Learned Hash-Index: A Collision Competition

Thao Nguyen, Eric Xu, Jesse Doan
{thao2605, ericxu0, jdoan21} @ stanford.edu

4 June 2019

Abstract

Learned index structures are rapidly sparking more interest within the machine learning and data systems communities, as the work of Kraska et al. [4] reveals that learned indexes are competitive with traditional indexes in terms of both speed and memory. Learned indexes ultimately model the cumulative distribution function (CDF) of a dataset, and by scaling up the outputs, we can treat the learned indexes as hash functions. In this paper, we extend the recursive model in [4] to learn the CDFs of other distributions, such as uniform, lognormal, and normal distributions. Moreover, we leverage the monotonicity of the CDF by using a Calibrated Linear Model as the submodules in the recursive model. We show that our model yields improvements, in terms of collision rate, bucket utilization, and average bucket height, over traditional hash functions for synthetic linear and lognormal datasets.

1 Summary of Learned Index Structures

1.1 Introduction

Databases are becoming more and more widespread in a number of fields including business, education, and computer science, just to name a few. It's used not only to store and sort data and important information, but also to have quick and accurate retrievals of data.

When it comes to efficiency, we have seen usage of various index structures that correspond to different operations on the database. One example that we have seen from CS 166 is B-Trees, which help us retrieve an entry from a sorted array given the key.

However, these data structures are general-purpose and disregard any inherent patterns that may exist in this data. For example, if we have collected data about a company's sales throughout a month, there may be temporal trends that one can take advantage of to create an efficient data structure for querying. This is where Kraska et al. [4] draws a similarity between traditional index structures and machine learning (ML) models. An index (e.g. a B-Tree index) can be thought of as a function that given some key, it outputs a definite value, often measured in relation to the whole dataset (e.g. position within an array of sorted records). Similarly, a model takes in an input and outputs a prediction.

The original paper argues for the potential of machine learning to learn the underlying patterns in the data and consequently produce specialized index structures, which are called **learned indexes**. This would, in turn, drastically improve the performance of database systems on real-world data sets. The paper studies 3 types of indexes, namely existence index, range index, and point index, the last of which is the main focus of our project.

In the next few sections, we briefly summarize our main takeaways from the paper that we piggy-back off in this paper.

1.2 From CDFs to Indexes

One observation from [4] is that a model that predicts the position of a key inside a sorted array is capable of approximating the cumulative distribution function (CDF) of the data. We can leverage this learned CDF to predict the position as: $p = F(\text{Key}) * N$, where p is the position estimate, $F(\text{Key})$ is the estimated CDF for the data to estimate the likelihood to find a key smaller or equal to the look-up key $P(X \leq \text{Key})$, and N is the total number of keys. In simpler terms, given a desired look-up key, we have an estimated CDF which we then scale by N keys to find a position estimate.

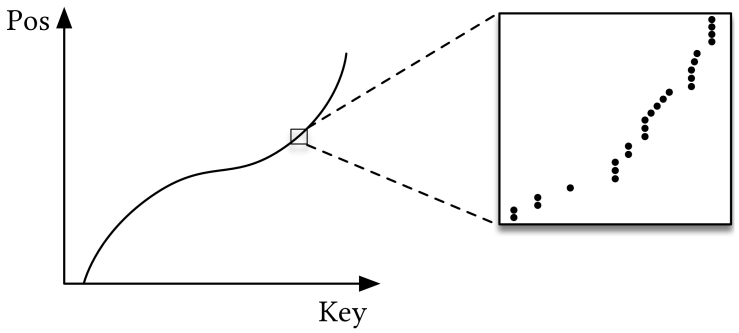


Figure 1: Indexes as CDFs

1.3 The Last Mile Problem/Recursive Model Index

While learned models are excellent at capturing the general distribution of the input data, they sometimes have trouble with the "last mile" - learning the fine-grained details in a small range. Typically, models can achieve higher accuracy in narrowing down output positions at the cost of significantly more space and CPU time. This trade-off explains how a model can be efficient in approximating, say, the general shape of a CDF, but have difficulty being precise at the singular data instance level.

In order to address this issue to achieve better accuracy, we can use simpler models on different portions of the data so that they can just focus on learning a subset of the data. From this, Kraska et al. proposes the recursive regression model (seen in the figure below).

