Cost Models for Learned Index with Insertions

Rasmus Bilgram, *Department of Computer Science, Aalborg University, Denmark.*
Per Hedegaard Nielsen, *Department of Computer Science, Aalborg University, Denmark*

Abstract—Machine learning models have recently been used as indexing structure. This novel approach, termed *learned index* show an increase in lookup performance compared to B-Trees. In this exploratory paper we further study the potential and limitations of this novel approach. In many real-world scenarios the data maintained is dynamic, thus new elements are inserted. We show how insertions can be supported and propose cost models to assist in decisions of when the machine learning models should be retrained and if *learned index* should be used in a given setting. We also analyzed the effect of insertions. Based on our analysis we propose structural optimizations to increase the insertion and lookup performance efficiency of *learned index*. Finally we evaluated the influence of machine learning model training and argue that overfitting makes the *learned index* less susceptible to insertions. This contradicts the objective of the machine learning models in the *learned index* structure.

Index Terms—Learned index, Insertion, Cost models, Overfitting, Mixture of Experts, Machine Learning

June, 2019

1 Introduction

Efficient data access is useful and valuable in many systems. One way to achieve this is by applying indexing structures. Indexing structures such as B-Tree, Hash-maps and Bloom Filters perform well in different contexts and can be used regardless of the dataset, which makes them uniformly applicable. In [4] by Kraska et al. a new concept is introduced, *learned indexes*. *Learned indexes* is used for efficient data access by utilizing machine learning models to learn the data pattern of a specific dataset. The traditional indexing structures as stated above are all general purpose, hence they do not exploit data patterns. In real-world scenarios, indexing structures are often specialized for various use cases, but this requires additional engineering efforts, which reduces applied attractiveness [4]. Since *learned indexes* utilize machine learning models, the concept uses each dataset's unique patterns to improve efficiency without imposing significant additional engineering efforts.

Given that *learned indexes* is a new concept there are many properties to study. An attrachtive property is insertion. In [4], insertion is at first sight referred to as the *achilles' heel* of *learned indexes*, however later argued that since the machine learning models learn data patterns it would only require $O(1)$ operations to update the *Learned index* if the insertions follow the training distribution. This argument raises several questions. In the *learned index*, overfitting is generally considered a good thing, however, Kraska et al. [4] argue that it could make the *learned index* less susceptible to inserts. This hypothesis is further studied in this paper.

In this paper we explore how insertions affect the performance of *learned index*. When inserting elements into a dataset indexed by *learned index* the insertions reduce the lookup performance of the *learned index*. Subsequently, the machine learning models in the *learned index* must be retrained to maintain efficient lookup performance. Additionally, there may be settings, where the *learned index* is not suitable due to frequent insertions. Such challenges can be accommodated in several ways, one of which is by using cost models. Inspired by query optimization where cost models are used to determine the most efficient strategy to execute a given query [11]. In this paper, we apply that approach and create estimates, which provide information to decide if a *learned index* is suitable for a given setting and if retraining should be conducted. This paper focuses on insertions and as a consequence, deletion and updates are not considered, which means that our setting is always growing. We also assume a standalone system, which implies that all training and prediction is performed on the same system.

Additionally, by analyzing the relationship between *learned index* and insertions, we are able to optimize the structure of the *learned index*, which increases the number of insertions it can handle. This creates a baseline, which ensures that further testing is fair and that the different models have a comparable starting performance independent of the data distribution trained upon.

This thesis contributes to the research domain in the following ways:

- We create cost models to assist in decisions of when retraining of the *learned index* should be conducted and if *learned index* should be used in a given setting. The former estimates the cost of retraining specific models or every model in the *learned index*. The latter estimates the number of insertions that *learned index* can handle before it has poorer lookups than *binary search*.
- We analyze the *learned index* and implement structural optimizations, which makes it capable of handling more insertions.
- We study the influence of overfitting machine learning models in regards to insertions when the data

distribution changes.

- We evaluate using synthetic data distributions to verify model accuracy. This provides an environment where we can, thoroughly, test hypotheses such as: What happens when the data distribution changes.

The rest of the paper is organized as follows: Section 2 reviews related work within this research domain. Section 3 describes our proposed cost models. Section 4 explains the experimental setup of our testing. Section 5 reports and analyzes our results. Finally, Section 6 concludes the paper and discusses directions for future research.

2 Related Work

Cost estimation is useful, because it can provide important information efficiently. The *learned index* has multiple areas to be explored.

2.1 Query Optimization

Query optimization is the process of finding the most efficient query execution plan among many depending on the query complexity. There are multiple aspects of query optimization. Some exist at the relational-algebra level, where the focus is to perform transformations that make the query more efficient. Another approach is selecting a strategy to process a query. This could be choosing a suitable algorithm to perform a certain operation based on cost estimations. The cost of an operation depends on statistics maintained in the database-system catalog and the size of its inputs. Such statistics could be the height of the B^+-Trees indices and the number of leaf pages in the indices. By using the statistics and the size of its inputs, estimates on the results of different operations can be created. The estimates may not be accurate because they depend on various assumptions that may not be upheld. Such an assumption could be all values which may occur, have equal probability. Additionally, to maintain accurate statistics they must be updated each time a relation is modified, however, such updates lead to overhead, hence most systems do not update at each modification [11].

In this study, the cost modeling approach has also been utilized to create estimates that help the learned index, handle insertions. As an example, the estimates provide information about when it is worth to re-train the models. Furthermore, the statistics are updated at each modification in order to provide the most accurate tests.

2.2 Mixture of experts

Shazeer et al. [10] discusses the concept of using *experts*, based on the original idea proposed by jacobs et al. [3]. Essentially they use a gating network that outputs an n-dimensional vector of experts. The only requirement for the experts is that they accept the same input size and produce the same output size. They also mention that if the number of vectors is large, they are able to reduce the branching factor by introducing a hierarchical model. In the hierarchical model, the experts themselves are gating networks allowing them to distribute further. Furthermore, they also take steps to balance the usage of the experts, however, this is not done in this paper. This approach remains similar to *learned index* since the models at each stage predict the models to be used at the next stage. In addition, aspects such as making the model deeper to reduce the branching factor is also a possibility.

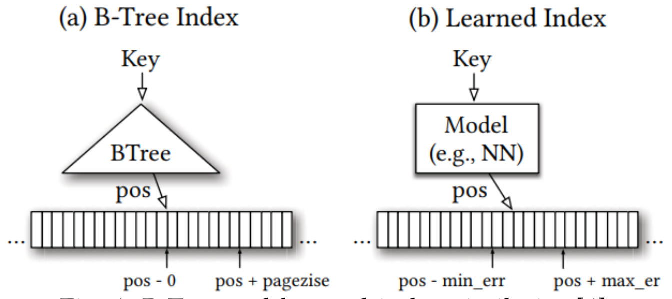

Fig. 1: B-Tree and learned index similarity [4].

2.3 Learned Index

The Case for Learned Index [4] by Kraska. et al. replace database indexing structures with machine learning models. This novel approach called *learned index* is based on the idea that indexing structures, such as B-trees can be seen as models. The indexing structure maps the key to the position of the corresponding record. As an example, B-tree ”predicts” a space in which the record exists, which can also be achieved by machine learning models. Figure 1 shows this similarity.

