# Accelerating B+tree Search by Using Simple Machine Learning Techniques

Anisa Llaveshi*
TUM
anisa.llaveshi@tum.de

Utku Sirin
EPFL
utku.sirin@epfl.ch

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

Robert West
EPFL
robert.west@epfl.ch

## ABSTRACT

Index performance is ever important for in-memory OLTP systems. In this paper, we propose using simple machine learning models to accelerate B+tree searches. We train a linear regression model per each index node, and use the models to predict the index nodes at successive levels. Having predicted the leaf node, we simply jump to the predicted leaf node, and do a local search to find the record. We develop an algorithm that integrates the learned models into B+tree search operation, and adaptively trains the learned models in the presence of inserts.

The results show that the proposed method can predict leaf nodes with 80% to 96% accuracy, and can reduce the search time by 33% to 57%. In addition, the proposed method is robust against inserts, and can update the machine learning models fast enough to account for the newly inserted keys and nodes.

## 1. INTRODUCTION

Index performance has ever been crucial for transaction processing systems' performance. As in-memory systems gain popularity, numerous in-memory index structures are proposed to improve in-memory OLTP systems' performance [4, 6, 7]. Despite that in-memory indices are designed for efficiently using on-chip CPU caches, the research has shown that in-memory OLTP search time is dominated by index search operations [8, 9].

Kraska et al., 2018 [3] has recently showed that B+tree index structures can be replaced by a set of recursive machine learning models, i.e., a Recursive Model Index (RMI). RMI requires finding an optimal recursive structure for a set of machine learning models, which can be used to directly jump to the final position of the search key, and hence, accelerate index search operation. While Kraska et al., 2018 [3] has shown that index search operations can be significantly accelerated by machine learning models, they assume a read-only workload, and a clustered index organization where the data is stored in a contiguous memory location.

In this paper, we take one step forward, and propose using simple machine learning models to accelerate the search operation of a B+tree that also supports inserts, and that stores data as data pages in non-contiguous memory locations. Inserts and storing data in non-contiguous memory locations brings the following challenges. Inserts deteriorate the learned models. Hence, a retraining phase is required to update the learned models. Storing data in non-contiguous memory locations prohibits using key positions directly as labels for the learned model as Kraska et al., 2018 [3] does. Hence, a labeling mechanism is needed to map search keys to leaf nodes.

We use a basic B+tree. Unlike the RMI structure that requires finding the optimal recursive model structure out of a set of complex neural network models by grid search, we use a simple linear regression model per B+tree node, and use the trained linear regression models to predict the child node that the key would follow in its search in the next level. By successively predicting the nodes in the lower tree levels, we finally predict the leaf node. We, then, simply jump to the predicted leaf node, and do a local search inside/around to find the actual record. Hence, we propose integrating simple linear regression models into the existing B+tree index, rather than replacing the B+tree index by a set of recursively connected machine learning models.

The results show that our proposed algorithm can predict the correct leaf nodes with 80% to 96% accuracy, and can reduce the search time by 33% to 57%. We make the following contributions:

- We show that linear regression with simple features such as the search key or logarithm of the search key can predict leaf nodes with 80% to 96% accuracy.
- We develop an algorithm that integrates linear regression models into B+tree search. The results show that the proposed algorithm can accelerate B+tree search by 33% to 57%.
- We develop an algorithm that adaptively trains the machine learning models in the presence of inserts.

*The work was done while the author was at EPFL.


*1st International Workshop on Applied AI for Database Systems and Applications (AIDB'19), August 26, 2019, Los Angeles, California, CA, USA.*

The results show that machine-learning-accelerated B+tree search is robust against inserts, and can update the machine learning models fast enough to account for the newly inserted keys and nodes.

# 2. LINEAR REGRESSION FOR B+TREES

This section describes the linear regression models used to accelerate B+tree search as well as the additional data structures used to integrate the learned model into the search algorithm. We use a single linear regression model per B+tree index node. A linear regression model of a node, given a search key, predicts the node that the search key would visit in the next level. Hence, a linear regression model maps search keys to a particular node in the next level. In the following sections, we describe the features and labels we use to train the linear regression models.