For this model, we create a hierarchy of models where, at each stage, the model takes the key as an input and picks another model based on it, until we reach the final stage to predict the position.

The formal definition is as follows: define model $f(x)$ where x is the key and $y \in [0, N)$ the position. At stage ℓ, there are M_ℓ models. At stage 0, train the model to be $f_0(x) \approx y$. For model k in stage ℓ, denoted by $f_\ell^{(k)}$, train it with the following loss:

$$L_\ell = \sum_{(x,y)} (f_\ell^{\lfloor M_\ell f_{\ell-1}(x)/N \rfloor}(x) - y)^2 \qquad L_0 = \sum_{(x,y)} (f_0(x) - y)^2.$$

A simpler way to think about the models is that each model makes a prediction with some error about the position for the key and then that prediction is used to select the next model,

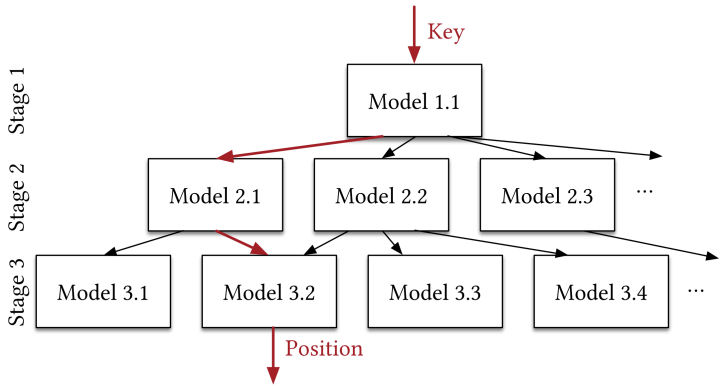
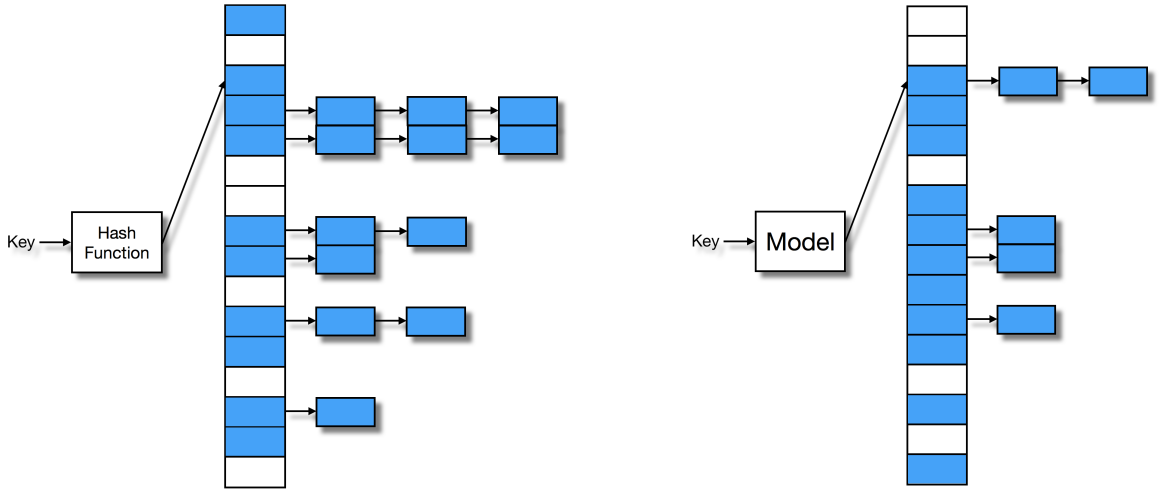


Figure 2: Staged Models

which seeks to make a better prediction and lower the error. Kraska et al. propose that this architecture has several benefits including how it resembles a B-Tree in in dividing the search space into smaller sub-ranges, making it easier to reach the "last mile" accuracy with fewer operations.

1.4 Point Index



(a) Traditional Hash-Map

(b) Learned Hash-Map

Figure 3: Traditional Hash-Map vs. Learned Hash-Map

When performing point look-ups, hash-maps play an important role. We can apply the same concept of learning an ML model that functions as a traditional hash-map (see figure below). This is done by leveraging what we saw earlier in learning the CDF. We can scale the CDF by the targeted size M of the hash-map and use $h(K) = F(K) * M$, with key K as our hash function. As a result, no conflicts would exist if the model F perfectly learned the empirical CDF of the keys, provided that we have as many buckets as the number of keys,

which is commonly the case in real-world applications. To estimate the CDF, we again use the recursive model architecture as described in the previous section.

