RMI, the sorted array is searched for the key. In Figure 1, the position estimate for key 42 points to key 37 in the sorted array. Since 37<42, we have to search to the right of 37 to find 42. To facilitate the search, an RMI may store error bounds that limit the size of the interval that has to be searched. The RMI guarantees that if a key is present, then it can be found within the provided error bounds. A simple way of achieving this is to store the maximum absolute error *err* of the RMI. The left and right search bounds, i.e., the error interval, is set to $[R(x) - err, R(x) + err]$. If the key exists, it must be within these bounds. We search the interval for key x using an appropriate algorithm like binary search.

Listing 1 RMI Training Algorithm.

Input: Dataset D, number of layers k, array of layer sizes l
Output: RMI R

```
1:  procedure BuildRMI(D, k, l)
2:      R := Array2D()                          ▷Initialize dynamic array to store models.
3:      keys := Array2D()                       ▷Initialize dynamic array to store each model's keys.
4:      keys[0, 0] := D                         ▷Assign all keys to the root model.
5:      for i ← 0 to k − 1 do
6:          for j ← 0 to l[i] − 1 do
7:              R[i, j] := TrainModel(keys[i, j])      ▷Train model j of layer i.
8:              if i < k − 1 then                      ▷Check whether current layer is not last layer.
9:                  for all x in keys[i, j] do
10:                     p := GetModelIndex(x, R[i, j], l[i + 1], |D|)
11:                     keys[i + 1, p].add(x)          ▷Assign key x to next-layer model p.
12:     return R

13: function GetModelIndex(x, f, q, n)
14:     return ⌊⟦q · f(x)/n⟧_0^{q−1}⌋          ▷Compute model index according to Equation (3).
```

2.3 Training Algorithm

The goal of the training process is to minimize the prediction error. The training algorithm is shown in Listing 1. Its core idea is to perform a top-down layer-wise bulk loading. We start by assigning all keys to the root model (line 4). Then, the root model is trained on those keys (line 7). Afterwards, the keys are assigned to the next-layer models based on the root model's estimates (lines 9–11). We proceed by training the models of the next layer on the keys that were assigned to them. This process is repeated for each layer until the last layer has been trained. Finally, if desired, error bounds can be computed on the trained RMI (after line 11).

2.4 Hyperparameters

RMIs offer a high degree of freedom in configuration and tuning. In the following, we briefly describe each hyperparameter. We provide a set of possible configurations for each parameter in Section 4.2 when describing the experimental setup.

Model types. Model types are crucial to the predictive quality of RMIs. While simple models, e.g., linear regression, are small and fast to train and evaluate, complex models, e.g., neural networks, might offer higher accuracy, but are slow to train and evaluate.

Layer count. The number of layers k determines the depth of an RMI. While a deeper RMI might distribute the keys more evenly over the last-layer models, deeper RMIs are larger in size and take longer to train and evaluate.

Layer sizes. The size of a layer defines the number of models in that layer. A higher number of models leads to more accurate predictions since the segments that the models have to cover are smaller.

Error bounds. Error bounds facilitate the error correction by limiting the size of the interval that has to be searched. Error bounds can be chosen on different granularities or be omitted altogether.

Search algorithm. Depending on the error bounds, several search algorithms may be applied to perform error correction, e.g., binary search, linear search, or exponential search.

3 RELATED WORK

The introduction of learned indexes by Kraska et al. [18] caused both excitement and criticism within the database community. Early criticism mainly focused on the lack of efficient updates, the relatively weak baselines, and the absence of an open-source implementation [2, 24]. Later, Crotty [5] claimed that the performance advantages of learned indexes are primarily due to implicit assumptions on the data such as sortedness and immutability. Subsequently published learned indexes addressed some of these weaknesses [7, 9, 10]. Nevertheless, RMI remains one of the fastest indexes in experimental evaluations [5, 14, 15, 22].

3.1 Learned Indexes

Existing learned indexes commonly approximate the CDF. These indexes most notably differ in (1) the type of model they use to approximate the CDF, (2) whether they are trained bottom-up or top-down, and (3) whether they support updates.

FITing-tree. FITing-tree [10] models the CDF using *piecewise linear approximation* (PLA). During training, a dataset is first divided into variable-sized segments by a greedy algorithm in a single pass over the data. The segments are created in such a way that their linear approximation satisfies a user-defined error bound. Segments are then indexed by bulk loading them into a B-tree. A lookup consists of traversing the B-tree to find the segment that contains the key, computing an estimated position based on the linear approximation of the segment, and searching the key within the error bounds around the estimated position. FITing-tree supports inserts, either in-place by shifting existing keys within the segment or using a buffering strategy, where each segment has a buffer that is merged with the other keys in the segment whenever the buffer is full. Unfortunately, at the time of writing, no open-source implementation of FITing-tree was available which kept us from including it in our experiments.

ALEX. ALEX [7] uses a variable-depth tree structure to approximate the CDF with linear models. Internal nodes are linear models which, given a key, determine the child node. Leaf nodes hold the data, the distribution of which is again approximated by a linear model. During a lookup the tree is traversed until a leaf node is reached, then a position is predicted using the leaf's linear model, and finally, the key is searched using exponential search. Like RMI, ALEX is trained top-down, however, ALEX has a dynamic structure that is controlled by a cost model, which decides how to split nodes. ALEX supports inserts by splitting or expanding full nodes.

PGM-index. PGM-index [9] also approximates the CDF by means of PLA. Similar to FITing-tree, PGM-index starts by computing segments that satisfy an error bound. However, in contrast to FITing-Tree, PGM-index creates a PLA-model that is optimal in the number of segments. Each segment is represented by the smallest key in that segment and a linear function that approximates the segment. Afterwards, this process is continued recursively bottom-up by again creating a PLA-model on the smallest keys of each segment. The recursion is terminated as soon as a single segment is left. So unlike ALEX, each path from the root model to a segment is of equal length. A lookup is an iterative process where on each level of the PGM-index (1) a linear model predicts the next-layer segment containing the key, (2) the correct segment is searched within the error bounds around the prediction using binary search, and (3) the process is continued for the next-layer segment until the sorted array of keys is reached. Ferragina and Vinciguerra [9] also introduce variants of PGM-index that support updates (dynamic

PGM-index) and compression on the segment level (compressed PGM-index). The size of PGM-index depends on the number of segments required to satisfy the used-defined error bound.

RadixSpline. In contrast to the aforementioned learned indexes, RadixSpline [15] approximates the CDF using a linear spline. The linear spline is fit in a single pass over the data and to satisfy a user-defined error bound. The resulting spline points are inserted into a radix table that maps keys to the smallest spline point with the same prefix. The size of the radix table depends on the user-defined prefix length. A lookup consists of finding the spline points surrounding the lookup key using the radix table, performing linear interpolation between the spline points to obtain an estimated position, and applying binary search in the error interval around the estimated position to find the key. Like RMI, RadixSpline has a fixed number of layers and does not support updates.