## 2.1 Features

We use different features for different data distributions. We examine two data distributions: uniform and skewed (Zipfian). For uniform data distribution, we use the key itself as the single feature that the linear regression model is trained by.

For skewed data distribution we use the natural logarithm of the key as the feature. The reason is that skewed data follows a non-linear relationship among the keys and the children nodes. We transform the non-linear relationship into a linear relationship by taking the logarithm of the key (and also the logarithm of the node number in the next level; see the following Section 2.2).

## 2.2 Labels

The linear regression models predict the node in the next level given a search key. However, a child node is merely a pointer to a particular location in main-memory, which is a hexadecimal number that does not follow a particular distribution.

To avoid the randomly assigned hexadecimal number, we assign a number to each B+tree index node (except the root). At every level, we number the nodes consecutively from 1 to $N$, where $N$ is the last node in that level. For uniform distribution, we directly use the node numbers as the labels. For skewed distribution, we use the natural logarithm of the node number as the label to transform the non-linear relationship into a linear relationship as explained in the previous Section 2.1.

## 2.3 Model and Mapping Table

We use two helper data structures to integrate linear regression models into the B+tree search algorithm: (i) **model table** and (ii) **mapping table**. **Model table** is a vector of vector of linear regression models. For every B+tree level, there is one vector of models keeping the linear regression models for every node in that level. As the linear regression models have a single feature, every linear regression model is composed of two doubles, one for the weight and one for the intercept of the model.

**Mapping table** maps the node numbers to the pointers which points to the memory location that the particular node resides. Linear regression models only predict the node numbers, which are artificially created labels to train linear regression models (see Section 2.2). To jump to the predicted leaf node, we use a mapping table that associates every leaf node number to the pointer of the leaf node. Hence, mapping table is a vector of pair of a 32bit integer for the node number and a double for the pointer.

# 3. SEARCH WITH PREDICTION

This section describes the algorithm that uses learned linear regression models for accelerating B+tree search. Algorithm 1 presents the algorithm. The algorithm firstly predicts the leaf node number that the key would go, and then, simply jumps to the predicted leaf node and does a search inside the node to find the record that the key belongs to.

To do that, the algorithm firstly makes successive accesses to the model table to retrieve the models of the nodes that the search would follow. For every level, the algorithm (i) accesses to the model table and retrieves the model of the node that the search is in, (ii) and then uses the retrieved model to predict the node that the key would go in the next level of search. We round up/down the outcome of the linear regression prediction to the closest integer to obtain a discrete integer number. The algorithm repeats the same procedure for the successive levels until predicting the leaf node numbers at the last index node level. See Line 2 to 6 in Algorithm 1.

Having predicted the leaf node number that the search would go, the algorithm then accesses the mapping table and retrieves the pointer that points to the memory location where the predicted leaf node resides. See Line 7 in Algorithm 1. Lastly, the algorithm accesses the predicted leaf node. It firstly checks whether the predicted node is the correct node by checking the boundaries of the predicted node. If the key is outside the boundaries of the predicted node, the search continues by following the left/right sibling nodes until the leaf node whose boundary has the key inside is found. At every iteration of the sibling search, the algorithm increments a variable called *distance* keeping the information on how many nodes off than the correct node that the algorithm has predicted. The distance information signals how well the learned models predict the leaf node. A correct prediction would result in a zero distance. See Line 8 to 14 in Algorithm 1. Having found the correct leaf node, the algorithm simply does a binary search inside the node to find the record that the key belongs to. Lastly, the algorithm returns the record pointer and the distance variables. See Line 15 to 16 in Algorithm 1. In case no record is found, a null pointer is returned.

# 4. ADAPTING TO INSERTS

This section describes the algorithm used to update the learned models in the presence of inserts. Inserts deteriorate the learned models in two ways. First, newly allocated nodes will not have a node number and a model associated with them, and hence, will not be accessible during the prediction process. Second, the children node numbers of existing nodes will be deformed due to the newly inserted children nodes in between the existing children nodes.