Ultimately, [4] shows some promising results that make a case for the benefits that learned indexes can provide. Comparing the performance between the learned index and B-Tree reveals memory reductions and faster look-ups offered by the former. They also show considerable decrease in conflicts with the learned hash-map.

Overall, Kraska et al.'s main contribution is to explain and justify a new approach to building indexes, opening up an exciting new direction in research. Although ML is traditionally considered to be computationally expensive, the paper takes into account existing developments in hardware and argue for the increased viability of these learned models.

2 Our Approach

Inspired by the results in [4], we further explore the ability for a model to learn the CDF of various distributions, and we report our findings in this paper. More specifically, we develop a model that learns CDFs to act as a hash function. While traditional hash functions satisfy nice properties such as n-independence, they still have a sizeable expected collision rate (as shown in Section 3.4). However, as mentioned in [4], a model that correctly learns the CDF will have no conflicts. Hence, we benchmark traditional hash functions against our learned model using various metrics described in Section 3.2. We also discuss the memory usage (in terms of the number of trained parameters) and the training time for our learned model.

2.1 Baseline

As baselines, we use a MD5 hash function from the built-in Python hashlib library, and Murmur3. For Murmur3 and MD5, we convert our input datapoints into strings from their numeric value (floats or integers). From here, we feed these strings into the hash function.

2.2 High Level Model Architecture

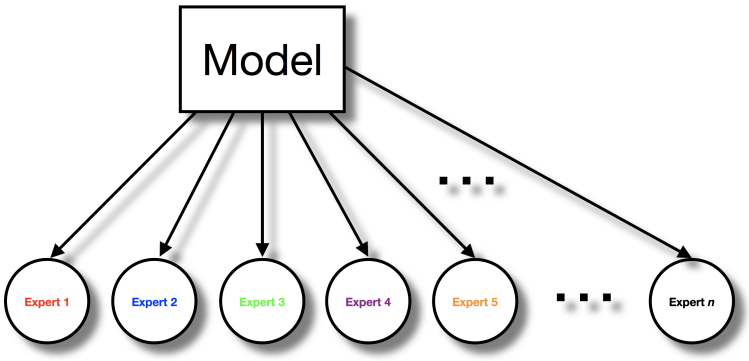


Figure 4: Learning Experts

Our method is based off the Recursive Model Index (RMI) as discussed in section 1.3. We have 2 stages: stage 1 model takes in a key (i.e. a value sampled from the distribution) and tries to predict its position in the CDF. Based on this rough estimate, it finds the expert on the lower layer in charge of that range and the expert tries to do the same prediction task again (see Figure 4). We experiment with various architectures for both the stage 1 model and the stage 2 experts.

2.3 Replicating Results in [4]

We first try to reproduce the performance in the original paper for the lognormal dataset, but encounter a lot of instability during training. For our initial experiments, we use the same architecture as mentioned in [4]: neural networks with fewer than 2 hidden layers and ReLU activation for stage 1, and linear regression models (no hidden layer) for stage 2. As a result, we explore several alterations to the original architecture, including:

- Increase the gap between consecutive positions: Given that our dataset size is on the order of 10^5, the margin of error between 2 consecutive positions is on the order of 10^{-5}. We hypothesize that this small difference may incur some difficulty for weight updates and learning rate selection. Thus, we experiment with scaling the gap between labels by orders of magnitude. We observe that the model also scales its outputs correspondingly to lower the training loss, but stops improving after predicting values of median magnitude.

- Tree-based Regressor: we try LightGBM [3], which can enforce monotonic constraints between inputs and outputs during tree splitting and are able to get more overfitting compared to using neural networks. However, the accuracy level is still not good enough.

- Convert stage 1 to classification task: we obtain high accuracy in expert assignment by converting stage 1 task from position regression to multiclass classification. However, this does not seem to align with the purpose of having stage 1 as a coarse learned point index.

Learning rate and the magnitude of the data after preprocessing seem to play an important role in making the model converge to good optima. Unfortunately, the original paper did not disclose full details on how their hyperparameters were tuned. We also apply the same architecture to a dataset that follows a uniform distribution, which is supposed to be easy given the linear relationship between keys and positions, but we are unable to obtain a good performance.