3.2 Experiments and Analysis

Marcus et al. [23] published an open-source implementation of RMIs along with an automatic optimizer in December 2019. The reference implementation differs in some respects from the original description [18]. For instance, model types like B-tree nodes and neural networks are missing and error bounds are determined on a different granularity. Given a dataset, the optimizer uses exhaustive enumeration to determine a set of pareto-optimal (in terms of lookup time and index size) two-layer RMI configurations consisting of first-layer model type, second-layer model type, and second-layer size. Instead of blindly performing this costly enumeration, our work aims to understand the impact of each hyperparameter and to develop a simple guideline. Further, in addition to model types and layer sizes, we also consider error bounds and search algorithms when configuring RMIs.

Kipf et al. [14] introduced the *Search On Sorted Data* (SOSD) benchmark, a benchmarking framework for learned indexes. Besides providing a variety of index implementations, they supply four real-world datasets. In their preliminary analysis, the authors conclude that RMI and RadixSpline are able to outperform traditional indexes including ART [19], FAST [13], and B-trees while being significantly smaller in size. The authors also state that the lack of efficient updates, long building times, and the need for hyperparameter tuning are notable drawbacks of learned indexes.

As a follow-up, Marcus et al. [22] conduct a more detailed experimental analysis of learned indexes based on the framework and datasets from SOSD [14]. The authors perform a series of experiments to explain the superior performance of learned indexes and conclude that a combination of fewer cache misses, branch misses, and instructions account for most of the improved performance compared to traditional indexes. Further, the authors show that learned indexes are pareto-optimal in terms of size and lookup performance independently of dataset and key size.

Both aforementioned studies [14, 22] involve inventors of the RMI and aim to explain the performance of learned indexes in general. Since the evaluated RMI configurations were obtained using the optimizer [23], the studies neither show the impact of incorrectly configuring an RMI, nor do the studies provide advice on how to configure RMIs outside of using the optimizer. In contrast, to the best of our knowledge, we conduct the first independent and holistic analysis of RMIs that directly compares configurations and aims to explain their performance.

Ferragina et al. [8] take a theoretical approach at understanding the benefits of learned indexes, specifically of indexes based on PLA. The authors show that for a number of distributions, PGM-index [9], while achieving the same query time complexity as B-trees, offers improved space complexity. To support their theoretical results, the authors conduct several experiments both on synthetic and real-world datasets. The theoretical results build a solid foundation for further research. However, since RMIs are neither limited to PLA nor do RMIs aim to construct the optimal number of segments, the results cannot be transferred to RMIs.

4 EXPERIMENTAL SETUP

In this section, we introduce the implementation, hyperparameters, datasets, and workload used in our experiments and baselines considered for comparison. All experiments are conducted on a Linux machine with an Intel® Xeon® CPU E5-2620 v4 (2.10 GHz, 20 MiB L3) and 4x8 GiB DDR4 RAM. Our code is compiled with `clang-12.0.1`, optimization level `-O2`, and executed single-threaded.

4.1 Implementation

Our implementation of RMIs is written in C++. RMI classes have a fixed number of layers and model types are passed as template arguments. This implies that all models in a layer are of the same type. Training algorithms of the model types are adapted from the reference implementation [23]. When assigning keys to the next-layer models, the reference implementation always copies keys to a new array. We optimized the training process based on the observation that the models considered here are monotonic and will never create overlapping segments. Thus, when assigning keys to next-layer models, we simply store iterators on the sorted array of the first and last key of each segment. We then train the next-layer models by passing them the respective iterators and thereby avoid copying the keys. Further, instead of training all models on a mapping from key to position in the sorted array, we train inner layers on a mapping from key to next-layer model index which is obtained by scaling the position to the size of the next layer similar to Equation (3). In other words, we train inner layers directly on a targeted equal-width segmentation. This approach saves a multiplication and division during lookup that are otherwise required for computing the model index from the position estimate.

4.2 Hyperparameters

In the following, we give a list of hyperparameter configurations evaluated in our experiments and briefly compare them against those considered by the reference implementation's optimizer [23].

Model types. Table 1a lists the model types considered in our evaluation. Linear regression (LR) is a linear model that minimizes the mean squared error (MSE). Linear spline (LS) and cubic spline (CS) fit a linear respectively cubic spline segment through the leftmost and rightmost data points. Radix (RX) eliminates the common prefix and maps keys to their most significant bits. Models most notably differ in three respects.

(1) *Built time.* LS, CS, and RX are fast to build from the leftmost and rightmost key. LR, a regression method, is built on all keys.

Table 1: Evaluated hyperparameter configurations.

(a) Model types

Abrv.	Method	Formula
LR	Linear Regression	$f(x) = ax + b$
LS	Linear Spline	$f(x) = ax + b$
CS	Cubic Spline	$f(x) = ax^3 + bx^2 + cx + d$
RX	Radix	$f(x) = (x \ll a) \gg b$

(b) Error bounds

Abrv.	Method	Granularity
LInd	Local Individual [18]	max +/- error per model
LAbs	Local Absolute [23]	max abs error per model
GInd	Global Individual	max +/- error per RMI
GAbs	Global Absolute	max abs error per RMI
NB	No Bounds [18]	-

(c) Search algorithms

Abrv.	Method
Bin	Binary Search
MBin	Model-biased Binary Search [18]
MLin	Model-biased Linear Search
MExp	Model-biased Exponential Search [18]

(2) *Evaluation time.* RX is the fastest to evaluate with only two bit shifts. LR and LS are equally fast to evaluate, CS is the slowest.
(3) *Predictive quality.* LS and CS are spline techniques whose predictive quality is based on how representative the leftmost and rightmost keys are. LR minimizes the error across all keys. RX is radix-based and therefore only used for segmentation.

In addition to the four models listed, the optimizer [23] considers radix tables and a specialized variant of linear regression (see Section 9.1) for the first layer and cubic splines for the second layer. We decided to evaluate a smaller set of model types to analyze the impact of model types in general. Since the optimizer always recommends LR for the second layer, we only consider LR and LS for the second layer.

Layer count. Like the optimizer [23], we only consider two-layer RMIs. It was previously reported that in most cases two layers are sufficient to accurately approximate a CDF [22, 23], which we verified for the considered datasets in preliminary experiments. We plan to explore multi-layer RMIs as part of future work.

Layer size. We cover the same wide range of second layer sizes between 2^6 and 2^{25} in power of two steps like the optimizer [23].

Error bounds. We consider five different variants of error bounds listed in Table 1b, which differ in the granularity of the stored bounds in two respects. (1) Error bounds might either be computed for each last-layer model (local) or for the entire RMI (global). Global bounds, while being more memory efficient, are prone to outliers as the single largest error determines the search interval size of all lookups. Local bounds are more robust against outliers as an outlier only affects the respective model. (2) We can either store the maximum absolute error (absolute) or both the maximum positive and negative error individually (individual). While the former is again more space efficient, the latter allows for tighter bounds, especially, if a model either overestimates or underestimates the actual position. Additionally, we might not store any bounds (NB). Both local individual (LInd) and NB were suggested by Kraska et al. [18]. The reference implementation supports local absolute bounds (LAbs) and NB, but the optimizer [23] always recommends LAbs.