Using machine learning models for indexing provides low memory consumption and a possible speed up in lookups as shown by Kraska et al. [4]. Additionally, structures such as B-Trees, Hash-maps or Bloom Filters are general purpose and provide strong semantic guarantees, such as error margins and complexity. Machine learning models learn from the data and are able to exploit data patterns. To provide similar semantic guarantees as traditional structures Kraska et al. created the *recursive model index* (RMI). It is inspired by the *mixture of experts* approach using a hierarchy of models where each model takes a key as input and selects a model, expert, in the next stage. In the final stage, the models predict the position of the record with an error bound. Figure 2 illustrates the RMI and the path of predicting a position.

2.3.1 *Building the RMI*

There are multiple ways of building the RMI. It can be done bottom up, top down, using gated neural networks, etc. In this paper it is done similar to Algorithm 1 in [4], however, Algorithm 1 is unclear about how the data

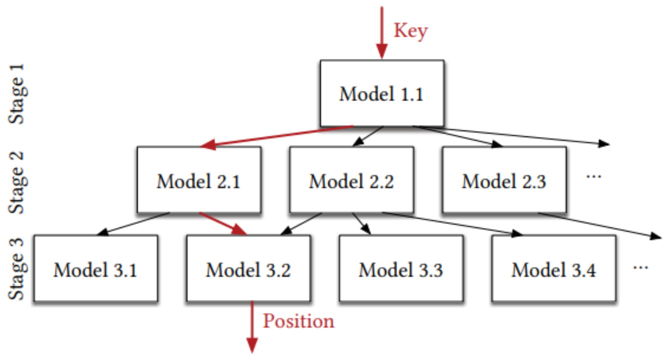

Fig. 2: Recursive Model Index [4].

is distributed. Imagine *Model 1.1* in Figure 2 the data is distributed across the models in stage 2. One could distribute the data evenly, thus each model retrieves the same amount of data. It does not seem appropriate, because some models could receive data in which they are not experts. In this paper, we distribute the data in a top down fashion by dividing the prediction of a model with a distributor d. d is created with the following formula:

$$d = \frac{D}{M_{next}} \tag{1}$$

D is the data cardinality and M_{next} is the number of models in the next stage. How M_{next} is selected is explained in Section 4.3.1. We create d assuming the data are distributed equally, but we do not restrict how the data is distributed, which solely depends on the predictions of the models. As an example, if we have 10000 data points and an RMI with two stages; the top stage with a single model and the second stage with 4 models. If the model in the top stage, the top model, predicts index 10 then we calculate the next model index by $\lfloor 10/(10000/4) \rfloor = 0$. Using this setup the top model must predict 2500 or above in order to distribute data into the model with index of 1.

2.3.2 Error

In this paper, error measures the distance from the predicted index to the actual index given a key. The error is only measured in the last stage of the RMI. The error of a model is split into a minimum and a maximum error. The minimum error is the largest underestimate of the predicted indices covered by the model, whereas the maximum error is the largest overestimate. Figure 3 represents the area covered by the minimum and maximum error of a model based on a prediction.

E, the average error of the RMI, is calculated by summing the minimum and maximum error, which for each model is multiplied by the number of elements covered by a model. The multiplied values are summed and then divided by the data cardinality. This calculation

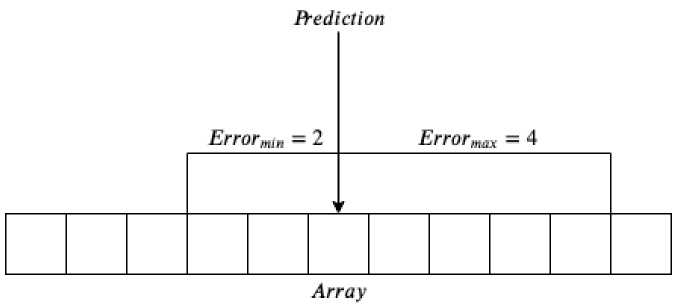

Fig. 3: A prediction and the area covered by the minimum and maximum error.

is shown in Equation 2.

$$E = \frac{1}{D} \sum_{m=1}^{M} D_m * (Error_{min} + Error_{max}) \tag{2}$$

D_m is the number of data points used to train a given model. We multiply with D_m to get the most accurate E since the data is not necessarily evenly distributed. Therefore the occurrence of each model must be considered. We assume that each element has an equal chance of being accessed, which means that the probability of accessing a model depends on the amount of elements it covers. As an example, if we have two models with a combined data cardinality of 100 and first is trained on 25 elements with an error of 3, while the second is trained on 75 elements and has an error of 12. If we do not consider the number of elements covered by each model, we get $E = (3 + 12)/2 = 7.5$ which is inaccurate in the average case since 12 occurs 3 times as often as 3. Using Equation 2 we get $E = (3*25+12*75)/100 = 9.75$, which we consider a more accurate result. We consider this more accurate because there will be more cases where the model with error 12 occurs. Compared to the cases where the model with error 3 is used.

Besides the average error E there also exists a maximum and minimum error across the entire RMI. When elements are inserted, it is probable that a single model receives more elements compared to the other models. This depends on the distribution of the inserted data and the data used for RMI construction. Potentially, the maximum error across the entire RMI could increase equal to the amount of inserted elements if a single model received every element. This aspect also influences the performance of the RMI, as lookups in such models are slower, due to the larger error. In this paper we focus on the average case, hence this problem is a topic for future work.

2.3.3 Prediction based binary search

In order to ensure we find the actual index, we employ the *prediction based binary search*. This initially checks if the key is located at the predicted index and if it is, the element is returned. Otherwise, it checks whether or not the key in the array is greater or smaller than

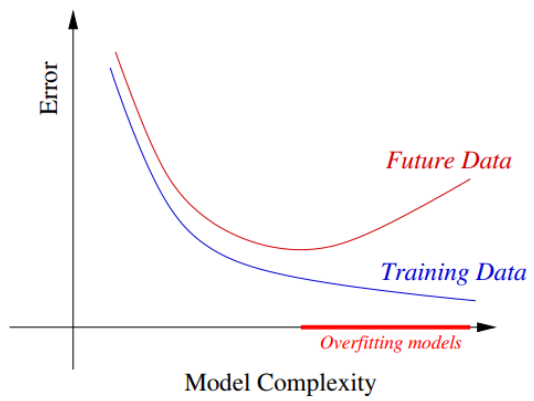

Fig. 4: Overfitting [6].

the searched key. If it is greater a left subset is searched otherwise a right subset is searched. These subsets are based on the minimum and maximum error of the model performing the prediction and searched using *binary search*. As an example, the subsets of a model with a minimum error of 2 and a maximum error of 3 has a cardinality of 2 and 3 respectively.

2.3.4 Overfitting

In machine learning, overfitting happens when a model makes predictions based on regularities that occur in the training data but not in the test data. This means that the model learns the training data instead of learning from the training data [9]. Consequently, a less overfitted model makes more accurate predictions on new data. This relationship is visualized in Figure 4, where the overfitted models gain error on future data.