To mitigate the deterioration of the learned models, we implement an adaptive training algorithm that gradually updates the learned models as data is being ingested. Algorithm 2 presents the algorithm. The algorithm has two modes: (i) data collection on, and (ii) data collection off.

**Algorithm 1** Search with Prediction

```
1: Input: key
2: lin_reg_model = model_table[0][0]
3: child_node = apply_lin_reg_model(lin_reg_model, key)
4: for L = 1 to N − 1 : /* N is leaf level number */
5:     lin_reg_model = model_table[L][child_node]
6:     child_node = apply_lin_reg_model(lin_reg_model, key)
7: leaf_pointer = mapping_table[child_node]
8: distance = 0 /* misprediction distance */
9: while(key < leaf_pointer->keys[0]):
10:    leaf_pointer = leaf_pointer->left_sibling
11:    distance + +
12: while(key > leaf_pointer->keys[leaf_pointer->count]):
13:    leaf_pointer = leaf_pointer->right_sibling
14:    distance + +
15: record_pointer = binary_search(leaf_pointer, key)
16: Output: (record_pointer, distance)
```

Data collection on mode is the mode where the algorithm has decided that the existing models do not perform well, and hence, has started to re-train the existing models. Data collection off mode is the mode where the algorithm assumes that the existing models are successful enough to satisfy the search queries.

The adaptive algorithm starts in data collection off mode and it accepts both search and insert queries. The search queries are performed using the search with prediction algorithm shown in Algorithm 1. The insert queries are performed regularly without any acceleration. The algorithm keeps track of the average misprediction distance of the search queries. When the average misprediction distance exceeds a certain threshold, the algorithm switches to data collection on mode. In this mode, the machine learning models are updated through a retraining process that runs as a separate thread. During retraining, the algorithm continues serving for both search and insert queries in the same way it has been serving before the retraining phase. However, inserts that require splits are kept in $overflow_buffer$s to avoid node splits that can interfere with the retraining process (see Section 4.2 for more details). An $overflow\_buffer$ is a per-node buffer keeping the inserted elements that require a split for that particular node during the retraining process. All the elements in the $overflow_buffer$s are inserted into the tree right after the retraining process as regular inserts to the B+tree. Having cleared the $overflow_buffer$s, the algorithm then continues serving the search and insert queries in data collection off mode. Next, we explain details of each mode.

Note that accelerating insert queries is outside the scope of this paper. However, accelerating insert queries would be similar to accelerating search queries as an insert query does a search before the insert.

## 4.1 Data Collection Off Mode

Algorithm 2 starts with data collection off mode. During data collection off mode, the algorithm firstly checks if $overflow\_buffers$ is null or not. Having non-empty $overflow_buffer$s shows that retraining phase has just finished, and there are some left-over records from the retraining phase that need to be inserted to the tree. To do that, the algorithm uses a hash table, named as $overflow\_buffer\_nodes$, that keeps track of which nodes possess an $overflow\_buffer$. Based on whether the $overflow\_buffer\_nodes$ hash table is empty or not, the algorithm inserts the left-over records from the $overflow_buffers$s into the tree bringing the tree to its latest form. See Line 3 to 7 in Algorithm 2.

Next, the algorithm runs search queries by the search with prediction algorithm presented in Algorithm 1. The algorithm, then, updates the two variables keeping the total distance and number of searches. These two variables are later used to decide whether the learned models perform well enough (see Line 16). See Line 8 to 12 in Algorithm 2.

Finally, the algorithm runs insert queries by regular B+tree inserts followed by a condition check to decide whether the algorithm should switch to data collection on mode, and retrain the regression models. The condition is simply checking the average distance obtained by the division of the total distance by the number of searches. If the average distance is greater than a certain threshold, the algorithm assumes that the quality of the learned model is not good enough. Hence, if the algorithm decides on switching to data collection mode on, it spawns a new thread running a retraining algorithm in parallel to the query execution, and resets the intermediate variables. See Line 13 to 19 in Algorithm 2.