Search algorithm. The evaluated search algorithms are listed in Table 1c. We generally distinguish between two types of search algorithms: (1) search algorithms that only consider the error bounds and (2) search algorithms that also utilize the estimated position (model-biased) [18]. Standard binary search is an example of the first type of search algorithm. We search the key in the interval between the two error bounds and ignore the position estimate. However, binary search can be adjusted to become model-biased. Instead of choosing the middle element of the interval as first comparison point, we pick the estimated position. Similarly, linear search and exponential search can be tweaked to become model-biased. Instead of searching the interval from left to right, we start the search from the estimated position and search to the left or right, depending on whether the prediction is an overestimation or an underestimation. The search is stopped once it is certain that the key cannot be found anymore. Initially, we also considered standard linear search and exponential search for our experiments but both always performed worse than their model-biased counterparts. Note that not all combinations of error bounds and search algorithms make sense, e.g., in the case of absolute error bounds, model-biased binary search and standard binary search are essentially the same as the estimate will be the center of the interval. Further, model-biased linear and exponential search do not require bounds. Previous studies compared binary [14, 18, 22], linear, and interpolation search [22]. Model-biased variants of linear and exponential search have not been studied in the context of RMIs so far.

4.3 Datasets

Learned indexes are know to adapt well to artificial data sampled from statistical distributions [22]. Therefore, we use the four real-world datasets from the SOSD benchmark [14]. Each dataset consists of 200M 64-bit unsigned integer keys. The CDFs of the four datasets are depicted in Figure 2, zoom-ins show a segment of 100 consecutive keys and indicate the amount of noise in the dataset.

books: keys represent the popularity of books on Amazon.
fb: keys represent Facebook user ID. This dataset contains a small number of extreme outliers, which are several orders of magnitude larger than the rest of the keys, at the upper end of the key space. These outliers were not plotted in previous studies [14, 22].
osmc: keys represent cell IDs on OpenStreetMap. This dataset has clusters that are artifacts of projecting two-dimensional data into one-dimensional space [22].
wiki: keys are edit timestamps on Wikipedia, contains duplicates.

4.4 Workload

For the lookup performance, we consider lower bound queries, i.e., for a given key, the index returns an iterator to the smallest element in the sorted array that is equal to or greater than the

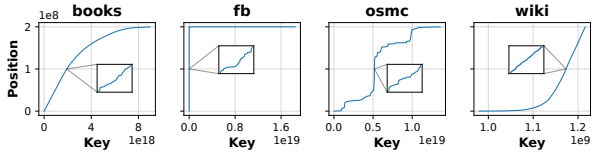


Figure 2: CDFs of four real-world datasets from SOSD [14]. Zoom-ins show segments of 100 consecutive keys.

Table 2: Overview of the considered baselines.

Method	Type	Hyperparameters	Source
RMI [18]	Learned	model types, layer size	[20]
ALEX [7]	Learned	sparsity	[6]
PGM-index [9]	Learned	max error	[28]
RadixSpline [15]	Learned	radix width, max error	[16]
B-tree [3]	Tree	sparsity	[4]
Hist-Tree [5]	Tree	num bins, max error	[27]
ART [19]	Trie	sparsity	[21]
Binary search	Search	-	[26]

key. The sorted array is kept in memory and we perform 20M lookups per run, the keys of which are sampled from the sorted array uniformly at random with a fixed seed. Reported execution times are the average execution time of the median of three runs.

4.5 Baselines

In Section 9, we compare our RMI implementation against a number of baselines listed in Table 2 for which we use the referenced open-source implementations. Due to our focus on ranking the performance of RMIs, we consider all publicly available learned indexes but only some representatives of traditional indexes.

Learned indexes. ALEX [7], PGM-index [9], and RadixSpline [15] are learned indexes discussed in Section 3.1. The index size of PGM-index and RadixSpline is varied based on the maximum error. Additionally, RadixSpline provides a parameter to adjust the size of the radix table that is used to index the spline points. Since we do not consider update performance here, we use the standard variant of PGM-index, which does not support updates. ALEX does not provide any parameters itself, so we vary its size by adjusting the number indexed keys (sparsity) by inserting only every k-th key. In addition, we also consider the reference implementation of RMIs [23] that is configured using its integrated optimizer.

Traditional indexes. B-tree [3] and ART [19] are traditional in-memory index structures. We vary the size of B-tree and ART by adjusting the number of keys that are inserted. Therefore, we use an implementation of ART that supports lower bound queries from SOSD [14]. The recently published Hist-Tree [5] is a tree-structured index. Each inner node in a Hist-Tree is a histogram that partitions the data into equal-width bins. Like learned indexes, Hist-Tree exploits that the data is sorted. Hist-Tree provides two tuning parameters: the number of bins determines the size of inner nodes and the maximum error defines a threshold for the size of a terminal node. We use an implementation of a Compact Hist-Tree that does not support updates in favor of lookup performance [27].

Binary search. We also consider standard binary search over the sorted array without any index as provided by `std::lower_bound`.

5 PREDICTIVE ACCURACY ANALYSIS

In this section, we analyze the impact of hyperparameters on the predictive accuracy of RMIs. Our analysis is divided into three parts.

Segmentation (Section 5.1): We investigate how root models of different types divide the keys into segments.

Position Prediction (Section 5.2): We analyze how accurately different combinations of models approximate the CDF.

Error Bounds (Section 5.3): We examine how different types of error bounds limit the error interval to be searched.

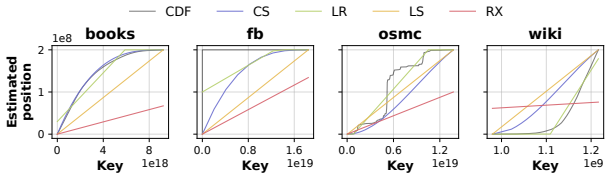


Figure 3: CDF approximations by root models of different types based on which keys are segmented.

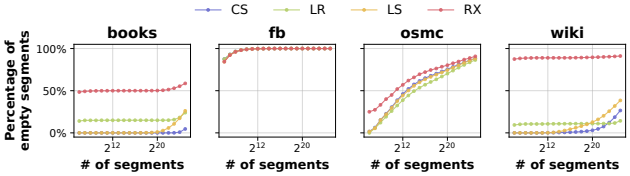


Figure 4: Percentage of empty segments when segmenting the keys with different first-layer models.

5.1 Segmentation

An RMI divides the keys into segments based on its root model's approximation of the CDF. Assuming a root model correctly predicts the position of each key, each segment would consist of the same number of keys. Therefore, RMIs aim for an equal-depth segmentation by design. This approach to segmentation has a crucial weakness: it ignores whether the resulting segments can be accurately approximated by the next-layer models. In contrast, other learned indexes like PGM-index [9] and RadixSpline [15], which are built bottom-up, explicitly create segments that meet a certain error tolerance. Consequently, the quality of an RMI's segmentation cannot be assessed independently of the next layer. In the following, we address two problems that may occur when segmenting keys in an RMI: (1) *empty segments*, which do not contain any keys, and (2) *large segments*, which contain significantly more keys than others. Figure 3 shows the CDFs and the corresponding root model approximations.