2.4 Calibrated Linear Model

Note that our keys and positions follow a strict monotonic relationship (as the key value increases, the position also increases). This narrows down the space of possible outputs for a given key given its neighboring keys, so a model that can understand this relationship should be able to learn the distribution better. As a result, we make use of Calibrated Linear Models (CLM) which enforces this monotonic relationship during training.

We use TensorFlow Lattice `calibrated_linear_regressor` which instantiates a linear regression model that calibrates the input using piece-wise linear calibrated functions and then linearly combines the inputs [1]. The algorithm for training this regressor is based off of [2], which Howard and Jabara attempt to simultaneously learn a monotonic transformation Φ and a hyperplane classifier that predicts a label y given an input $\Phi(x)$.

More specifically, they parameterize Φ as a piece-wise linear function using a set of K knots $\{z_1, z_2, ..., z_K\}$ and associated positive weights $\{m_1, m_2, ..., m_K\}$. Then we have $\Phi(x) = \sum_{i=1}^{K} m_i \phi_i(x)$ where

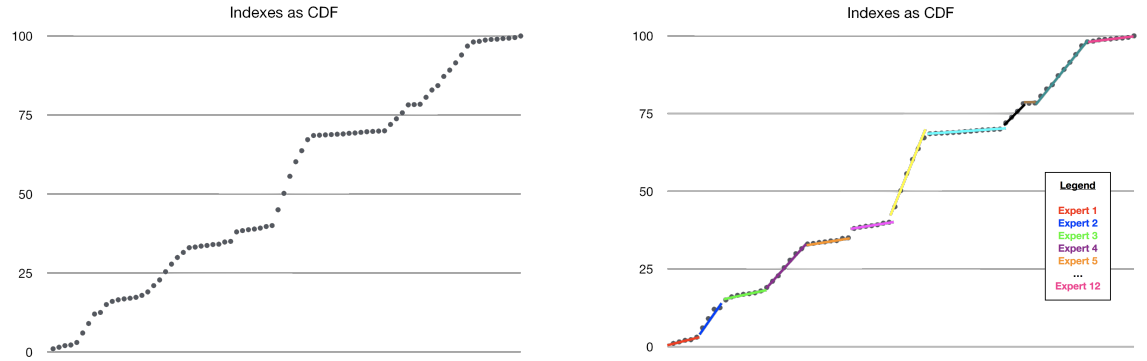
$$\phi_i(x) = \begin{cases} 0 & x \leq z_i \\ \frac{x - z_i}{z_{i+1} - z_i} & z_i \leq x < z_{i+1} \\ 1 & x \geq z_{i+1} \end{cases}$$

They then combine these constraints with a standard support vector machine formulation to obtain the final optimization problem.

The architecture provided by TensorFlow Lattice, `calibrated_linear_regressor`, has a similar formulation; it learns a monotonic mapping from the inputs to a feature space and a linear regression model in that feature space. As a result, the number of parameters trained is proportional to the number of knots, or $O(K)$.

Consequently, we utilize CLMs as the submodules in our final architecture. At stage 1, we train a CLM that predicts a rough estimate for the position of each key. Then for stage 2, we train E experts, each of which are also CLMs. The ith expert is responsible for positions in the interval $\left[\frac{i}{E}, \frac{i+1}{E}\right)$ for $0 \leq i < E$. Therefore, if the stage 1 CLM predicts $\hat{y}$ for some key k, key k is then fed through the expert with index $\lfloor \hat{y} \cdot K \rfloor$ to obtain its final prediction.

Ideally, each expert learns a small interval of the CDF. The ideal performance is depicted in Figure 5.



(a) Indexes as CDF

(b) Creating Experts

Figure 5: From Input Data to Experts

3 Experiments

3.1 Datasets

We generate several synthetic datasets, each of size $N = 100000$. To generate the values for each dataset, we sample N points from the given distribution to use as input values and sort them in increasing order. Then the CDF at the i^{th} point in sorted order is approximately i/N, which we use as output value for our predictive models. Each dataset is split with the ratio of 60:20:20 for train, validation and test sets respectively. These datasets contain different characteristics that we hope illuminate the ability (and inability) of the trained models to capture the underlying known CDFs:

- **linear**: This dataset is drawn from a uniform distribution in the interval $(-5, 5)$. The CDF of a uniform distribution is linear, which should be easy for our model to learn.
- **lognormal**: This dataset is drawn from a log-normal distribution with $\mu = 0$ and $\sigma = 2$. It contains a heavy-tail, making its CDF very non-linear. In addition, Kraska et. al used this log-normal dataset with the same parameters as a dataset in [4].
- **normal**: This dataset is drawn from a normal distribution with $\mu = 0$ and $\sigma = 0.0001$. The extremely low variance causes the CDF to contain a large jump within a small range (centered around 0), which increases its difficulty.