## 4.2 Data Collection On Mode

Having switched to data collection on mode, the algorithm keeps running search and insert queries in a similar way to that of the data collection off mode. The only difference is that inserts are prohibited to cause node splits during retraining. The reason is that a node split that is concurrent with the retraining of the same node can clash with the retraining. Hence, to satisfy insert queries during retraining, we keep an overflow buffer per leaf node, into which we insert the records when the node is full, and hence, prevent the node being split. Similarly, search queries checks the overflow buffer if they cannot find the record in the node itself. See Line 21 to 24.

Having retraining finished, and data collection mode is turned off, the algorithm inserts the records of the overflow buffers into the tree before starting to satisfy search and insert queries. See Line 3 to 7 in the Algorithm 2.

**Retraining:** Retraining requires finding the newly inserted nodes and training models for them as well as retraining the existing models with the new set of keys that the nodes include. Retraining includes three main tasks for each tree node:

- Updating the node numbers used as labels.
- Collecting training data and training a new regression model.
- Updating the mapping and model table entries.

These tasks are efficiently combined such that a single traversal of the tree is enough to complete the three tasks. We use classic breadth-first search for traversing the tree. At the visit of each node, the node numbers are updated. In addition, during the traversal of the leaf-level nodes, the new mapping table is created, and also a set of training keys are collected. The training keys are later used to generate the training data (see below). We collect training keys uniformly at random once at every leaf-level node making sure that there is enough training data for every non-leaf-level nodes.

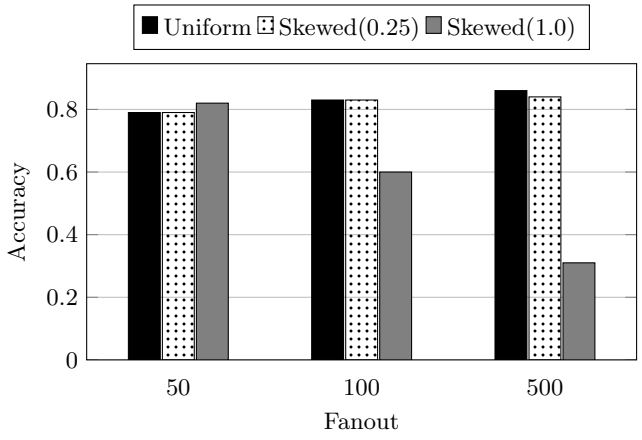


**Figure 1: Prediction accuracy for leaf level for synthetic datasets as fanout increases.**

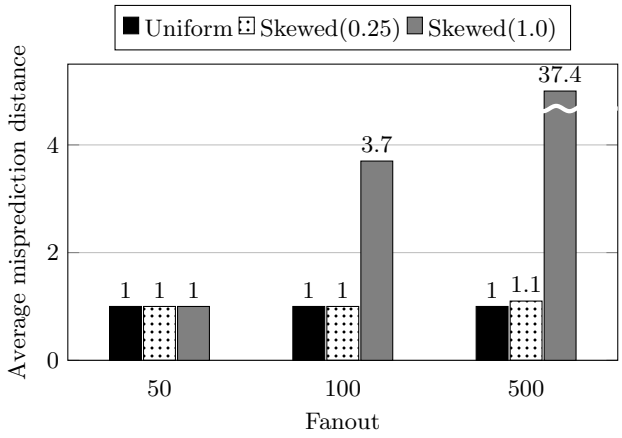


**Figure 2: Average misprediction distance at leaf level for synthetic datasets as fanout increases.**

Having traversed the tree, training data generation process starts. The training data generation process performs a search query for the each collected key (see above), and saves the key and node number for the nodes that the search traverses. The training data is kept in a per-node vector of pairs, where each pair is a key-value tuple. Lastly, retraining process uses the generated training data to create a new linear regression model for each node.

To be able to satisfy the search queries during retraining, we keep using the old mapping and model table. We create a new mapping and model table during retraining. At the end of retraining, we update the mapping and model tables to the new ones.

# 5. EXPERIMENTAL SETUP AND METHODOLOGY

This section describes the experimental setup and methodology.