As explained in [4] B-trees are good at overfitting the data, which makes them an excellent indexing structure. By replacing B-trees with machine learning models the objective of overfitting the data still stands. This contradicts the machine learning theory that overfitting generally should be avoided. However, a well performing RMI can still be created using models that do not overfit, such as linear regression. This is due to the fast execution of linear regression, hence it becomes a trade off between execution time and accuracy, but that is beyond the scope of this paper. Kraska et al. [4] made the hypothesis that there is a trade off between the generalizability of the model and the last mile predictions. This is a matter of overfitting and this hypothesis is further investigated in this paper.

3 Cost Models

The cost models for different metrics are presented and explained. The primary notations used in this paper is shown in Table 1.

Notation	Definition
τ	Cost of step in binary search
β	Cost of model prediction
γ	Cost of retrieving a model
α_{train}	Average cost of training a single data element
$\alpha_{prediction}$	Average cost of prediction
H	Height of the RMI
M	Number of models in last stage of the RMI
M_{total}	Total number of models in the RMI
E	Average error of the RMI
D	Number of elements in data (data cardinality)
D_{path}	Number of data elements in RMI path
D_{new}	Number of new data elements

Table 1: Notations.

3.1 Lookup

We create a cost model for lookup to quickly determine the lookup performance of the RMI. Additionally, it also provides an overview, of what is impacted when additional data points are added to the data, which is used to create other estimates. Finally, it also gives insight about when the RMI is not performing adequately, and should be retrained.

The cost of lookup using the *learned index* can be divided into two parts. The first part is the cost from the RMI and the second part is the last mile search. In the RMI part, we must first retrieve the required model in order to perform predictions. Retrieving the model and predicting with it has a constant cost, which are executed at each stage. The last mile search in the paper is done using *prediction based binary search* and the number of steps performed by *prediction based binary search* is $log(N)$ where N is the error of a given model. We are to calculate the cost, hence the number of steps performed by *prediction based binary search* should be multiplied with the cost of a single step. Adding these two parts we get the following cost formula:

$$cost_{lookup} = H * (\beta + \gamma) + log(E + 1) * \tau \qquad (3)$$

H is the height of the RMI, β is the cost of model prediction and γ is the cost of retrieving the model. E is the average error as shown in Equation 2.

Using the RMI as shown in Figure 2. We assume $\beta = 14ns$, $\gamma = 13.5ns$ $E = 15$ and $\tau = 10ns$. β, γ, β and τ depend on the hardware, however β also depends on the model type, which in this paper is linear regression. Linear regression is used because it has fast training and prediction. The estimate then becomes $cost_{lookup} = 3 * (14 + 13.5) + log(15 + 1) * 10 = 122.5$.

3.2 Training

A cost model for retraining can be useful in order to determine if retraining should be done. As an example, if an actual system used *learned index* as indexing structure and the lookup performance was not satisfactory, a training estimate could provide value to the decision of retraining. Finally, the cost of building the RMI is the same as retraining every model in the RMI.

3.2.1 Building the RMI

The cost of creating the RMI is based on the cost of training and prediction. First, every model in each stage must be trained and the sum of the trained data points of a stage is equal to the data cardinality D. This means that by multiplying the cost of training a single element by the height of the RMI and D, we get the total cost of training the models. Second, in order to build the RMI each data point must be predicted to determine, which models cover certain data points. The prediction is done by each model in every stage except the last stage because there are no further models to be selected. Finally, the prediction part can be calculated by multiplying the cost of a single prediction with D and the height H subtracted by 1. Combining the training and prediction part we get the following formula to estimate the cost of building the RMI:

$$cost_{full\_retrain} = (\alpha_{train} * H * D) + (\alpha_{prediction} * D * (H-1)) \quad (4)$$

α_{train} is the average cost of training a single data point and $\alpha_{prediction}$ is the average cost of predicting a single data point. α_{train} and $\alpha_{prediction}$ is shown in Equation 5 and 6 respectively. D is the data cardinality and H is the number of stages in the RMI. We use an average cost of training a single data element because analysis has shown that the cost of training a single model on D is approximately the same as training two models on $D/2$ each.

α_{train} is calculated by the average cost of training every model in the RMI. We multiply D with H because this is done across all stages. $cost_{train\_actual}$ is the combined cost, it took to train the RMI initially.

$$\alpha_{train} = \frac{1}{D * H} \sum_{m=1}^{M_{total}} cost_{train\_actual} \quad (5)$$

$\alpha_{prediction}$ is determined by the average prediction cost of a single model on all data. $cost_{prediction\_actual}$ is the cost for a model, to predict on every element.

$$\alpha_{prediction} = \frac{1}{D} \sum_{d=1}^{D} cost_{prediction\_actual} \quad (6)$$

$cost_{train\_actual}$ and $cost_{prediction\_actual}$ also depend on the hardware. To calculate α_{train} we sum the cost of training all models and then divide by all data points multiplied by H. As an example, if we have a data cardinality of 10000, an RMI with $H = 2$ and a total $cost_{train\_actual}$ of 332.167.625. The estimate then becomes $\frac{1}{D*H} * 332.167.625 = 11072$. $\alpha_{prediction}$ is calculated in the same fashion except it is done by summing the prediction cost of a single model and then the average is determined.

3.2.2 Retraining

Retraining the RMI can be done by retraining every model or only specific models. Retraining every model has the same cost as building the RMI as shown in Equation 4, and is referred to as *full retrain*. It is not always necessary to train every model instead models can be selected based on a threshold. These models are trained on the new data which they cover in addition to the data they trained on previously. This is referred to as *partial retrain*. Using this intuition we get the following estimate:

$$\begin{aligned} cost_{partial\_retrain} =& \alpha_{train} * (D_{path} + D_{new}) \\ &+ (H * \beta * D * 2) \end{aligned} \quad (7)$$

D_{new} is the number of new data points to be used for training and D_{path} is the former training data of the specific models. This way each model that is to be retrained is trained on both old and new data in order to make it more robust. The remaining cost accounts for finding the elements each model was initially trained on and recalculating the average error E. The threshold property is set to 32 in order to retrain a significant number of models. The threshold property is an interesting topic to further research.

3.2.3 When to retrain?

When inserting elements the error E of the RMI increase. At some point, it becomes so large that *binary search* on all data has a lower cost than using the RMI. Deciding when to retrain is based on Equation 3 and the cost of using *binary search*. We consider *binary search* because if it has faster lookups than the RMI we should replace the RMI with a B-Tree as explained in [4]. The idea is to compare the reduced steps of *binary search* and the last mile search using the RMI.

In other words, if we subtract the *prediction based binary search* cost of the RMI from the *binary search*, the remaining cost of the RMI must be less than the remaining *binary search* cost. In short, $cost_{lookup}$ must be less than the cost of using *binary search* on D in order to use the RMI. This constraint is shown below:

$$cost_{lookup} > log(D) * \tau \quad (8)$$

$cost_{lookup}$ is equal to Equation 3 and $log(D) * \tau$ is the cost of *binary search* on the entire dataset. Using this constraint one can achieve suitable RMI lookup performance if the lookup performance is to be as fast as possible, retraining should be done by each insertion. The constraint can also be represented as a decision tree as shown in Figure 5.

