

MIT Open Access Articles

CDFShop: Exploring and Optimizing Learned Index Structures

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Marcus, Ryan, Zhang, Emily and Kraska, Tim. 2020. "CDFShop: Exploring and Optimizing Learned Index Structures."

As Published: <https://doi.org/10.1145/3318464.3384706>

Publisher: ACM|Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data

Persistent URL: <https://hdl.handle.net/1721.1/145662>

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.



CDFShop: Exploring and Optimizing Learned Index Structures

Ryan Marcus
MIT CSAIL

ryanmarcus@csail.mit.edu

Emily Zhang
MIT CSAIL

eyzhang@csail.mit.edu

Tim Kraska
MIT CSAIL

kraska@csail.mit.edu

ABSTRACT

Indexes are a critical component of data management applications. While tree-like structures (e.g., B-Trees) have been employed to great success, recent work suggests that index structures powered by machine learning models (learned index structures) can achieve low lookup times with a reduced memory footprint. This demonstration showcases CDFShop, a tool to explore and optimize recursive model indexes (RMIs), a type of learned index structure. This demonstration allows audience members to (1) gain an intuition about various tuning parameters of RMIs and why learned index structures can greatly accelerate search, and (2) understand how automatic optimization techniques can be used to explore space/time tradeoffs within the space of RMIs.

ACM Reference Format:

Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*, June 14–19, 2020, Portland, OR, USA. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3318464.3384706>

1 INTRODUCTION

Index structures are a fundamental part of many data management applications. Compared to searching an entire database, index structures enable accelerated data access by narrowing the search range to a small portion of the data.

At a high level, all index structures act as an approximate function from a lookup key to an index. For example, a B-Tree realizes this function with a tree structure, providing $O(\log n)$ lookup time. Other index structures [2, 6, 9] deeply optimize this tree structure using a wide array of techniques. Regardless of optimizations, all index structures map a key to

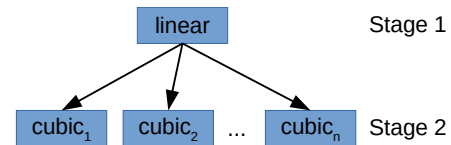


Figure 1: An RMI. The linear model (stage 1) makes a coarse-grained prediction. Based on this, one of the cubic models (stage 2) makes a refined prediction.

a small range of the underlying data, allowing the application to scan just the small range (e.g., only a few pages, depending on the granularity of the index). Viewing indexes as functions, the size of this small required scan can be thought of as the error of the index. When the underlying data is sorted, this function can be thought of a scaled version of the cumulative distribution function (CDF).

Instead of tree or table-based realizations of lookup-key-to-index functions, we can also use machine learning (ML) models, resulting in a learned index structure [8]. By training a model to predict the correct index given a lookup key, theoretically any machine learning model can be used as an index structure. Practically, a good learned index structure requires a model that has both fast inference time and can learn the fine-grained resolution of the key-to-index function.

Recursive model indexes (RMIs) are one such class of models [8] (although others [3–5, 11] exist as well), combining simpler machine learning models together into a multi-staged structure. For example, as depicted in Figure 1, an RMI with two stages, a linear stage and a cubic stage, would first use a linear model to make an initial prediction of an index for a specific key (stage 1). Then, based on that prediction, the RMI would select one of several cubic models to refine this initial prediction (stage 2). RMIs can often achieve 70% faster lookup times with an order-of-magnitude smaller memory footprint than tuned B-Trees [7].

Like traditional index structures, RMIs require tuning to achieve excellent performance. Most machine learning models will have some training error, and thus RMIs, just like B-Trees, will not perfectly predict the location of each key. This error can be decreased by adding more parameters to the ML model, thus decreasing the range of the underlying data to search. However, adding more parameters also increases the model’s memory footprint and inference time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-6735-6/20/06...\$15.00

<https://doi.org/10.1145/3318464.3384706>

Different configurations make different tradeoffs between space and time, but not all configurations are valuable. Users are likely interested in the Pareto front of RMIs with respect to memory footprint and time: RMIs that have the best lookup times for their size. However, these ideal configurations are not always obvious, and vary between datasets.