3.2 Metrics

For all of the data structures, we use a load factor of 1, meaning that the number of slots/ buckets is equivalent to the size of the dataset.

Compared to the reported results for learned hash index in the original paper (see Figure 8 in [4], we feel that having multiple metrics other than conflict rate give a more complete picture of our model's ability to capture the spread of the data. Thus, we define the following metrics:

- Collision Rate: The ratio of the number of buckets containing at least 2 elements to the number of buckets containing at least 1 element (i.e. fraction of the utilized buckets with hash conflicts)
- Bucket Utilization: The ratio of the number of buckets containing at least 1 element to the number of all available buckets.
- Average Bucket Height: From buckets that contain at least 1 element, the average number of elements in each bucket. This metric is also equivalent to the average runtime for accessing, inserting, or removing an element in a hash-map.

3.3 Baseline Results

Data Set	Collision Rate	Bucket Util	Avg Bucket Height
linear.test	0.416	0.6341	1.577
lognormal.test	0.414	0.6350	1.575
normal.test	0.416	0.6312	1.584

Table 1: MD5 Performance

Data Set	Collision Rate	Bucket Util	Avg Bucket Height
linear.test	0.413	0.6328	1.580
lognormal.test	0.420	0.6310	1.585
normal.test	0.419	0.6307	1.586

Table 2: Murmur3 Performance

In general, we see that MD5 and Murmur3 produce similar performance across different datasets, with a collision rate of around 0.41, bucket utilization of around 0.63, and average bucket height of around 1.58. As we show below in Section 3.4, these numbers are nearly optimal for randomized hash functions.

3.4 Theoretical Verification for Expected Performance

We now prove two results on the expected fraction of nonempty buckets and the expected fraction of buckets with a collision. Here, we assume we have access to an "ideal" hash function. Knowing these ratios help us understand how close or far our hash functions are to the optimal hash function.

To hash n elements into n buckets, we assume our "ideal" hash function to be a uniform, n-independent hash function.

Theorem 1. *Let h be a uniform, n-independent hash function that hashes elements to n buckets. The expected fraction of nonempty buckets after hashing n elements is $1 - \frac{1}{e}$.*

Proof. Let X_i be an indicator variable equal to 1 if bucket i contains at least one element. The probability at least one element hashes to bucket i is one minus the probability no element hashes to bucket i. Since h is uniform, the probability any one element does not hash to bucket i is $\frac{n-1}{n}$. Since h is n-independent, the probability none of the n elements hash to bucket i is the product of the probabilities that each element does not hash to bucket i, or $\left(\frac{n-1}{n}\right)^n$.

It follows that $E[X_i] = Pr(X_i = 1) = 1 - \left(\frac{n-1}{n}\right)^n$, and by linearity of expectation, $E\left[\frac{\sum_{i=1}^n X_i}{n}\right] = \frac{\sum_{i=1}^n E[X_i]}{n} = 1 - \left(\frac{n-1}{n}\right)^n = 1 - \left(1 - \frac{1}{n}\right)^n$. Noting that $\lim_{n\to\infty}\left(1 - \frac{1}{n}\right)^n = \frac{1}{e}$, we conclude that the expected fraction nonempty buckets is $1 - \frac{1}{e}$, as desired. □

Theorem 2. *Let h be a uniform, n-independent hash function that hashes elements to n buckets. The expected fraction of buckets containing a collision after hashing n elements is at most $1 - \frac{2}{e}$.*

Proof. Let X_i be an indicator variable equal to 1 if bucket i contains at least at least two elements. The probability at least one element hashes to bucket i is one minus the probability at most one element hashes to bucket i.

We consider two cases for a particular bucket i. If no element hashes to bucket i, we have from the previous theorem that this occurs with probability $1 - \left(1 - \frac{1}{n}\right)^n$.