Empty segments. Since there is a second-layer model for every segment, empty segments increase the size of an RMI without improving the prediction accuracy. Thus, we should aim for as few empty segments as possible. Figure 4 shows the percentage of empty segments of each model type on each dataset for a varying number of segments. We generally observe that the percentage of empty segments increases with an increasing number of segments. The more accurately a model approximates the CDF, the fewer empty segments it creates. For instance, CS produces empty segments on `books` only after a high number of segments is reached. In contrast, radix predictions often do not cover the full range of positions, e.g., on `wiki`, leaving the segments associated with the non-covered positions empty. The clustered distribution of `osmc` dataset causes percentages to be generally higher and to increase more quickly since the keys are distributed over a small number of segments. Due to the few extreme outliers that strongly affect the CDF approximation of `fb`, all models map the majority of keys to the same position, causing all of these keys to be assigned to the same segment. Increasing the number of segments gradually removes the outliers

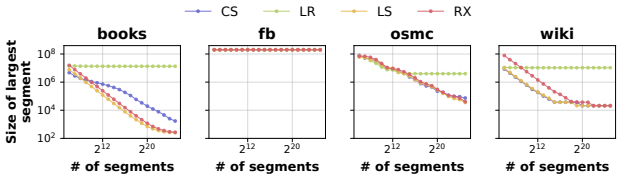


Figure 5: Size of the largest segment when segmenting the keys with different first-layer models.

from this segment, but the segment will continue to contain most keys.

Large segments. Large segments potentially follow a more complex distribution and are more difficult to approximate by the second-layer models. Therefore, large partitions may negatively affect the prediction quality of an RMI. Figure 5 shows the number of keys that reside in the largest segment. Again, the more accurate a model approximates the CDF, the more evenly the keys are distributed over the segments. Logically, the average segment size decreases as the number of segments increases. However, this does not necessarily apply to the largest segment. For LR, the size of the largest partition often remains near-constant. The reason for this is that LR may produce estimates outside the range of valid positions. These out-of-range predictions are then clamped to either the first or last valid position. All keys whose prediction is clamped will be assigned to the same segment. Increasing the number of segments only decreases the size of these segments until the segments consist exclusively of keys whose prediction had to be clamped. CS, LS, and RX do not produce estimates outside the range of valid positions and therefore do not exhibit this problem. As discussed before, on `fb`, almost all keys reside in a single segment, regardless of the number of segments and type of the root models. As we will see in subsequent experiments, the inability of the considered model types to segment datasets with extreme outliers is the main reason for inaccurate predictions, large error intervals, and slow lookups on `fb`.

Summary. When choosing a first-layer model type for segmentation, empty and large segments should be avoided. In our experiments, LS and CS produced the most uniform segments. RX tends to produce many empty segments. LR often creates large segments at the upper and lower end of the key space due to clamping. If none of the models satisfactorily segments the keys as with `fb`, more complex models must be considered.

5.2 Position Prediction

To analyze the impact of model types on prediction accuracy, we train RMIs of all combinations of first-layer and second-layer model types with different second-layer sizes on the four datasets. In Figure 6, we report the median absolute error over all keys as a measure of deviation between predicted position and actual position. We decided against reporting the mean absolute error due to variances caused by high errors on the large partitions when segmenting with LR. In the remainder, we refer to an RMI that uses RX and LR in the first and second layer, respectively, as RX↦LR.

As expected, RMIs with more segments and thus more second-layer models generally produce more accurate predictions. On both

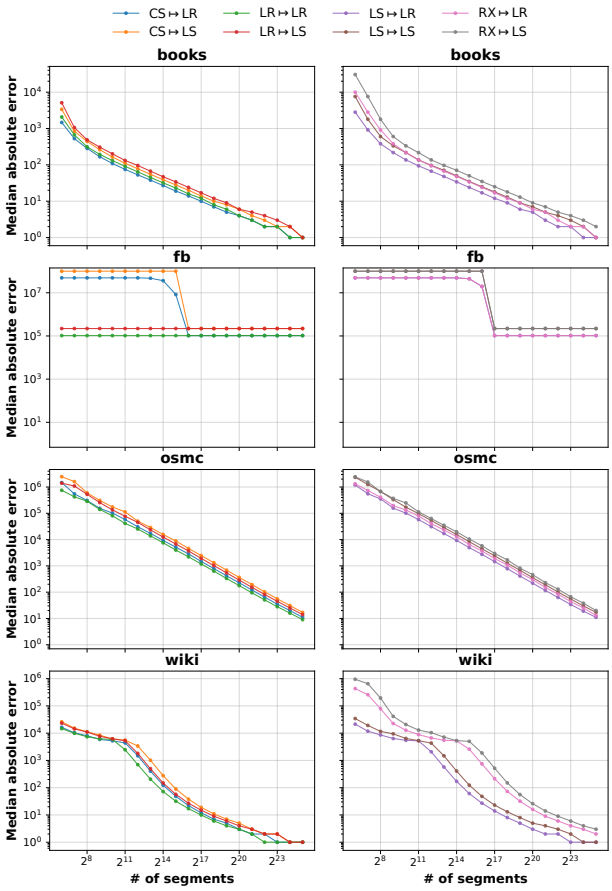


Figure 6: Median absolute error of RMIs with different combinations of first-layer and second-layer models.

the `books` and `wiki` dataset, RMIs with more than 2^{19} second-layer models even achieve errors in single digits. The `osmc` and `fb` dataset are more difficult to approximate. The `osmc` dataset has a clustered distribution that results in a high number of empty segments, making non-empty segments larger on average. Additionally, these segments often have a significant amount of noise and cannot be approximated precisely with the models considered here. Similarly, the large prediction error of `fb` can also be attributed to the single large segment. The sudden drop in prediction error between 2^{15} and 2^{17} segments is due to fewer of the outliers being assigned to the large segment anymore. Although the distribution within that large segment is close to uniform, it still contains a considerable amount of noise that leads to the persistent high prediction error.

Comparing the different RMI configurations, RMIs with LR, LS, and CS as root model achieve similar errors while RX performs slightly worse. This indicates that in terms of prediction accuracy, RX is less suitable for segmentation. Regarding the second-layer models, LR always achieves lower errors than LS. This is expected since LR is the only regression model and minimizes the MSE.

Summary. For the first layer, a segmentation that distributes the keys over many models is a prerequisite for high prediction accuracy. For the second layer, regression models like LR achieve higher

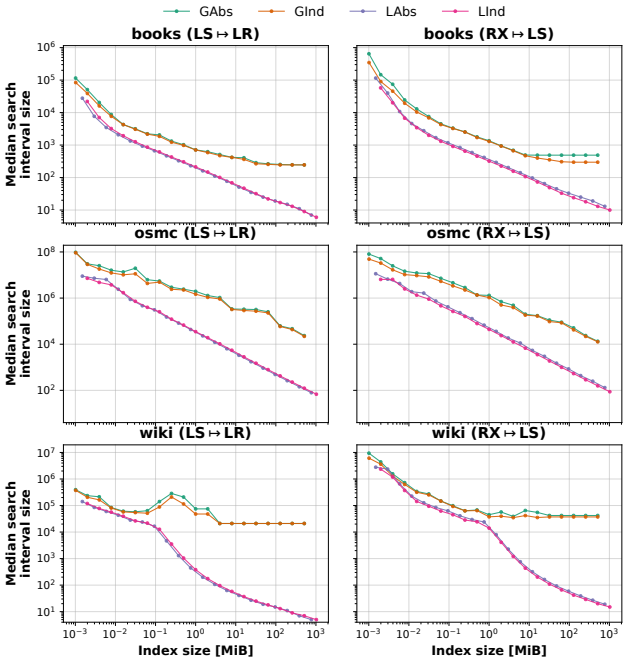


Figure 7: Comparison of error interval sizes for different error bounds for two example model combinations.

accuracy than spline models since regression models minimize the prediction error. Increasing the second-layer size of an RMI further improves its accuracy. Overall, LS↦LR and CS↦LR achieve good accuracy across datasets, except for `fb` due to poor segmentation.