This demonstration will showcase CDFShop, our tool for designing RMIs and exploring this Pareto front for in-memory data. CDFShop enables the rapid exploration of RMI configurations, giving direct visibility into how configuration changes affect the CDF approximation and the lookup latency. CDFShop also contains an automatic optimizer, which attempts to find RMI configurations on the Pareto front. Since the space of all possible RMIs is large, our optimizer uses a combination of modeling and iterative exploration inspired by game theoretic techniques.

Conference participants will be able to explore the space of RMI configurations over a variety of different datasets, observing how RMI hyperparameters affect prediction accuracy and lookup latency in real time. Additionally, attendees will be able to interact with CDFShop’s automatic optimizer, seeing visualizations of how our technique searches the large space of RMI configurations.

2 THE RECURSIVE MODEL INDEX

For simplicity, we explain only two-stage RMIs here. For a generalization to n -stages, see [8]. A two-stage RMI is a function F trained on N data points (key / index pairs). The RMI F is composed of a single first stage model f_1 , and B second-stage models f_2^i . The value B is referred to as the “branching factor” of the RMI.

Formally, the RMI is defined as:

$$F(x) = f_2^{[B \times f_1(x)/N]}(x) \quad (1)$$

Intuitively, the RMI first uses the stage-one model $f_1(x)$ to compute a rough approximation of the index of key x . This coarse-grained prediction is then scaled between 0 and B , and this scaled value is used to select a model from the second stage, $f_2^i(x)$. The selected second-stage model is then used to produce the final prediction. The stage-one model $f_1(x)$ can be thought of as partitioning the data into B buckets, and each second-stage model $f_2^i(x)$ is responsible for predicting the index of only the keys that fall into the i th bucket.

Training. As described in [8], RMIs can be trained end-to-end by minimizing a loss function. Let $(x, y) \in D$ be the set of key / index pairs in the underlying data. Then, an RMI is trained by adjusting the parameters contained in $f_1(x)$ and $f_2^i(x)$ to minimize the squared error:

$$\sum_{(x,y) \in D} (F(x) - y)^2 \quad (2)$$

Name	Form	Parameters
Linear	$f(x) = ax + b$	a, b
Log-linear	$f(x) = \exp(ax + b)$	a, b
Cubic	$f(x) = ax^3 + bx^2 + cx + d$	a, b, c, d
Radix	$f(x) = x \gg a$	a
Normal	$f(x) = \frac{1}{2} \times \left(1 + \operatorname{erf}\left(\frac{x-a}{b\sqrt{2}}\right)\right)$	a, b

Table 1: A subset of available model types

Intuitively, minimizing Equation 2 is done by training “top down”: first, the stage one model is trained, and then each stage 2 model is trained to fine-tune the prediction. Details can be found in [8]. Our implementation is available at [1].

2.1 Hyperparameters

Equation 1 makes obvious the primary hyperparameters of an RMI. First, the exact form (model) of $f_1(x)$ and $f_2^i(x)$ must be selected. Second, a branching factor B must be chosen.

Models. Table 1 lists a few of the model types available in CDFShop. Choosing the form of the stage-one and stage-two models can have a large impact on the performance of an RMI for two reasons: *error* and *inference time*. For example, if the CDF of the underlying data is more-or-less linear, then linear models may provide a low-error fit. If the CDF has an exponential shape, a log-linear model may have low error. However, computing $\exp(ax + b)$ is substantially more expensive than computing $ax + b$. More complex models may provide a better fit, but at a higher computational cost: thus, sometimes a model with higher error but faster inference may be preferable to a model with lower error.