3.3 Insertion

With *learned index* being a novel approach it can be difficult to determine whether to use it or not. There are many aspects which should be considered one of which is insertions. We propose an estimate which indicates the number of insertions an RMI can handle, before it should be retrained to maintain a low $cost_{lookup}$. This could help decide whether or not *learned index* should be used.

This estimate is constructed based on the lookup estimate and multiple assumptions. We assume that the

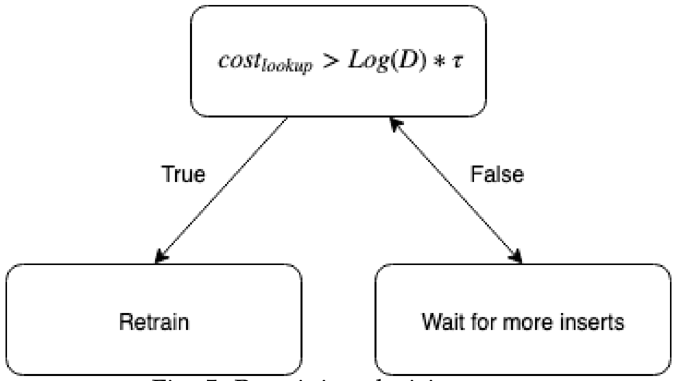

Fig. 5: Retraining decision tree.

RMI can learn the data distribution and that there is a correlation between the average error E and the amount of insertions the RMI can handle before it becomes worse than *binary search*. The latter is based on Equation 3, since it increases as E increases, which means that $cost_{lookup}$ approximate the lookup cost of *binary search*. In short, a large E yield a $cost_{lookup}$ close to the lookup cost of *binary search*, hence fewer insertions can be handled before they equal each other. Additionally, we assume that E increases by approximately 0.5 for every insert, since insertions in the dataset shifts the index of all elements to the right by 1. Also, an insert at the end of the dataset shifts zero elements. Then based on an initial RMI we can estimate, the amount of inserts required before the RMI has worse lookup performance than *binary search*. The estimate is shown below:

$$\begin{aligned} \Delta_{cost} &= log(D) * \tau - h * (\beta + \gamma) \\ steps &= \Delta_{cost} / \tau \\ E' &= 2^{steps} \\ inserts &= (E' - E) * 2 \end{aligned} \tag{9}$$

First, we calculate the Δ_{cost} between *binary search* on the entire dataset and the prediction part of the RMI. The Δ_{cost} is the cost the lookup of the RMI may increase before it is equal to *binary search*. The Δ_{cost} is then converted into *steps*, so we can calculate the average error E' the RMI must have to equal *binary search* in lookup using the inverse logarithmic function. Finally, we use the assumption that E increases by 0.5 for every insert by multiplying the Δ_{cost} between the start error E and the estimated error E'.

As an example on a dataset with a cardinality of 10000, an RMI with height $H = 2$, $E = 5$, $\tau = 10$, $\beta = 15$ and $\gamma = 15$ the Δ_{cost} are $log(10000) * 10 - 2 * (15 + 15) = 72.88$. The error E' will then be $2^{\frac{72.88}{10}} = 156.28$ and the amounts of insertions the RMI can handle is $(156.28 - 5) * 2 = 302.56$.

4 Experimental Setup

This section describes the applied datasets and how the tests are conducted. In addition, it is described how the number of models and stages in the RMI are determined.

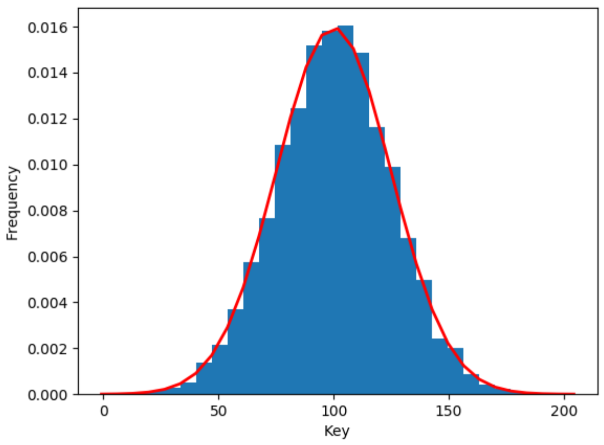

Fig. 6: Normal distribution

4.1 Hardware specifications

The tests were conducted on a machine with the following specifications:

- **CPU**: Intel(R) Core(TM) i9-9900k CPI @ 3.60 GHz
- **RAM**: 16 GB 2666 MHz

4.2 Datasets

In order to do the tests we created two different synthetic datasets; *Normal* and *Log-normal*. Both datasets are static and the names represent the distribution used for the given dataset. The distributions are also used for inserting elements to either keep or change the distribution of a given dataset. Furthermore, we assume that the index is a continuous variable.

4.2.1 Normal

The *Normal* distribution is a continuous probability distribution. It is widely used since it has many properties [7]. This often occurs in nature, as the sum of independent random variables added to a sample, tends to follow a normal distribution, this is the central limit theorem [8]. The *Normal* distribution has a shape of a bell curve as shown in Figure 6.

4.2.2 Log-normal

The *Log-normal* distribution is also a continuous probability distribution. However, in the *Log-normal* distribution, it is the logarithm of a random variable that follows a *Normal* distribution. Using the parameters, as used in this paper; $mean = 1$ and $sigma = 1$ the distribution becomes long tailed. Long tailed distributions are generally considered difficult to learn in recommender systems using traditional machine learning algorithms [13]. This is caused by different data points with low quantity. Additionally the *Log-normal* distribution has a large application area [5] [2].

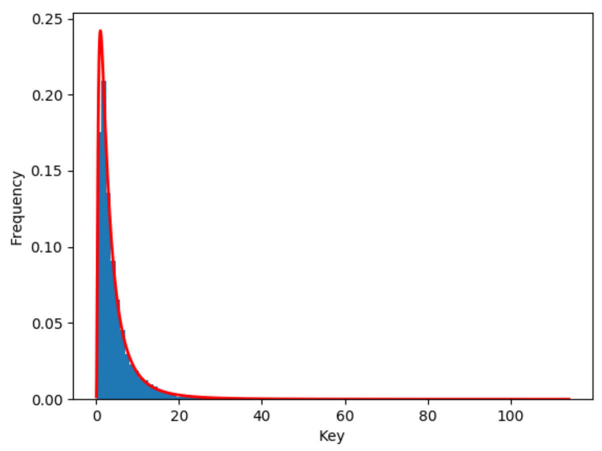

Fig. 7: Log-normal distribution

4.2.3 Insertions

All insertions in this paper are random but can be performed in two different manners. The first is a static insertion, where the insertions follow the known distribution of the dataset, thus the properties of a given dataset are approximately maintained. This means that for *Normal* such insertions do not change the *mean* or *standard deviation* significantly.

The second insertion manner is a dynamic insertion, where the inserted elements change the distribution of the original dataset. In [4] the question of the influence of such insertions is raised.