5.3 Error Bounds

Error bounds facilitate correcting prediction errors by limiting the size of interval that has to be searched during a lookup. To evaluate the impact of different error bounds, we again train RMIs with all combinations of first-layer and second-layer model type and varying second-layer size. For each configuration, we compute error bounds of different types and record the error interval sizes over all keys. In Figure 7, we report the median error interval size, i.e., the median number of keys that have to be searched during a lookup. Due to limited space, we only show two combinations of models and omit `fb` as the size of the error interval remains near constant due to inaccurate predictions. However, the observations made and conclusions drawn also apply to the combinations of model types which are not shown.

Global bounds consistently lead to significantly larger error intervals than local bounds, despite the fact that at a similar index size, global bounds allow for more second-layer models and achieve on average more accurate predictions. Global bounds, however, are prone to single bad predictions, whereas local bounds are more robust because they refer to only one model. LInd and LAbs achieve similar error interval sizes. Spline models, which tend to either overestimate or underestimate, profit from LInd. LR, which often achieves similar positive and negative errors, works better with LAbs as LAbs allows for more second-layer models at a similar size.

Summary. Considering RMIs of similar size, local bounds consistently result in smaller error intervals than global bounds. For the preferred second-layer model type LR, LAbs achieves smaller error intervals due to more second-layer models at a similar index size.

6 LOOKUP TIME ANALYSIS

In this section, we analyze the impact of hyperparameters on the lookup performance of RMIs. Our analysis is divided into two parts. **Model Types** (Section 6.1): We investigate the lookup performance of different combinations of first and second-layer model types. **Error Correction** (Section 6.2): We analyze the impact of error bounds and search algorithms on lookup performance.

6.1 Model Types

To evaluate the impact of model types on lookup performance, we train RMIs of all combinations of first-layer and second-layer model type with varying second-layer sizes. We use no bounds and model-biased exponential search (NB+MExp) for error correction as this configuration relies solely on the predictive power of the RMI and thus most clearly illustrates the differences between the various combinations of model types. In Figure 8, we report the average lookup time of each configuration. The dashed horizontal lines are the average time for obtaining a key using binary search.

For a fixed index size, the lookup times of different models within a dataset often differ only slightly, e.g., on `osmc` and `books`, all combinations of models have similar lookup times. However, lookup times vary significantly across different datasets. This observation is consistent with the prediction errors we saw in Section 5.2. The reason for this is that lookup time consists of evaluation time and error correction time. The error correction time accounts for the majority of lookup time and is determined by the prediction error. However, balancing evaluation time and error correction time is a trade-off that has to be carefully considered. In our experiments, we only consider relatively simple models that are fast to evaluate and, as a result, there are only minor differences in evaluation time. In preliminary experiments, we also considered neural networks, which achieved higher prediction accuracy, but the faster error correction was overshadowed by a significantly higher evaluation time ultimately resulting in considerably slower lookups. Of the models considered here, CS is the slowest to evaluate. We can observe the impact of its slower evaluation time compared to LS on `books` where despite CS↦LR being slightly more accurate than LS↦LR, LS↦LR achieves faster lookups. Differences in evaluation time are particularly noticeable when the error correction time is relatively short which often is the case for larger configurations.

Summary. Prediction accuracy is a strong indicator for lookup performance as it determines the error correction time. Therefore, models like CS↦LR and LS↦LR that achieve good accuracy across datasets should be chosen. However, the more accurate the predictions are, the more important become differences in evaluation time and models that are slightly less accurate but faster to evaluate have an advantage. Increasing the second-layer size improves accuracy and causes the lookup time to converge.

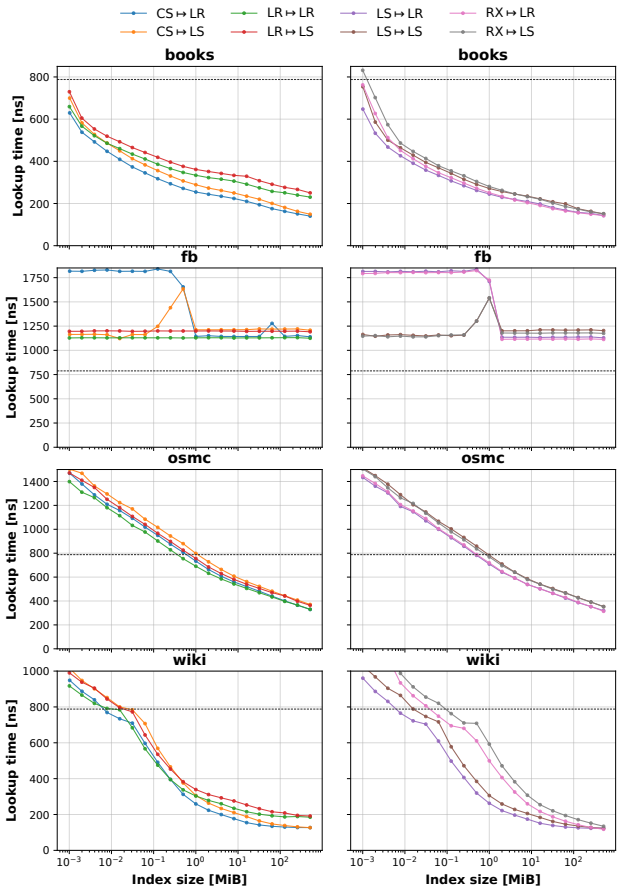


Figure 8: Comparison of lookup time for different combinations of models with NB+MExp for error correction.

6.2 Error Correction

Next, we examine the impact of eight combinations of error bounds and search algorithms for error correction on lookup time. We consider the following combinations. NB is evaluated with MLin and MExp as both search algorithms do not use bounds. GInd and LInd are evaluated with MBin and Bin. GAbs and LAbs are evaluated with Bin only as MBin and Bin are the same in case of absolute bounds. In Figure 9, we report the average lookup time. Due to limited space, we again show only two combinations of models and omit `fb` because lookup performance could not be significantly improved compared to Figure 8 in this experiment. However, our observations also hold for the combinations of models that are not shown here.