**B+tree:** We use a regular B+tree. Index nodes contain keys, child pointers, and sibling pointers, whereas leaf nodes contain records (including keys), and sibling pointers. A record is composed of a key and a value, where both key and value are unsigned 64bit integers. Each index and leaf node have a node number (used as label for training) kept in an unsigned 32bit integer. We evaluate with three different fanouts: 50, 100 and 500.

**Algorithm 2** Adaptive Training Algorithm

```
1: Input: query
2: if (DATA_COLLECTION_MODE == OFF):
3:     /* insert left-over records from overflow buffers */
4:     if( overflow_buffer_nodes! = NULL):
5:         for node in overflow_buffer_nodes do:
6:             for (key, val) in node->overflow_buffer do:
7:                 insert(key, val)
8:     if (query_type == SEARCH_QUERY) :
9:         /* search with prediction in Algorithm 1 */
10:        (rec_ptr, dist)=search_w_prediction(query.key)
11:        total_dist+ = dist
12:        num_search + +
13:    if (query_type == INSERT_QUERY):
14:        rec_ptr = insert(query.key, query.val)
15:        /*condition check for deciding to retrain or not*/
16:        if( total_dist/num_search > av_dist_threshold):
17:            DATA_COLLECTION_MODE = ON
18:            thread t(retrain, this)
19:            av_dist, num_search = 0
20: if (DATA_COLLECTION_MODE == ON):
21:    if (query_type == SEARCH_QUERY):
22:        (rec_ptr, dist)=srch_w_pred_w_buffer(query.key)
23:    if (query_type == INSERT_QUERY):
24:        rec_ptr = insert_w_buffer(query.key, query.val)
25: Output: rec_ptr
```

**Datasets:** We use three synthetic and one real-world datasets. The synthetic datasets are composed of one uniformly distributed, and two skewed datasets following a Zipfian distribution of a factor 0.25 and 1.0. All the datasets contain 300 millions of keys.

We use the IMDB movie dataset as the real-world dataset. In particular, we use the main *title* table, and use its keys to index [5]. The dataset contains 2.5 millions of keys.

For all the datasets, we initially bulkload the data. Before starting each experiment, we train models for each node, create the mapping and model tables.

**Training data collection:** To collect training data, we select a set of keys by going through each leaf node and picking one key from each leaf node uniformly at random. Next, we query each of the keys, and save the key and node numbers of the nodes that the search traverses. Lastly, we train a linear regression model for each index node with the collected (key, node number) pairs.

**Machine learning model evaluation:** We perform 10-fold cross validation for evaluating the machine learning models we trained. On synthetic data, we run the accuracy experiments with 10 test sets of 1 million keys and on the real-world data we run them with 10 test sets of 10000 keys. The keys are chosen uniformly at random from all keys in the tree. The test sets are mutually exclusive with the training set and among each other. We, then, report the average of the 10 repetitions.

For the performance evaluation, we run the experiments on a test set of 1 million keys on the synthetic data and 10000 keys on the real-world data. The keys are chosen uniformly at random among all keys in the tree and mutually exclusive with the training set.

**Workload:** For the accuracy and performance evaluations we run 100% search queries as we accelerate only search queries. For the evaluation of adaptivity, we initially bulkload the tree as before, and then start doing 100 thousands of searches and inserts in an alternating order. Hence, the workload is 50% searches and 50% inserts. We select the keys to insert from the same distribution of the B+tree data. We use average distance threshold of 1 to decide whether start retraining or not (see Section 4.2, Algorithm 2, Line 16).

**Metrics:**

- **Accuracy:** Ratio of the number of correctly predicted nodes to the total number of predictions.
- **Average distance:** Average distance between the predicted node and the node that the key actually resides in. If the node the key resides in is the sibling of the predicted node, the distance is 1. If the prediction is correct, the distance is 0.
- **Average misprediction distance:** A similar metric to average distance, except that it averages distances only for mispredicted nodes.
- **Standard deviation of misprediction distances:** Standard deviation of the distances for the mispredicted.
- **Search time:** The amount of time for performing a search query. As a single search query is in the orders of few micro-seconds, we report search time as per 10 thousands of keys in milliseconds.