Recalculating error: Whenever elements are inserted into the RMI, the error bounds must be updated. If the error bounds are not updated, the inserted elements may not be found. This is similar to rebalancing a B-Tree, where the tree is rebalanced whenever elements are added. In every test where insertions are performed this recalculation is executed for every insert. If multiple elements are inserted before the next lookup, the recalculation of error may only occur once since the error bounds of every model are updated. The cost of recalculating the error bounds are shown below:

$$D * (H * (\beta + \gamma)) \tag{10}$$

D is the data cardinality, H is the height of the RMI, β is the cost of predicting a single element and γ is the cost of retrieving a model. This procedure could be optimized by using the RMI to find the model, which is to cover the new elements. Then only the models that cover data with larger indexes are to be updated as well. This is in the average case faster than iterating D. This optimization does not affect our results and is therefore a topic which is beyond the scope of this paper. It should be optimized if *learned index* was to be used in an actual system.

4.3 Test setup

The estimates created in Section 3 must be tested before they can be used. Every cost model is tested and various related hypotheses that provide insights about *learned index* and insertions are evaluated. The codebase is written in C++ and compiled using GCC.

4.3.1 Model and stage count

When creating the lookup and training test, we study the cost with regards to the data cardinality D. However, there are multiple variables that influence the performance of the RMI such as the height H and the number of models in the last stage M. It can be difficult to represent all these variables at the same time, which is why we made the influence of H and M constant by selecting an optimal H and M at each RMI creation. To determine these variables we analyzed the influence of H and M, in regards to the average error E, in order to determine patterns.

For optimizing H we used Condition 11 and for M we selected the number of models that provided a low E. The idea of Condition 11 is that the number of steps reduced in *binary search* when adding an additional stage must have a lower cost than the cost of the additional stage. It boils down to the number of steps performed by the *binary search* vs the cost of the additional stage.

$$cost_{lookup} > (H + 1) * (\beta + \gamma) + log(E' + 1) * s \tag{11}$$

The constraint denotes that if the current $cost_{lookup}$ is larger than the $cost_{lookup}$ with an additional stage and a different error E' we add the stage.

For M we optimized for a low E, and Figure 8 illustrates the relationship between M and E. In this analysis we included data following a *Uniform* distribution as our ideal baseline [12]. By comparing with the *Uniform* distribution we reduce the influence of model type, because linear regression fit the *Uniform* distribution well. By analyzing the graph we determined that as M increase E is decrease. This is because each model covers less data points, thus the prediction become more accurate. In our experiments we wanted the average model to cover more than two data elements, while maintaining a low E. Finally we decided to use a model count of $\frac{D}{10}$, which is where E begins to flat line with respect to M.

Empty models: Using Figure 8 we optimized for the average error E, however, new issues arose. When optimizing the number of models M for *Log-normal*, we ended up with a lookup performance similar to an RMI train on *Normal*. *Log-normal* is significantly less linear than *Normal*, hence using linear regression as model type, the lookup performance should be worse. This is caused by empty models being added since M is not suitable for the prediction accuracy of the top model. E is low because the data is still distributed across many models, although some are empty. When building the

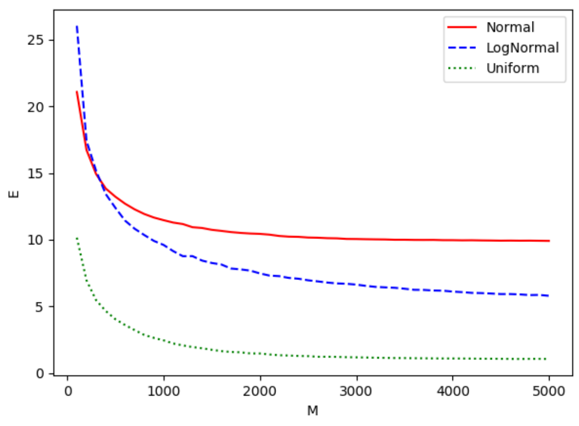

Fig. 8: The Average error E with respect to Model count M.

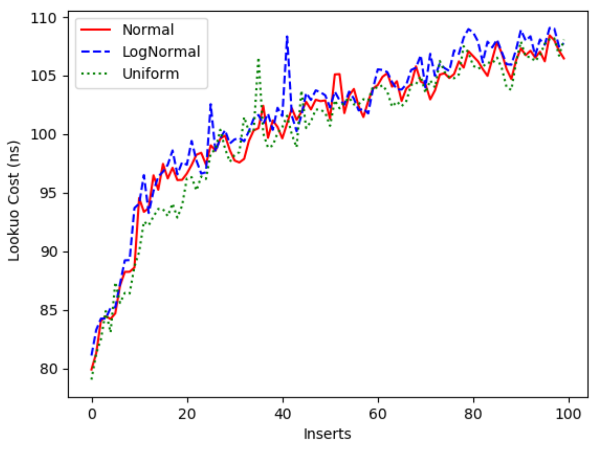

Fig. 9: Decay test. Lookup cost with respect to inserts comparing each distribution.

RMI we primarily focus on improving lookup performance with regards to H and M but we also consider the H and M with a low number of empty models.

The empty models could be removed, but then the learned index would no longer work as the indexes of the models would be shifted. This could be solved by reconstructing the RMI using the new M. This procedure was executed, but the RMI on *Log-normal* consistently had empty models at the beginning of the last stage. Consequently, the procedure was carried out recursively, but only few models would remain, thus its lookup performance was inferior to *binary search*. In addition, we gave up the advantage of having M being a specific fraction of D. Having empty models is memory inefficient, but as this paper does not have memory as a focus point, the empty models are kept to investigate the performance potential of *learned index*. Keeping the empty models improves the performance because M remains a specific fraction of D, which greatly reduces the starting error of the RMI. This does not effect Equation 1, since the manner in which the data is distributed remains the same. However, it can cause larger worst case performances and jumps in the average case, when inserted elements are distributed to the empty models. This happens because the empty models always predicts 0, then their error must cover the entire data cardinality, D.

4.4 Nanoseconds

The primary part of the conducted tests depends on time in nanoseconds. When measuring the time with such a small unit it is probable that the outliers that occur are significant. This could be solved by running the tests many times, however, that can be time consuming, when there are numerous tests.

Besides running the tests multiple times we also made each run more valid by executing the time measuring part may times, where we removed outliers by calculating the standard deviation. This yielded a vector that was used to calculate the actual average. For instance, if we were to calculate the average lookup time of an RMI, we would measure the average lookup time multiple times, remove the outliers and then return the average of the remaining times. This significantly smoothed the results.

5 Results

This section describes our results. In the tests where the lookup cost is visualized with respect to insertions, we do not plot every distribution, because the results are similar. Instead, we created a test, 5.1, to show how similar the results are.

5.1 Decay test

Figure 9 shows the lookup performance when elements are inserted. The test is conducted by constructing an RMI for each distribution, and gradually inserting elements while continuously measuring the average lookup time. In addition to the *Normal* and *Log-normal* distributions we added the *Uniform* distribution to illustrate the ideal data distribution for our models. The rates of which the lookup performance deteriorates across the distributions is similar. Since the rates are similar, it is indicated as general applicability of the RMI regardless of the data distribution. The rates also indicates other measurements such as estimates of insertions before requiring to retrain, which can be applied regardless of the data. As an example in Figure 10 we do not show every distribution, since the result is the same, due to this test.