Otherwise, exactly one element hashes to bucket i. There are n possible choices for some element to hash to bucket i, and since h is uniform, this happens with probability $\frac{1}{n}$. Again, since h is an n-independent hash function, the probability the other elements do not hash to bucket i is $\left(\frac{n-1}{n}\right)^{n-1}$.

Putting everything together, we find that

$$E[X_i] = 1 - \left(\frac{n-1}{n}\right)^n - n \cdot \frac{1}{n}\left(\frac{n-1}{n}\right)^{n-1} = 1 - \left(\frac{n-1}{n}\right)^{n-1}\left(\frac{n-1}{n} + 1\right)$$

Moreover, by linearity of expectation,

$$E\left[\frac{\sum_{i=1}^n X_i}{n}\right] = \frac{\sum_{i=1}^n E[X_i]}{n} = 1 - \left(\frac{n-1}{n}\right)^{n-1}\left(\frac{n-1}{n} + 1\right) = 1 - \left(\frac{n-1}{n}\right)^{n-1}\left(\frac{2n-1}{n}\right).$$

But $\frac{2n-1}{n} > \frac{2(n-1)}{n}$, so we have

$$E\left[\frac{\sum_{i=1}^n X_i}{n}\right] < 1 - 2\left(\frac{n-1}{n}\right)^{n-1}\left(\frac{n-1}{n}\right) = 1 - 2\left(1 - \frac{1}{n}\right)^n$$

Again, $\lim_{n\to\infty}\left(1 - \frac{1}{n}\right)^n = \frac{1}{e}$, so we conclude that the expected fraction of buckets containing a collision is at most $1 - \frac{2}{e}$, as desired. □

Observe that our metric bucket utilization is the empirical estimate of the expected number of nonempty buckets. Since $1 - \frac{1}{e} \approx 0.632$, we see from Tables 1 and 2 that MD5 and Murmur3 in practice perform comparably to an "ideal" hash function.

Since collision rate is the number of buckets with at least two elements divided by the number of buckets with at least one element while bucket utilization is the fraction of total buckets with at least one element, the product of these two metrics is the fraction of buckets containing a collision. Multiplying the two values shown in Tables 1 and 2, we see that they indeed approach $1 - \frac{2}{e} \approx 0.264$, demonstrating that MD5 and Murmur3 also achieve the ideal fraction of buckets containing a collision.

3.5 Learned Index Results

.

Data Set	Collision Rate	Bucket Util	Avg Bucket Height
linear.test	0.419	0.633	1.579
lognormal.test	0.419	0.631	1.584
normal.test	0.417	0.632	1.582

Table 3: Learned-Index Stage 1 Test Performance

Data Set	Collision Rate	Bucket Util	Avg Bucket Height
linear.test	0.384	0.675	1.482
lognormal.test	0.394	0.666	1.503

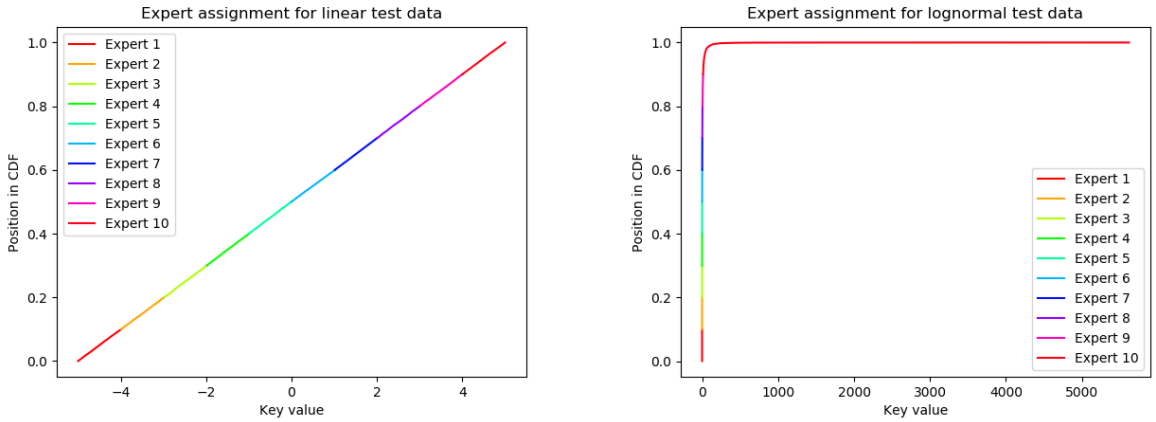
Table 4: Learned-Index Stage 2 Test Performance

Our best model uses ten experts in stage 2, and all eleven of the CLMs were trained with 1000 knots. As a result, approximately 11000 parameters are trained in the model. Indeed, the weights in each model is dominated by $m \in \mathbb{R}^{1000}$, for parameterizing Φ. Thus, the total number of parameters is about 18% of the number of training examples. In general, we observe a trade-off between the size of the model and its performance.