Branching factor. Intuitively, the branching factor B determines the number of “buckets” that data is divided into by the stage-one model. Creating more buckets (high values of B) is likely to improve prediction accuracy, but may increase inference time. For example, if B is chosen so that all of the RMI’s parameters fit on a single cache line, then the entire RMI is likely to become resident in cache. If B is large, every inference may potentially require a last-level cache miss.

To alleviate the difficulty of tuning an RMI, we next describe a simple optimization approach that can help identify Pareto efficient RMI configurations.

3 OPTIMIZER

Unfortunately, the simplest strategy for finding good RMI configurations, a grid search, is not practical. Evaluating the lookup time of an RMI requires performing a large number of lookups over a large dataset. Worse yet, this evaluation time is related to the error of the RMI: evaluating RMIs with high error takes longer. Luckily, measuring (1) the inference time (the time needed to predict the index of a key), (2) size, and (3) training error of an RMI can be done quickly. Thus, we propose a simple optimization strategy inspired by [10].

Let the set of all RMI configurations be C . For any $c \in C$, we say that the lookup time of the RMI is T_c and the size of the RMI is S_c . We thus seek $Opt(C)$, the subset of configurations that are Pareto optimal (i.e., the set of all RMI configurations for which no other configuration is both smaller and faster).

$$Opt(C) = \{c \mid \neg \exists c' (c' \in C \wedge c \neq c' \wedge T_{c'} \leq T_c \wedge S_{c'} \leq S_c)\}$$

Assumption 1. We model the lookup time T_c of an RMI c as the sum of its inference time I_c and an unknown monotonically increasing function β of its error, E_c .

$$T_c \approx I_c + \beta(E_c) \quad (3)$$

Intuitively, $\beta(x)$ represents the time it takes to search for a key given a search range of x key, but we note that we do not model β explicitly, we merely assume $\beta(x)$ increases as x increases. We note that, as long as Equation 3 holds, any configuration that is not on the Pareto front of (I_c, E_c, S_c) cannot be in $Opt(C)$. Thus, searching the Pareto front of inference time, error, and size can be used to more efficiently, compared to a grid search, generate candidates of $Opt(C)$.

Assumption 2. Imagine two pairs of RMIs that each use the same model, but different branching factors (c_1, c_2) and (d_1, d_2) . Then we assume that:

$$(S_{c_1} = S_{d_1} \wedge S_{c_2} = S_{d_2} \wedge E_{c_1} < E_{d_1}) \rightarrow (E_{c_2} < E_{d_2}) \quad (4)$$

For example, imagine (c_1, c_2) both use cubic models, whereas (d_1, d_2) both use linear models, and that the RMI with cubic models c_1 has a lower error than the RMI with linear models d_1 . If c_1 and d_1 are the same size, and c_2 and d_2 are the same size, then we assume that c_2 has a lower error than d_2 . Concretely, if a 1MB cubic model has lower error than a 1MB linear model, then we assume that an 8MB cubic model will have lower error than a 8MB linear model.

We make an identical assumption about inference time:

$$(S_{c_1} = S_{d_1} \wedge S_{c_2} = S_{d_2} \wedge I_{c_1} < I_{d_1}) \rightarrow (I_{c_2} < I_{d_2}) \quad (5)$$

Both of these assumptions have known exceptions. For example, many neural network architectures are designed with a large or small number of parameters in mind, and do not exhibit good performance otherwise. However, we believe that these assumptions are reasonable for simpler models, such as those in Table 1.

3.1 Search procedure

Given the assumptions above, we find Pareto efficient RMIs using a four step process. Assume that M is the set of all possible models (e.g., Table 1).

In step 1, we measure the inference time and error of the $|M|^2$ possible model configurations at different sizes (e.g.,

4KB, 1MB, 4MB, etc.). We then determine M' , the set of model configurations that have at least one representative of the Pareto front of inference time, size, and error. From our modeling assumptions (Equation 3), we know that all RMIs in $Opt(C)$ use a model configuration that is in M' .