5.2 Lookup

In Figure 10 a comparison between Equation 3 and actual lookup is visualized. This is done by creating an RMI trained on data following a *Normal* distribution

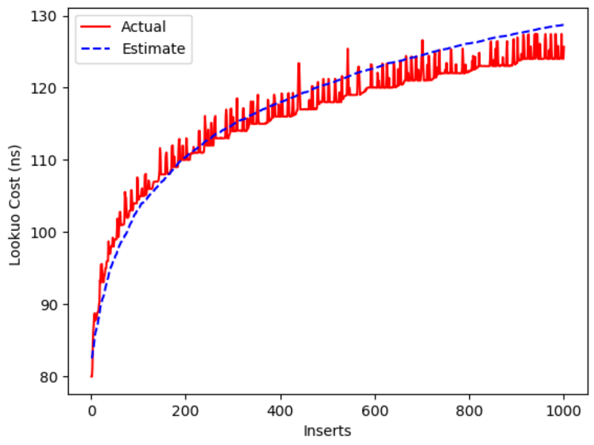

Fig. 10: Lookup test. Lookup cost with respect to inserts using data following the normal distribution.

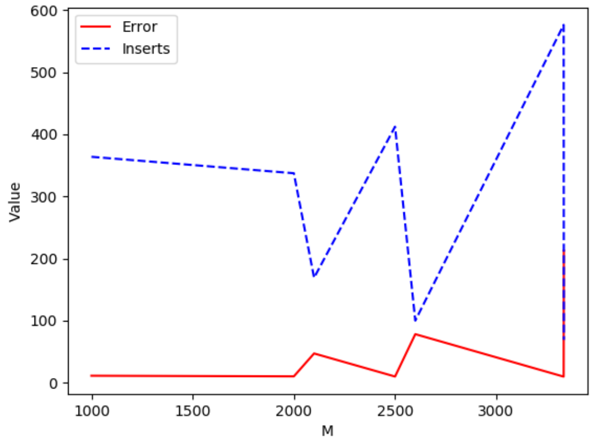

Fig. 11: Error test. Relationship between inserts and error with respect to the model count M. Tested on data following the normal distribution.

with a data cardinality of 10000. Then 1000 elements are inserted and for each insertion, the actual lookup time is measured and the estimate is calculated. For this test we calculate an overhead of the prediction to be $9.5 * H$. The overhead has the purpose of ensuring that the predicted models are not out of bounds as well as calculating the selected model index in the next stage. The estimate is consistent and only deviates slightly near the final insertions. This is an acceptable result because, as the insertions increase the more likely we are to have retrained.

5.3 Insertion

We conduct test to determine the correlation between error and inserts and show the accuracy of the estimate presented in Section 3.3.

5.3.1 Error

The insertion estimate as explained in Section 3.3 is based on various constants and a single variable. The variable is the average error E, which increases as insertions are performed. This assumption is important and must be true if the estimate is found correct. To verify it, we test if there exists a correlation between the E and insertions by creating multiple RMIs' with very different errors. These RMIs' are then tested by inserting until Constraint 8 is true. The intuition is that a small E provides more insertions than a larger E.

The test was conducted using a data cardinality D of 10000. Figure 11 illustrates the correlation between error and inserts. The y-axis is termed *Value* because it represents two different domains: Insertion count and average error. Because these domains are close the correlation is visible. The inferior models are trained on the same data as the good models. However, a high error is forced through worse distribution of data in the RMI, in order to test this premise.

The behavior of E and the number of inserts mirror each other, hence when E decreases, inserts increase, and vice versa. This confirms the hypothesis that E is an indicator of how many inserts the model can handle before condition 8 is true.

It is evident that the larger E is, a less amount of insertions can be handled. In the final peak in E, it is notable, that the inserts are less than the previous peak in E.

5.3.2 Estimate

The inserts estimation test was conducted by creating an RMI of some data cardinality D, the first being 500 and increasing at 500 with each step. Then we used Equation 9 to estimate the amount of insertions that the RMI could handle before Constraint 8 is true. The estimated insertions were performed and the actual average lookup cost was measured. The lookup cost of the RMI should then equal the lookup cost of *binary search* on D, post inserts.

Figure 12 shows the comparison of the lookup cost using the RMI after the insertions and the lookup cost using *binary search*. The estimate are slightly underestimating, the actual amount of insertions we can handle. This is because Equation 9 only consider the initial *binary search*, thus the increase in cost as more elements are added are not included. This could be remedied by recursively calculating the expected insert with the previous result added to the cardinality, until the increase in inserts is negligible. There are minor fluctuations in the estimate, due to the slight variations of the average error E. It is not guaranteed that E increases in the same manner as D.

5.4 Training

We show the accuracy of the two training estimates explained in Section 3.2.

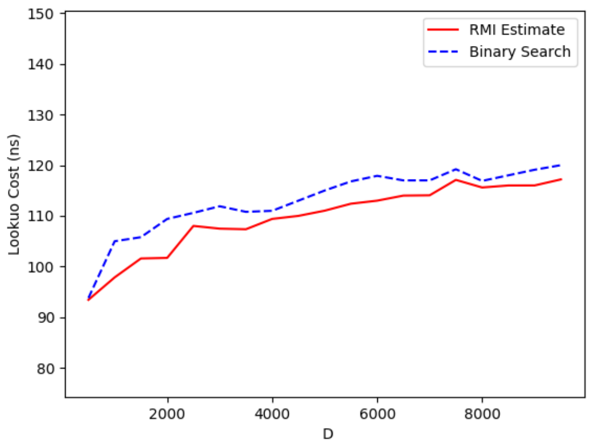

Fig. 12: Insertion estimate test. Lookup cost with respect to the data cardinality D. Tested on data following the normal distribution.

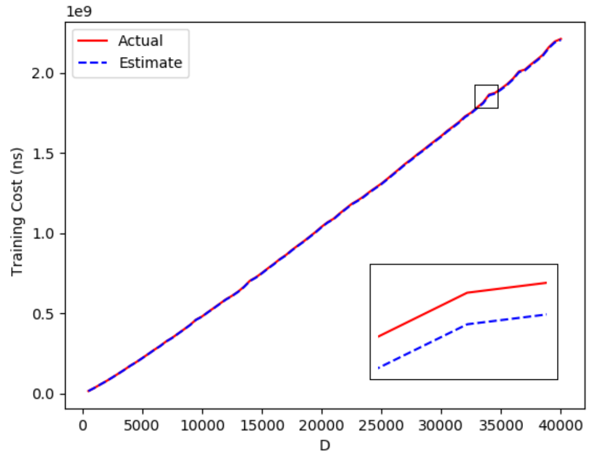

Fig. 13: Full retrain test. Training cost with respect to the data cardinality D, where the data follows a normal distribution..

5.4.1 Full retrain

The training test is conducted by training every model in the RMI, thus a *full retrain*. The results are presented in Figure 13. We use RMI configuration based on the explanation in Section 4. The training cost is increasing linearly with the data cardinality and the estimate is similar to the actual.

5.4.2 Partial retrain

The retraining test was conducted using a data cardinality of 10000. After the RMI was created, we began inserting 300 elements at each step and use *partial retrain* to retrain the RMI. The cardinality of the dataset was 25000 when all insertions had been performed. Using this workload we retrained 50 times, one after each 300th insert. The results are shown in Figure 14.