We observe that either a configuration with local bounds or without any bounds performs best. Local bounds generally perform better than global bounds, which is consistent with our observation from Section 5.3. Nevertheless, binary search mitigates differences in search interval size drastically, e.g., global and local bounds perform almost identical with LS↦LR on `books`, although the search interval sizes differ by more than an order of magnitude. LInd and LAbs perform almost identical with a maximum performance difference of factor 1.1x. Similar to what we saw in Section 5.3,

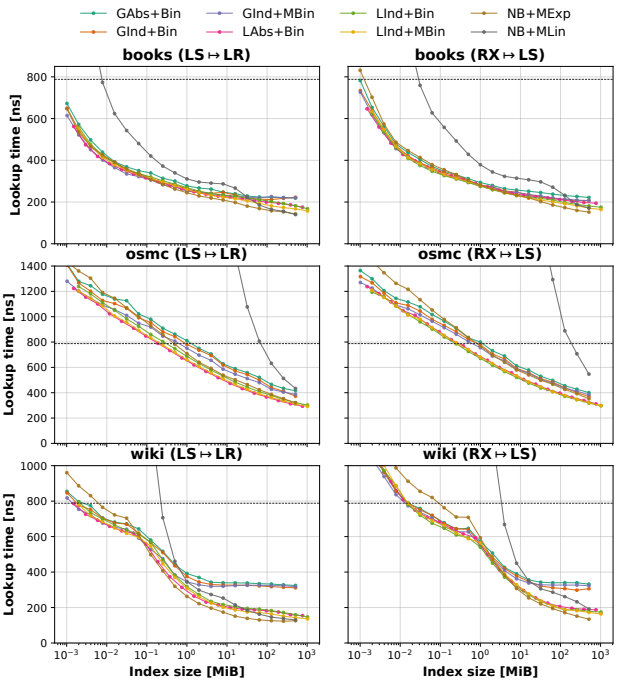


Figure 9: Comparison of lookup time for different error corrections using two combinations of models as examples.

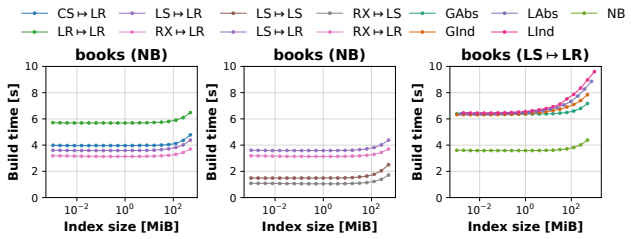
LS works better with LInd as it tends to either overestimate or underestimate, LR works better with LAbs as its loss function often causes the maximum overestimation and underestimation to be similar. Considering LInd, there is hardly any difference between Bin and MBin.

Similar to what we saw in Section 6.1 with respect to model types, the choice of error bounds not only affects error correction time but also evaluation time as error bounds induce overhead for computing the search interval's limits. Hence, RMIs without error bounds are faster to evaluate. The faster evaluation is particularly noticeable when the RMI achieves a high prediction accuracy and thus fast error correction. In these cases, NB+MExp performs better than configurations with bounds as can be seen with `books` and `wiki`. To further analyze when to use NB+MExp over configurations with bounds, we also recorded the mean $\log_2$ error as an estimate of the number of search steps required by MExp. Starting at an mean $\log_2$ error of around 7 to 10, NB+MExp is faster than LAbs+Bin. NB+MLin requires even lower errors to be similarly fast as NB+MExp.

Summary. The best combination of error bounds and search algorithm depends on the predictive accuracy of the RMI. If the mean $\log_2$ error is sufficiently small, NB+MExp performs best due to RMIs without bounds being faster to evaluate. For larger errors, configuration with local bounds such as LAbs+Bin perform better.

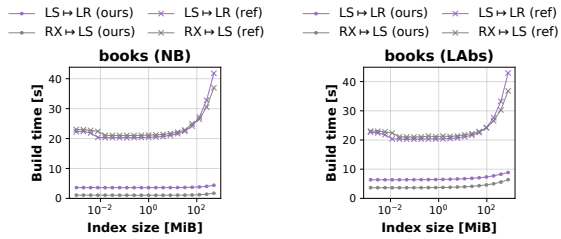
7 BUILD TIME ANALYSIS

In this section, we analyze the build time of our implementation of RMIs and compare it with the reference implementation [23]. Recall that the build process of a two-layer RMI consists of four



(a) Layer 1 type (b) Layer 2 type (c) Error bound

Figure 10: Build times when varying hyperparameters.



(a) No bounds (b) Local absolute bounds

Figure 11: Build times of our implementation (ours) and the reference implementation (ref) with NB and LAbs.

steps: (1) training the root model, (2) creating segments, (3) training the second-layer models, and (4) computing error bounds. Figure 10 shows build times on books. Other datasets are not shown because except for minor caching effects on large configurations, the build time is independent of the dataset. We discuss each aspect that affects build time individually below.

First-layer type. Consider Figure 10a for a build time comparison of different root models. Models in general and root models in particular not only differ in training time, which affects step (1), but also in evaluation time, which affects steps (2) and (4). The most notable difference between the models in terms of training time is whether a model considers all keys, like LR, or a constant number of keys, like LS, CS, and RX. Since the evaluation time of LR and LS is the same, the difference in build time in Figure 10a can be attributed entirely to the training time of the root model. Like LS, RX also considers only two keys for training. Here, the faster build time of RX is caused by the faster evaluation of RX during segmentation. CS is faster than LR because it again only considers a constant number of keys but slower than LS because training and evaluation are slightly slower.

Second-layer type. Consider Figure 10b for a build time comparison of different second-layer models. Analogously to the root model type, the second-layer model type affects training time and evaluation time. Second layers consisting of LS models takes about two seconds less to train than second layers consisting of LR models. Note that in this example, the second layer is never evaluated because we do not compute bounds. Otherwise, evaluation time would be the same for LR and LS.

Error bounds. Consider Figure 10c for a build time comparison of different error bounds. Computing error bounds requires evaluating the RMI on every key plus the actual computation of the bounds. This additional effort explains the difference in built time between NB and configurations with bounds. The difference between individual configurations with bounds is mainly due to branch misses when calculating the bounds. At similar index size, local bounds trigger more branch misses than global bounds and individual bounds trigger more branch misses than absolute bounds.

Index size. Consider again the RMI configuration without bounds in Figure 10c. The build time remains almost constant as long as the entire RMI fits in cache (20 MiB). Once the RMI no longer fits in cache, the build time increases due to cache misses. Next, consider the configurations with bounds in Figure 10c. Here, the previously described branch and cache misses add up and the build time already increases for configurations that are smaller than the cache size. The increase in build time is less pronounced if a configuration produces many empty segments due to less cache misses.

Reference implementation. Figure 11 shows build times of our implementation (ours) and the reference implementation (ref). Figure 11a and Figure 11b compare configurations with NB and LAbs, respectively. Build times for both types of bounds are almost identical for the reference implementation because the reference implementation always computes bounds during training and only decides later whether these computed bounds are kept or discarded. Considering only configurations with LAbs, our implementation improves build times by 2.5x to 6.3x. We attribute this improvement to our optimized segmentation for monotonous root models that avoids copying keys as described in Section 4.1.

Summary. RMIs can be built in a matter of seconds. For a given combination of models, the build time remains almost constant as long as the RMI fits in the cache. The computation of error bounds leads to additional cache and branch misses, which negatively impact build times.

8 RMI GUIDELINE

Based on our findings from the previous sections, we present a compact guideline for configuring RMIs. Our guideline does not guarantee to always provide the fastest lookups but it is easy to follow and achieves competitive lookup performance. Given a maximum allowed index size *budget*, we propose to configure RMIs as follows.