**Server:** We run our experiments on a typical Intel Xeon server processor that database systems run on with 256GB of main memory. It has Ivy Bridge micro-architecture featuring 32KB of split L1 instruction and data caches (per core), unified 256KB of L2 cache (per core), and 20MB of shared L3 cache. It has two sockets, where each socket has eight cores. It features two hyper-threads per core, but we disable hyper-threading to obtain more precise performance behavior.

**OS & compiler:** We run all the experiments using Ubuntu 16.04.6 with Linux kernel version 4.15.0-46-generic. We use Intel(R) Math Kernel Library 2019 Update 3 for Linux for the logarithm and exponential calculation because it resulted in being $\sim$ 35% faster than the functions from the C++ Standard Library.

**Measurements:** We do all the measurement by running a single thread. We rely on OS to schedule the spawned retraining thread to be run on a separate core. Before every experiment, we populate the data and keep the data in-memory throughout the experiments. We use Linux's numactl command to run everything locally on a single numa node.

**Table 1: Standard deviation for the average misprediction distances at leaf level for synthetic datasets as fanout increases.**

| | Fanout | | |
|---|---|---|---|
| | 50 | 100 | 500 |
| Uniform | 0.03 | 1 | 0 |
| Skewed(0.25) | 0.03 | 0.02 | 1.9 |
| Skewed(1.0) | 0.16 | 4.92 | 38.87 |

# 6. EVALUATION

This section presents the evaluation using sythetic and real-world datasets.

## 6.1 Synthetic Datasets

This section presents the experimental evaluation of the proposed method on the synthetic datasets with uniform distribution, skewed distribution with a factor of 0.25, and skewed distribution with a factor 1.0 with fanouts of 50, 100 and 500. We evaluate (i) accuracy, (ii) performance, and (iii) adaptivity aspects when running the proposed algorithms.

### *6.1.1 Accuracy*

This section presents the accuracy evaluation. Our goal is to show that simple linear regression models are successful enough to predict the leaf nodes given a search key.

Figure 1 and 2 show the accuracy and average misprediction distance for the three distributions and for the three fanouts we test. Figure 1 shows that, for the uniform and skewed (0.25) distributions, accuracies are all $\sim$0.8 for all the fanout values. It means that the models 80% of the time correctly predict the leaf node for the uniform and skewed (0.25) distributions. For the remaining 20% of the time, the models mispredict the leaf node. For those mispredicted leaf nodes, Figure 2 shows that the average misprediction distance is always 1 for the uniform and skewed distribution of a factor of 0.25. Hence, the infrequent case of mispredictions are off only by 1 node from the correct leaf node. Overall, the accuracy and average misprediction results show that simple linear regression models successfully predict the leaf nodes for the uniform and skewed (0.25) distributions.

On the other hand, for the skewed (1.0) distribution, Figure 1 shows that accuracies dramatically drop as fanout increases. While the accuracy is $\sim$0.8 for the fanout of 50, it is $\sim$0.6 and $\sim$0.3, for the fanout of 100 and 500. Similarly, Figure 2 shows that the average misprediction distance dramatically increases as the fanout increases. While the average misprediction distance is 1 for the fanout of 50, it is 3.7 and 37.2 for the fanout of 100 and 500. Hence, unlike the uniform and skewed (0.25) distributions, the results show that simple linear regression models are only successful for a fanout of 50 for skewed (1.0) distribution, and fails for higher degrees of fanouts, such as 100 and 500.