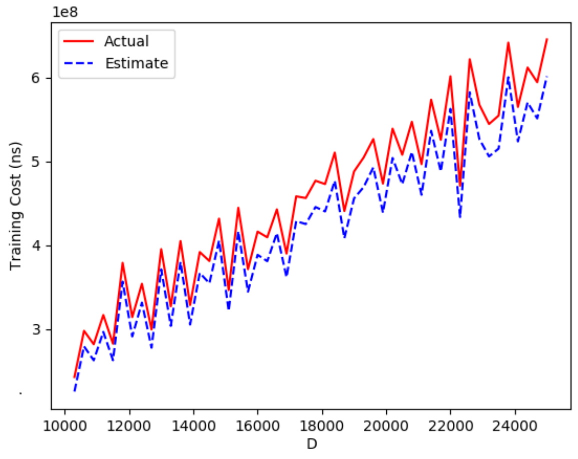

Fig. 14: Partial retrain test. Training cost with respect to the data cardinality D, where the data follows a normal distribution.

The graph fluctuates because of the number of models retrained after inserting 300 elements. Since we only train models with large error, the amount of models retrained is inconsistent. In this case, they vary because if 150 models were retrained. Then it is less probable that any of those are retrained in the next iteration, hence it is likely to be a different amount. The fluctuations could be reduced by retraining more infrequently. The estimate is similar to the actual although it is fluctuating by every retraining. Finally, there is an increasing trend, which is expected since there is more data to be retrained.

5.5 Comparison

The hypothesis states that some models become worse more frequently than others. By using *partial retrain* such models should be trained often, thus training of less influential could be spared. The *partial retrain* is compared with *full retrain* and *binary search*. The data cardinality on this test is 1000 following a *Normal* distribution. The drop in lookup time occurs when neither model satisfies the constraint described in Condition 8, hence retraining is conducted. The reason why *full retrain* drops more than *partial retrain* is that *partial retrain* only trains the last stage model which is performing poorly, therefore the increase in performance is less significant.

Figure 15 visualizes the two retraining methods and this shows that few inserts are necessary to reduce the lookup performance gained from *full retrain* compared to *partial retrain*. This is because the models retrained by *partial retrain* can handle data more efficiently since they previously received more data. Additionally insertions received by models that have a large error has little effect on the error compared to models that were freshly retrained. Although *full retrain* can receive more elements, the number of elements handled by *partial retrain* is done using a smaller retraining cost. Considering the

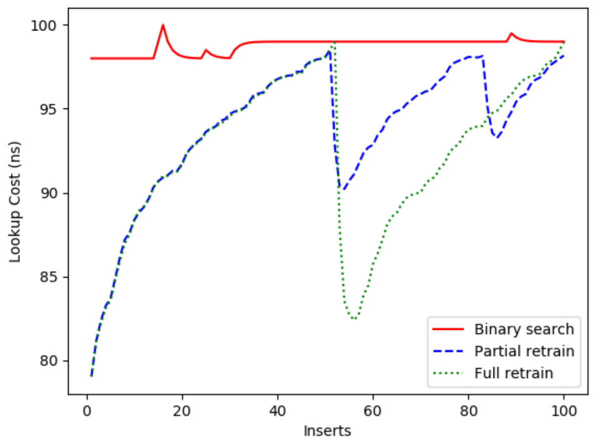

Fig. 15: Lookup comparison test. Lookup cost with respect to inserts to compare retraining methods. Conducted on data following a normal distribution.

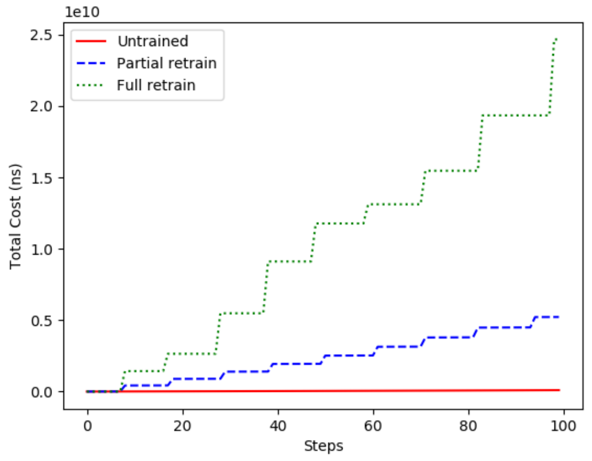

Fig. 16: Total cost test. Total cost with respect to steps on data following a normal distribution.

Cardinality	10000		1000	
Training	Overfitted	General	Overfitted	General
Normal	19.74137	19.6166	19.7888	19.79623
Log-normal	19.79887	19.67453	19.7885	19.77517
Uniform	19.7546	19.66967	19.7356	19.7688

Table 2: Lookup performance of RMI with neural network as top model.

retraining cost, *partial retrain* has the advantage in terms of handling inserts, which Section 5.6 describes.

5.6 Total cost

As seen in Section 5.4.2 the retraining cost is much larger than the lookup cost. By using Condition 8 the RMI can maintain a lower lookup cost than *binary search*. This does not necessarily mean that it is worth retraining in all situations, because some may require a low total cost when using the *learned index*. This means that the reduced cost of lookups gained by retraining, multiplied by the expected number of lookups must be less than the cost of retraining.

Our analysis showed that a large amount of lookups were required since the retraining cost is major compared to the lookup cost. To visualize this hypothesis we created a total cost test by summing the retraining and lookup cost during insertions. The lookups and insertions were carried out following a *Poisson* distribution, to indicate real-world events [12]. Figure 16 illustrates the total cost of three simple approaches to handle insertions with *learned index*. A step consists of an average of 100 inserts and then an average of 100 lookups. Retraining is conducted according to Condition 8. It is clear that the untrained RMI has the lowest cost, thus *learned index* would be geared towards better performance in settings where retraining is possible. This could be during periods of low workload. The difference between *full retrain* and *partial retrain* is also large because *full retrain* trains models, which does not necessarily require retraining.

5.7 Generalization

As explained in Section 2 Kraska et al. formulated the hypothesis that overfitting could make the *learned index* less susceptible to inserts [4]. This is based on the theory that not overfitting increases the prediction accuracy on future data points. Based on our estimates it is primarily the average error E which influences the number of insertions that the RMI can handle, but more general models could influence the gradient of how E increases with inserts.

To investigate this hypothesis we created four neural networks each with four fully connected hidden layers using ReLU activation functions. The four layers made overfitting straightforward. The neural networks was created using Python, but imported in C++ for testing. The first two were trained on a data cardinality D of 1000, whereas the others were trained on a D of 10000. One of the neural networks for each D was overfitted and the other was general. This was achieved by either training by a lot of epochs or a few, since more epochs overfit more. The neural networks were then used to replace the top model of the RMI, which decides how the data is distributed. An overfitted top model is able to distribute the data more evenly.