Models types. LS↦LR with the maximum second-layer size that is allowed by the *budget*. CS and LS both segment most datasets well, but we choose LS as it is slightly faster to train and evaluate. Although more accurate predictions can be obtained with CS, CS is only faster for small RMIs, where the improvement in search time outweighs the longer evaluation time. LR as second-layer model minimizes the error and thus always performs better than LS. Larger RMIs generally achieve smaller errors and thus perform better, which is why we choose the maximum number of second-layer models within the *budget*.

Error correction. LAbs+Bin or NB+MExp. Our experiments show that LAbs+Bin performs better than NB+MExp until a certain error threshold is reached. This error threshold is hardware-dependent and must be determined empirically once. We use the mean $\log_2$ error as measure of error to estimate the number of search steps with exponential search and determine the error threshold to be 5.8.

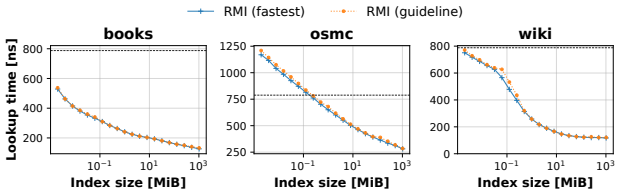


Figure 12: Comparison of lookup time of fastest configuration and guideline configuration.

Whenever the mean $\log_2$ error of our RMI with NB is below the threshold, we use NB+MExp and LAbs+Bin otherwise.

Figure 12 compares the lookup times of configurations obtained by our guideline with the fastest configurations. As before, we omit `fb` as none of the considered models segments `fb` well. We consider size budgets between 2 KiB and 1 GiB. Our guideline is on average only 2.0% slower than the fastest configuration with a maximum performance decline of 11.3% on `wiki`.

Implementing our guideline requires training at most two RMIs:

(1) Train an RMI with LS↦LR and NB that is within *budget*.
(2) Compute the mean $\log_2$ error of the RMI.
(3) If the error is above the threshold, train and use an RMI with LAbs within *budget*. Otherwise, use the already trained RMI.

Limitations. In order to be simple and induce as little overhead as possible, our guideline neglects some aspects that are required for optimal configuration. (1) Our guideline uses fixed model types. While LS↦LR works well for datasets without outliers, a more suitable first-layer model must be sought for datasets with outliers. (2) Our guideline only chooses between LAbs+Bin and NB+MExp based on a rough estimate of expected search steps. In some cases, other error correction strategies are slightly faster.

9 COMPARISON WITH OTHER INDEXES

In this section, we compare our guideline for configuring RMIs with the indexes introduced in Section 4.5 and vary the parameters listed in Table 2 to obtain indexes of different sizes. Configurations of our RMI implementation are chosen based on our guideline. Configuration of the reference implementation are chosen based on its optimizer [23].

9.1 Lookup Time

We first compare lookup times with respect to index size. During a lookup, each index yields a search range, either through error bounds or level of sparsity. We use binary search to find keys in that search range. In Figure 13, we report average lookup times. For indexes with multiple hyperparameters, i.e., RadixSpline and Hist-Tree, we show pareto-optimal configurations in terms of index size and lookup time for better readability. As a result, the number of data points shown differs across dataset. Hist-Tree and ART do not support duplicates and are therefore not evaluated on `wiki`. Overall, our results are consistent with previous reports [12, 14, 22].

Let us first consider the traditional indexes. Hist-Tree is the fastest index on all datasets except `wiki`, but Hist-Tree needs index sizes of 100 MiB and more to reach its full potential. The best-performing configurations of Hist-Tree use a high branching factor resulting in few levels while achieving search intervals of less than

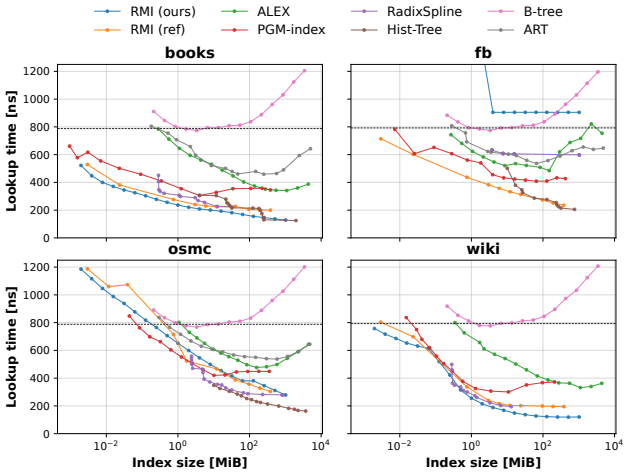


Figure 13: Comparison of lookup times w.r.t. index size.

64 keys. B-tree is the only index whose performance is completely independent of the data distribution but also the slowest index, barely beating binary search. ART is always faster than B-tree but noticeably slower than all learned indexes except ALEX.

The performance of learned indexes highly depends on the data distribution. Up to a certain index size from which Hist-Tree outperforms the other indexes, learned indexes achieve the fastest lookup times. This implies that learned indexes work particularly well for smaller index sizes. On `books`, `fb`, and `wiki` either our implementation of RMIs or the reference implementation dominates the other learned indexes. On `osmc`, both PGM-index and RadixSpline perform better than RMIs. ALEX is clearly the slowest learned index, which can be attributed to its more complex adaptive structure.

Let us now compare our RMI implementation and the reference implementation [23]. On `books` and `wiki`, our implementation dominates the reference implementation despite using our simple guideline. There are two reasons for this. (1) Unlike how the optimizer is described [23], the publicly available implementation [20] does not consider evaluation time in its optimization process and instead chooses configurations that achieve the smallest mean $\log_2$ error. While this results in selecting the configuration with the fastest error correction time, it does not guarantee to select the configuration with the fastest lookup time. The configurations chosen by our guideline consistently have fast evaluation times at the cost of potentially slower error correction. (2) The optimizer of the reference implementation always picks LAbs. Our experiments in Section 6.2 show that for accurate RMIs, NB+MExp performs better, which is considered by our guideline. On `osmc`, no implementation dominates the other. Here, RMIs are never sufficiently accurate for our guidelines to deviate from LAbs+Bin. Thus, differences in performance are solely due to the choice of models. On `fb`, the reference implementation clearly dominates our implementation. As discussed before, LS is not sufficient hor segmenting datasets with extreme outliers. Here, the reference implementation chooses a variant of LR that ignores the lowest and highest 0.01% of keys for segmentation. This approach, while effectively eliminating the outliers in `fb` from the segmentation process, only works if there are at most 0.01% of outliers at either end of the key space. We did

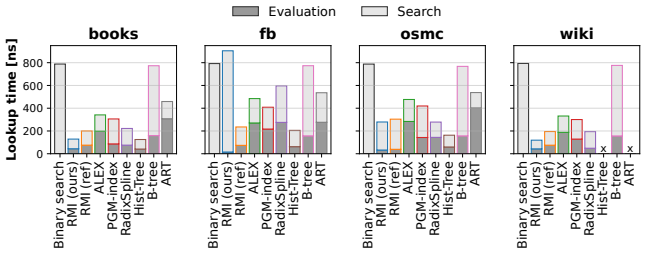


Figure 14: Comparison of evaluation time and search time for the best-performing configuration of each index.

not include this model type in our evaluation because we believe that a more robust solution to segmentation should be sought.