Lastly, Table 1 shows the standard deviation of the average misprediction distances. Standard deviation of the mispredictions distances is important as it shows if there is any unpredictable spikes in the prediction results despite a good average behavior. Table 1 shows that standard deviations are quite low for the uniform and skewed (0.25) distributions for all the fanouts. It is always less than 2 for all the fanouts. It means that, all the mispredictions

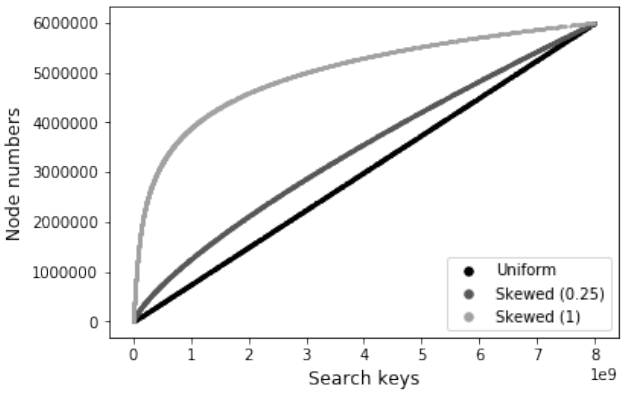


**Figure 3: Key to node number mapping for synthetic datasets.**

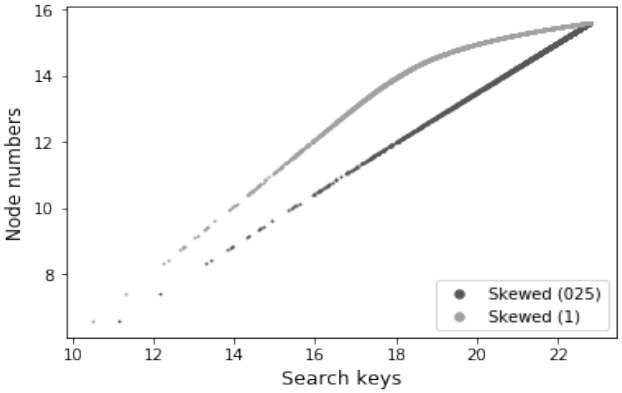


**Figure 4: Log-transformed key to node number mapping for synthetic datasets.**

are at most off by 2 nodes from the correct leaf node. On the other hand, similar to the accuracy and misprediction distance results, standard deviation dramatically increases as fanout increases for the skewed (1.0) distribution. This, once again, shows that simple linear regression models are successful only for lower degrees of fanout for the skewed (1.0) distribution.

The reason for the failure for the skewed (1.0) distribution for high fanout values is that the higher the fanout is the more skewed the data per node is. As a result, linear regression models fail to accurately model the key distributions per node. To understand why linear regression works for the uniform and skewed (0.25) distribution, but not for the skewed (1.0) for high fanout values, in Figure 3 we plot the function that maps keys to node numbers for the leaf-level. As the figure shows, there is almost a linear relationship between the search keys and node numbers for the uniform distribution. This explains why using the keys directly as features works well for the uniform distribution.

Unlike the uniform distribution, Figure 3 shows that the relationship is slightly skewed for the skewed (0.25), and dramatically skewed for the skewed (1.0) distribution. To understand how log transformation affects the key to node number distribution, in Figure 4 we plot the relationship between the log transformed key and node number. The figure shows that the key to node number mapping after the log transformation is almost a linear relationship for the skewed (0.25) distribution, and is a much less skewed relationship for the skewed (1.0) distribution. This explains why log transformation is so successful for the skewed (0.25), and fails for the skewed (1.0) for high fanout values. The relationship is always linear for the log-transformed data for skewed (0.25). As a result, linear regression models are successfully represent the data regardless the fanout. However, the key to node number relationship gets skewed towards the end of the search keys for skewed (1.0). As a result, high fanout values covering large range of key values suffer from skewed distribution inside the node, and hence, fails to be successfully modeled by linear regression.

### 6.1.2 Performance

This section measures how much the machine-learning-accelerated B+tree improves the B+tree search time. To do that, we compare the search time of a regular B+tree