However, training time is extremely low for the model. The stage 1 CLM only needs one epoch of training for all three datasets. For the linear dataset, the stage 2 models are also trained in one epoch, and for the lognormal dataset, the stage 2 models are trained in four epochs. Overall, the model takes around two to three minutes to train.

Inspecting the results in Tables 4 and 5, we see that even with only 1 layer, our learned index structure achieves performance on par with the standard hash libraries. When an additional layer is added to better deal with the last mile problem, we achieve performance better than tradition hash functions on all 3 metrics.

Unfortunately, we notice that this additional improvement does not apply to the normal dataset and thus exclude the performance for its corresponding stage 2 from this report. The low variance of this normal distribution causes the CDF to be practically a step function at around 0.5. Since the experts are responsible for evenly spaced ranges, its very possible that one expert is forced to learn this whole step function by itself while the other experts only need to learn a flat line. Learning the step function is difficult with a linear regression model, which causes the overall model to not learn the CDF properly. We tried increasing the number of experts (so that multiple experts can share the burden in learning the step function), but we did not see substantial improvement in the results.



(a) On Linear Test Data

(b) On Lognormal Test Data

Figure 6: Expert Assignment for Test Data

On the other hand, for the linear and lognormal dataset, we successfully approximate the CDF using the monotonic functions learned by the ten experts. We plot the estimated CDFs as a set of piece-wise monotonic functions, as shown in Figure 6.

4 Final Thoughts

As we have shown in this paper, training a model to learn the CDF of a data distribution yields improvements, in terms of collision rate, bucket utilization, and average bucket height, compared to traditional hash functions that are near optimal randomized hash functions. However, the model does not perform well for arbitrary distributions; when the CDF is very nonlinear, the experts in the model are unable to learn accurate piece-wise linear approximations.

In order to better learn more complex CDFs, a deeper model may be necessary. For example, we could use more stages to reduce the "last mile" issue. Alternatively, we could replace the CLMs with a lattice regressor [1], which also enforces the monotonicity between our inputs and outputs. Also, we could use some adaptive expert assignment algorithm; instead of dividing the range evenly across experts, we could assign ranges so that the complexity of the CDF is evenly distributed across the experts.

For future work, we would like to explore hashing multidimensional data and compare it with hash functions like MD5 and Murmur3.

Our code and experiments are available at `https://github.com/thaonguyen19/Learned_HashMaps`.

References

[1] Maya Gupta, Andrew Cotter, Jan Pfeifer, Konstantin Voevodski, Kevin Canini, Alexander Mangylov, Wojciech Moczydlowski, and Alexander Van Esbroeck. Monotonic calibrated interpolated look-up tables. *The Journal of Machine Learning Research*, 17(1):3790–3836, 2016.

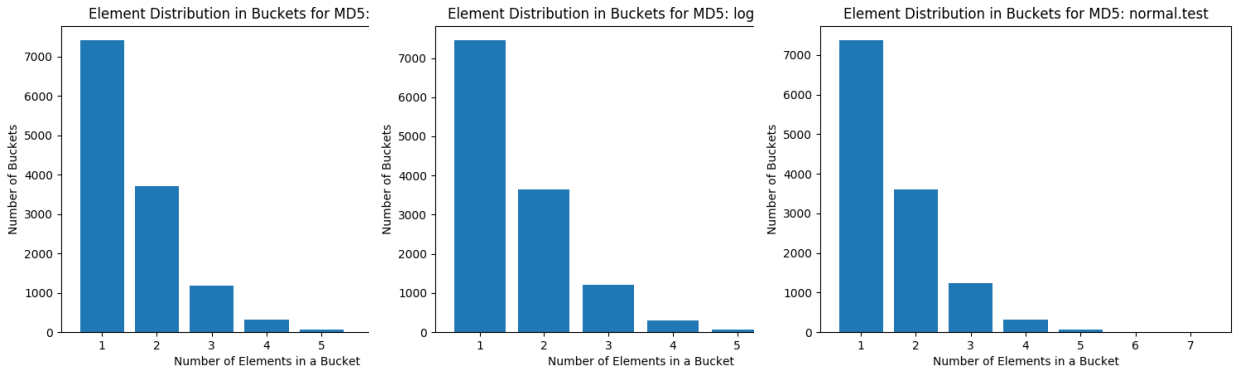
[2] Andrew Howard and Tony Jebara. Learning monotonic transformations for classification. In *Advances in Neural Information Processing Systems*, pages 681–688, 2008.