In step 2, we expand our search by measuring the inference time and error of additional configurations with models drawn only from M' , using different sizes than used in step 1 (e.g., in step 2, 8KB, 2MB, etc.). We combine these results with the results from step 1, and construct a candidate set of configurations that are on the Pareto front in terms of size, inference time, and error.

In step 3, we relax assumption 1. We measure the actual lookup performance of the candidate set produced by step 2. This is feasible because this candidate set will be much smaller than C . Then, we can produce a new candidate set containing only the RMI configurations that are Pareto efficient in terms of size and actual lookup time.

In step 4, we relax assumption 2. We measure the actual lookup performance of a small number of additional configurations selected by varying the sizes of the configurations in the candidate set from step 3. For example, if the candidate set from step 3 contained a 1MB linear model, we would additionally test a 512KB linear model and a 1.5MB linear model, providing a better coverage of the Pareto front.

After step 4, the RMI configurations are displayed to the user, so that the user can easily navigate the size vs. latency landscape. Experimentally, we found that our optimization technique always found RMI configurations as good or better than those found by hand tuning in [7].

4 DEMONSTRATION

Our demonstration is comprised of two scenarios. In the first scenario, users can explore how different RMI configurations affect lookup performance and memory footprint, while comparing RMIs against binary search and B-Trees. In the second scenario, users can investigate CDFShop's optimizer, visually exploring each step of the process.

Scenario 1: Exploring RMIs. Our first scenario will allow users to test and explore different RMI configurations. Users can (1) visualize how the resulting RMI approximates the CDF of the data distribution, (2) measure the lookup performance of any RMI configuration, and (3) explore the size vs. performance tradeoffs inherent to index structures, comparing against binary search and a cache-optimized B-Tree [7].

A screenshot of the interface for this scenario is shown in Figure 2. The pull-down menu in the top-right can be used to select from several datasets. The top-left plot shows the keys and indexes of the dataset (black) and the index predicted by the current RMI (red line). As pictured, the plot shows all 200 million keys in the dataset. Users can zoom in