The neural networks were all trained on data following a *Normal* distribution. We then created three RMIs on data following the *Normal* distribution, while using the neural networks as top model. Afterwards we inserted $\frac{1}{10}D$ data points following either *Normal*, a *Log-normal* or a *Uniform* distribution. After the insertions, we measured the average lookup time on the inserted elements only, which is presented in Table 2.

The results in Table 2, is the lookup represented in microseconds. Compared to our previous results the lookup time increases because neural networks require more computations than linear regression. When the data cardinality is 10000 the general models had the fastest lookup performance, hence the hypothesis in [4] holds in this case. We expected the overfitted neural networks to be faster on the *Normal* inserts, but that is not the case. A reason for this can be that evenly distributing data among the bottom models does not necessarily increase the performance on future data. Another explanation can be that although the data is evenly distributed, it does not necessarily mean that the linear model receiving the data, is able to construct a suitable model. However, when the data cardinality is 1000 the lookup performance between the overfitted and general models is similar. We believe this is due to the number of inserts becoming too low to make an actual difference.

6 Conclusion

In this paper we proposed cost models to estimate certain aspects of the *learned index*. We estimated lookup cost, training cost and the number of insertions the *learned index* could handle, before its lookup performance became worse than *binary search*. Our estimates has a maximum error percentage of 10%. These estimates have various use cases such as assisting in the decision of: When retraining should be conducted based on its estimated retraining cost and estimated lookup cost, since it is used in our proposed retraining constraint. Additionally, the insertion estimate supports in deciding whether or not *learned index* should be used instead of other indexing structures.

In order to handle as many insertions as possible and compare various retraining methods on different data cardinalities, this paper studied how to create a better RMI with regard to lookup. We considered the depth and width of the RMI and used those results as a baseline of how we ideally wanted the RMI structure. In this decision, we also considered empty models and ensured that the structure contained as few as possible of these while being insertion efficient. Also, we argue that the minimum number of models in the last stage of the RMI should be $\frac{1}{10}$ of the data cardinality, to ensure an efficient RMI. Finally, we investigated the question of overfitting the top model in the RMI when the data distribution changes. Our analysis showed that overfitting may reduce lookup performance compared to a more general model.

We also compared two training methods, one that retrained every model in the RMI, and one that retrained models in the last stage within some error boundary. We compared two training methods, *full retrain* and *partial retrain* with respect to lookup performance and total cost. In terms of total cost, *partial retrain* was found to be superior, while in terms of lookup performance *full retrain* had the advantage, as expected. Deciding which to use would be a design choice depending on the specific use case. We also conclude that in settings where the total cost and fast lookups is of importance *learned index* is not preferable.

6.1 Future work

The test conducted in this paper is based on synthetic data. Although we used the *Poisson* distribution in certain tests to simulate real world events, it is not sufficient. It could be interesting to further study whether the estimates are robust enough to work on a real dataset. Additionally, the estimates could also be integrated into a database system to test their influence.

Another topic that has a significant impact on insertions in regards to *learned index* is how inserts are performed. A method for doing this is delta-indexing, as used in [1], where the new elements are stored in a buffer. That buffer is then searched if the RMI fails to find the key. Delta-index would increase the average cost of lookup to $H * (\beta + \gamma) + log(E + 1) * \tau + log(N) * \tau * \frac{N}{D}$. N is the cardinality of new data elements and $\frac{N}{D}$ is the probability of using the buffer. Consequently, the average error would remain static until retraining, which would be conducted based on the cardinality of the buffer.

A different approach is to create spaces in the dataset where new elements could be stored. By using this method the error would probably increase at slower rate because more insertions would be required before the average error increases.

To train, we only used linear regression except in the generalization test. Other machine learning models could differently improve the *learned index* differently in terms of insertion handling since there is plenty of regression models to test. Perhaps the average error could be reduced and more insertions could be handled, however, that also depends on the prediction speed and accuracy of the machine learning models.

The generalization test could also be further expanded upon, since it can be essential to datasets with frequent insertions. The test conducted in this paper is brief and only on two different data cardinalities with insertions equal to $\frac{1}{10}$th of the given data cardinality. It could be interesting to test on larger data cardinalities and more insertions. The data distribution trained upon, and the distributions which the insertions follow could also be adjusted to gain more insight. Also, generalized and overfitted models could be added in every stage of the RMI.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[2] E. L. Crow and K. Shimizu. *Lognormal distributions*. Marcel Dekker New York, 1987.

[3] R. Jacobs, M. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixture of local expert. *Neural Computation*, 3:78–88, 02 1991.

[4] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[5] K. Krishnamoorthy. *Handbook of statistical distributions with applications*. Chapman and Hall/CRC, 2016.

[6] T. D. Nielsen. Lecture notes in machine intelligence, October 2016.

[7] J. K. Patel and C. B. Read. *Handbook of the normal distribution*, volume 150. CRC Press, 1996.

[8] P. Z. Peebles. *Probability, random variables, and random signal principles*, volume 3. McGraw-Hill New York, NY, USA:, 2001.

[9] D. L. Poole and A. K. Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[10] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[11] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997.

[12] C. Walck. Hand-book on statistical distributions for experimentalists. Technical report, 1996.

[13] H. Yin, B. Cui, J. Li, J. Yao, and C. Chen. Challenging the long tail recommendation. *Proceedings of the VLDB Endowment*, 5(9):896–907, 2012.

1 Summary

This study is based on the paper a Case for Learned Index Structures which propose a novel approach to data indexing by using machine learning models to learn the data distribution. From this paper as well as our pre-thesis several points of interest could be further investigated. One of these points is inserts, which the original paper describes as the Achilles heel of learned index.

This paper explores learned index with respect to insertions, but in order to conduct such investigations fairly, we create a stable baseline for a efficient learned index. This is achieved by running multiple tests in regard to what makes an optimal structure with respect to insertions.

We propose cost models that estimate the cost of the different properties of the learned index. We consider the cost of rebuilding the learned index and the cost of a single lookup. Additionally, we propose an estimate, which indicate the number of inserts the learned index can receive before it should to be retrained. In order to create that estimate we investigated the relationship between the starting error of the learned index, and the number of inserts it can handle.

To determine when a learned index model needs to be retrained, we consider its average lookup performance, which is compared with the lookup performance of binary search. We also propose an alternative training method and a cost model to estimate its training cost. This alternative training method only trains specific models on their associated data.

We show several tests to evaluate our cost models, where we compare our estimations to the actual performance of the learned index. We also test the different training methods, to study the performance differences. The lookup and training performance is better when using the alternative training method, but the learned index can handle less insertions. This is as expected because training every model in the learned index should make is able to handle more insertions. To get an overall impression of the alternative training method, we also measured the total cost of the two approaches. This was carried out because the cost of training every model is significantly larger than training specific models. In order to test the total cost of the models, the data inserts and the lookups performed follow a Poisson distribution. We also explore the rate at which the lookup performance of the learned index deteriorates, across various data distributions, as more elements are inserted. Lastly, we investigate a hypothesis regarding overfitting of models by comparing an overfitted model to a general model. This is carried out on both known and unknown data, where the unknown data follows a different distribution.

In the end, we developed accurate cost models that provide insights about various properties of the learned index. Some of these insights assist in deciding if *learned index* should be used in a given scenario. Based on our analysis we found what part of learned index that is affected by insertions. This also bring