Let us now examine the composition of lookup time from evaluation time (evaluating the model or traversing the tree) and search time (searching within the error interval or data page). Figure 14 shows the lookup time of the best-performing configuration of each index divided into evaluation time and search time. There is a trade-off between fast evaluation and fast search. RMIs clearly prioritize fast evaluation: The evaluation leads to the correct segment in a fixed number of steps, but the RMI does not provide any guarantees on the prediction accuracy. Adding more segments continuously improves the lookup performance because more segments hardly increase the evaluation time while improving the search time. If the evaluation time of our implementation is faster than that of the reference implementation, it is because our configuration does not use bounds. In contrast, PGM-index and RadixSpline prioritize fast error correction: Both indexes cap the maximum error at the cost of a slower evaluation that requires traversing multiple layers or performing intermediate searches. At some point, the improved search time of a smaller maximum error does not compensate the longer evaluation time and the lookup performance decreases. Thus, despite fewer hyperparameters than RMIs, configuring PGM-index and RadixSpline optimally is an elaborate task.

Summary. Learned indexes perform well even at small index sizes. Overall, Hist-Tree is the fastest evaluated index, but it requires sizes of 100 MiB and more to beat learned indexes. Other traditional indexes perform significantly worse on sorted data.

9.2 Build Time

Next, we compare build times with respect to index size. In Figure 15, we report build times which refer to the index configurations evaluated in terms of lookup time in Section 9.1. We show the raw build times without the time required to determine hyperparameters, e.g., by running the reference implementation's optimizer [23] or determining pareto-optimal configurations of RadixSpline and Hist-Tree. Some indexes require data preparation to be built. For instance, ALEX, B-tree, and ART are not only built on the keys but also explicitly require the positions to which these keys should be mapped. Since these preparation steps could be circumvented by a specialized implementation, we do not consider them part of the build time.

The index size of B-tree, ART, and ALEX is determined by the level of sparsity. In contrast to learned indexes, these indexes are built on a subset of the keys and therefore provide fast build times

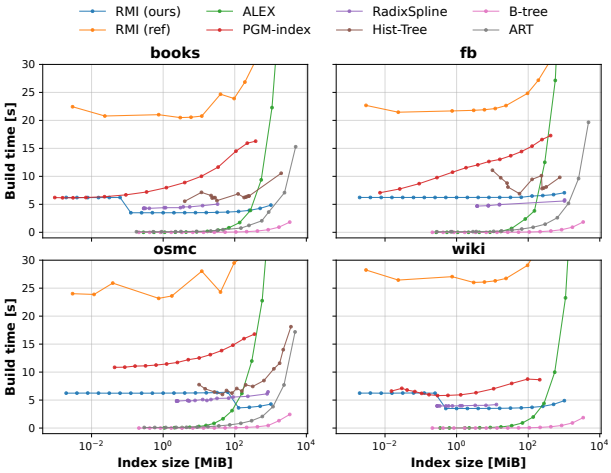


Figure 15: Comparison of build times w.r.t. index size.

especially at smaller index sizes. With an increasing number of keys, the structure of these indexes becomes more complex, e.g., more levels are introduced, and the build time increases. In contrast, RMI, PGM-index, and RadixSpline are always built on the entire dataset. This means that their build times are higher from the outset. RMI and RadixSpline have a fixed number of layers. Therefore, their build time is hardly impacted by the data distribution and only increases once the index no longer fits into cache. The sudden decrease in build time of RMIs on `books` and `wiki` is caused by the guideline choosing an RMI configuration without bounds which is faster to build. PGM-index, on the other hand, has a variable number of layers. Depending on the data distribution and the desired error, more layers have to be trained leading to a steeper increase in build times compared to RadixSpline and RMI. Causes for the differences in build time between our RMI implementation and the reference implementation [23] were already discussed in Section 7. The reference implementation's jumps in build time are caused by varying build times for different model types chosen by its optimizer. Hist-Tree exhibits similar build times to the learned indexes. However at larger sizes, its built time quickly increases due to the increasing depth of the Hist-Tree.

Summary. The benefits in terms of lookup performance of learned indexes come at the cost of significantly higher build times compared to traditional indexes. Thus, the improvement of build times should be a priority of future work.

10 CONCLUSION AND FUTURE WORK

We provided an extensible open-source implementation of RMIs and conducted a comprehensive hyperparameter analysis of RMIs in terms of prediction accuracy, lookup time, and build time. Based on this analysis, we developed a simple-to-follow guideline for configuring RMIs, which achieves competitive performance. In addition, we were able to improve the build time of RMIs by exploiting the monotonicity of models, thereby avoiding the copying of keys when assigning them to the second-layer models. In the future, we plan to extend our implementation to also support multi-layer RMIs and additional model types. We would also like to address the problem of segmenting datasets with extreme outliers.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1009–1024.

[2] Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia. 2018. Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes. https://dawn.cs.stanford.edu/2018/01/11/index-baselines/. (accessed: 2021-11-08).

[3] Rudolf Bayer and Edward M. McCreight. 1970. Organization and Maintenance of Large Ordered Indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*. ACM, 107–141.

[4] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://github.com/tlx/tlx. (accessed: 2021-11-08).

[5] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org.

[6] Jialin Ding. 2020. ALEX: A library for building an in-memory, Adaptive Learned indEX. https://github.com/microsoft/ALEX. (accessed: 2021-11-08).

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 969–984.

[8] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 3123–3132.

[9] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.

[10] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1189–1206.

[11] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7, 992–1005.

[12] Allen Huang, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. Learned Index Leaderboard. https://learnedsystems.github.io/SOSDLeaderboard. (accessed: 2021-11-08).

[13] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 339–350.

[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[15] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. 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*. ACM, 5:1–5:5.

[16] Andreas Kipf and Alexander van Renen. 2020. RadixSpline: A Single-Pass Learned Index. https://github.com/learnedsystems/RadixSpline. (accessed: 2021-11-08).

[17] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. arXiv:1712.01208v1 [cs.DB]

[18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 489–504.

[19] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 38–49.

[20] Ryan Marcus. 2019. RMI: The recursive model index, a learned index structure. https://github.com/learnedsystems/RMI. (accessed: 2021-11-08).

[21] Ryan Marcus, Andreas Kipf, and Alexander van Renen. 2019. SOSD: A Benchmark for Learned Indexes. https://github.com/learnedsystems/SOSD. (accessed: 2021-11-08).

[22] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[23] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2789–2792.

[24] Thomas Neumann. 2017. The Case for B-Tree Index Structures. http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html. (accessed: 2021-11-08).

[25] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org.

[26] C++ Reference. 2000. cppreference.com: std::lower_bound. https://en.cppreference.com/w/cpp/algorithm/lower_bound. (accessed: 2021-11-08).

[27] Mihail Stoian and Andreas Kipf. 2021. CHT: Implementation of the compact "Hist-Tree". https://github.com/stoianmihail/CHT. (accessed: 2021-11-08).

[28] Giorgio Vinciguerra. 2019. PGM-index: State-of-the-art learned data structure. https://github.com/gvinciguerra/PGM-index. (accessed: 2021-11-08).

[29] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*. ACM, 5:1–5:8.