[3] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[4] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. `https://arxiv.org/abs/1712.01208`, 2018.

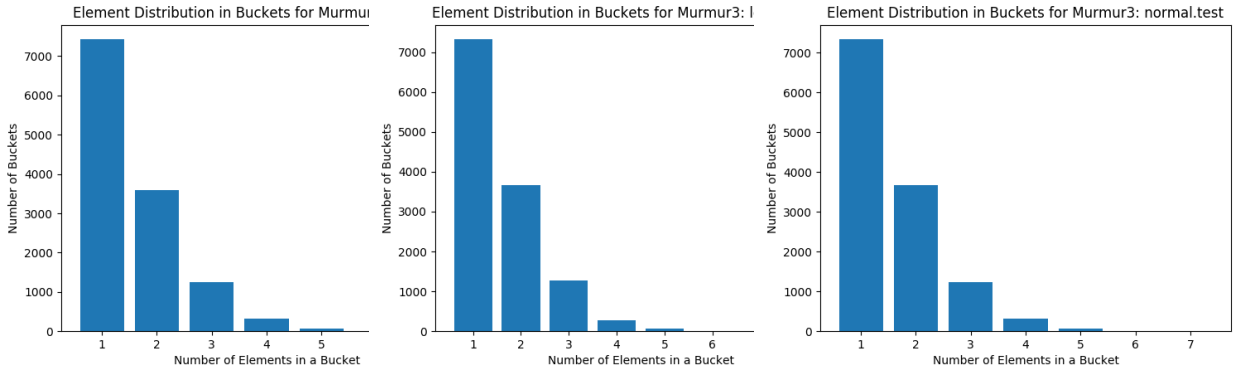
Appendix A Plots

The following plots display the element distribution across all buckets with at least 1 element in it.



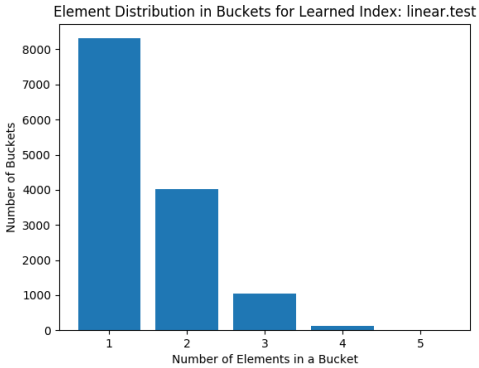
(a) On Linear Test Data (b) On Lognormal Test Data (c) On Normal Test Data

Figure 7: Element Distribution for MD5

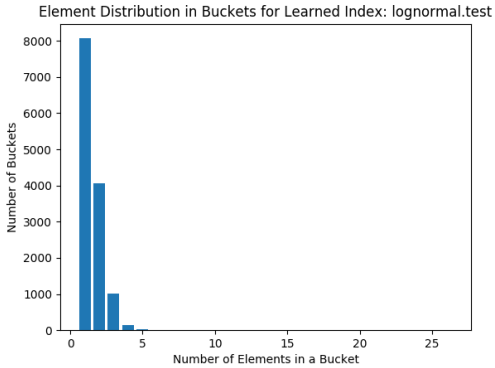


(a) On Linear Test Data (b) On Lognormal Test Data (c) On Normal Test Data

Figure 8: Element Distribution for Murmur3



(a) On Linear Test Data



(b) On Lognormal Test Data

Figure 9: Element Distribution for Learned Index

Note that for Figure 9 compared to Figures 7 to 8, we have larger bars corresponding to the number of buckets with 1 element in it (i.e. the first bar is higher for Figure 9). This implies a lower collision rate and higher bucket utilization (as we see from our tables from 3.5).