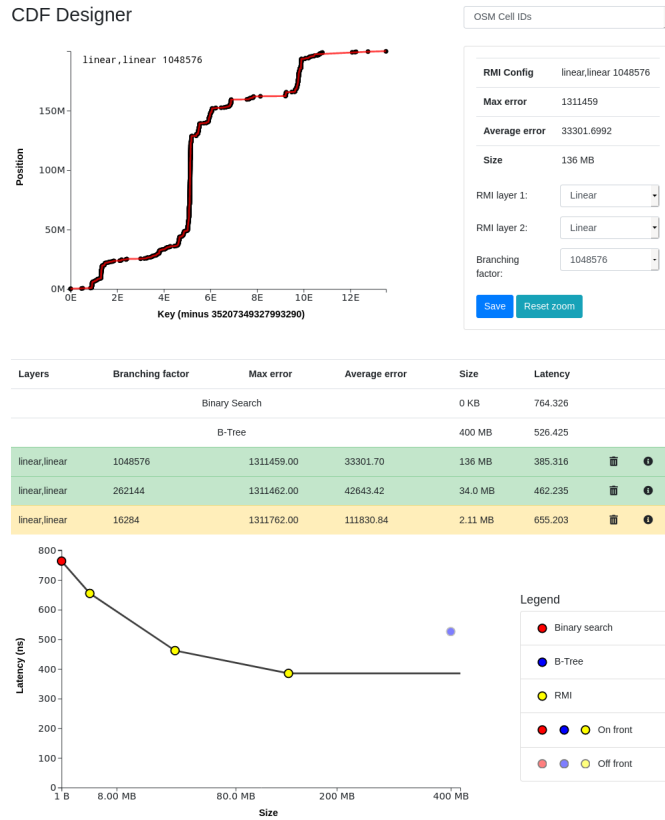


Figure 2: Testing RMI configurations

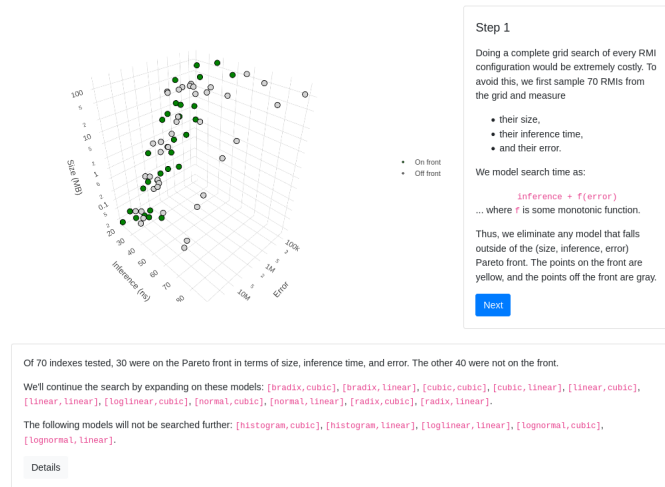


Figure 3: Exploring automatic optimization

on the plot using the mouse. The panel on the right shows information about the current RMI (maximum and average error, size), and contains three pull-down menus to select the configuration (stage 1, stage 2, and branching factor). As users select different configurations, the key/position plot is updated with the new prediction in real time.

When a user saves an RMI configuration, the currently selected RMI configuration is added to the table (middle), and its lookup latency is measured. Users can click the “info” icon on each row the table to see additional information about an RMI, including a plot of its error distribution. Each RMI configuration in the table also appears in the size/latency plot (bottom). The black line in this plot indicates the currently-discovered Pareto front, illustrating the tradeoff between index size and lookup latency. Users can explore the space of RMIs in an iterative fashion, by increasing or decreasing the size of the RMI or by modifying the models used.

Scenario 2: Automatic optimization. Our second scenario will visualize the automatic RMI optimization process. For a chosen dataset, users will see visual representations of each of the four steps described in Section 3.

A screenshot of the interface for the first step of the optimization process is shown in Figure 3. The plot shows the size/error/inference time relationships between configurations. The green points are on the Pareto front, and thus correspond to configurations that will be explored in the next step. Gray points are off the front, and correspond to model configurations that can be excluded from the search. The lower panel shows which models will be further explored, and which have been eliminated.

The blue “Next” button begins the second optimization stage (as described in Section 3), and the user is shown a similar display containing those results. After progressing through the four optimization steps, users are shown a plot of the discovered time/space Pareto front, and can inspect the configurations that fall on that front.

5 CONCLUSIONS

Learned index structures can provide fast lookup times with small memory footprints, but require proper tuning. Our demonstration will give audience members an intuition for how the parameters of RMIs affect the lookup latency and model size tradeoff. Additionally, our demonstration illustrates how this process can be automated under a modest set of assumptions.

REFERENCES

- [1] MIT RMI, <https://learned.systems/rmi>.
- [2] R. Binna et al. HOT: A height optimized trie index. In *SIGMOD '18*.
- [3] J. Ding et al. ALEX: An Updatable Adaptive Learned Index. *arXiv '19*.
- [4] P. Ferragina et al. The PGM-index. *arXiv '19*.
- [5] A. Galakatos et al. FITing-Tree. In *SIGMOD '19*.
- [6] C. Kim et al. FAST: Fast architecture sensitive tree. In *SIGMOD '10*.
- [7] A. Kipf et al. SOSD. In *MLForSystems @ NeurIPS '19*.
- [8] T. Kraska et al. Learned Index Structures. In *SIGMOD '18*.
- [9] V. Leis et al. The adaptive radix tree. In *ICDE '13*.
- [10] R. Marcus et al. NashDB: An Economic Approach to Fragmentation, Replication and Provisioning for Elastic Databases. In *SIGMOD '18*.
- [11] V. Nathan et al. Learning Multi-dimensional Indexing. In *MLForSystems @ NeurIPS '19*.