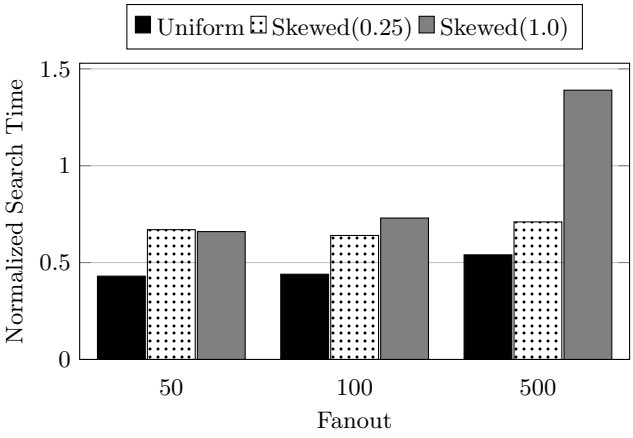


**Figure 5: Normalized B+tree search time for synthetic datasets as fanout increases.**

with that of the machine-learning-accelerated B+tree. Figure 5 shows the normalized search time for the three distributions and three fanouts we tested. Normalized search time represents the amount of time it takes for the machine-learning-accelerated B+tree search to complete compared to the regular B+tree search for each particular fanout value. For example, a value of 0.4 shows that the regular B+tree search time is decreased by 60% by the machine-learning-accelerated B+tree search.

The figure shows that the search time is decreased by 57%, 56% and 46% for the uniform distribution for the three fanouts used. Similarly, the search time is decreased by 33%, 36% and 29% for the skewed (0.25). Hence, following successfully predicting the leaf-level nodes, our proposed method significantly improves the search time for the uniform and skewed (0.25) distributions for all the fanouts.

For the skewed (1.0) distribution, the search time is decreased by 34% and 27% for fanout of 50 and 100, and is increased by 39% for the fanout of 500. This shows that our proposed method is robust enough to accelerate B+tree search for the fanout of 100 despite the modest prediction accuracy (0.6), and relatively high misprediction distance (3.7). For the fanout of 500, however, the search time significantly increases due to low prediction accuracy (0.32)

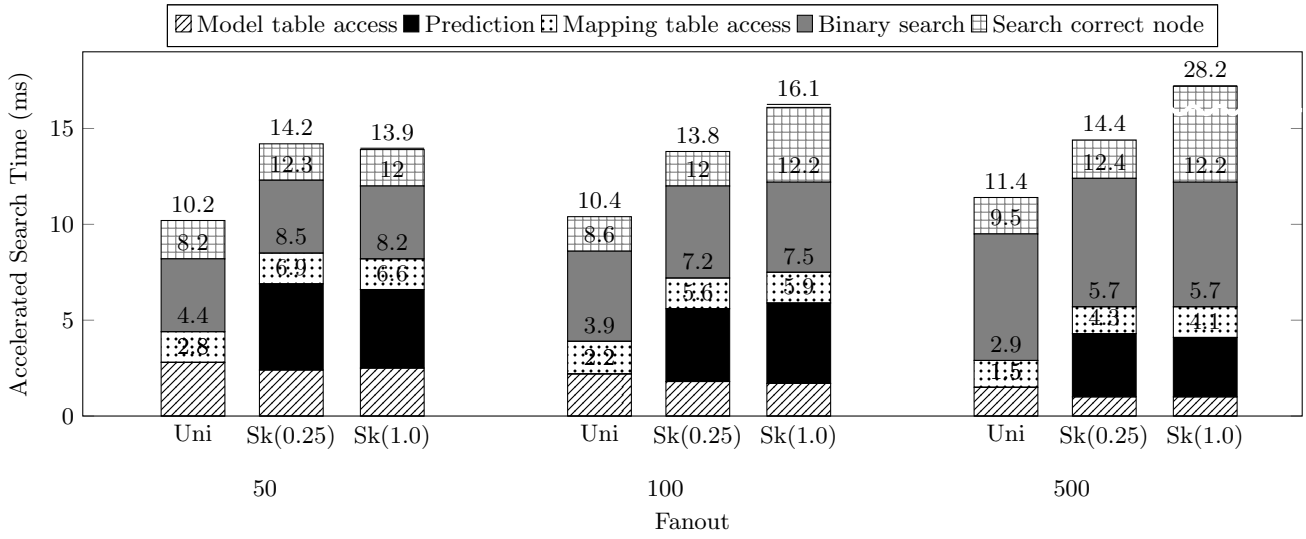


**Figure 6: Accelerated search time breakdown for synthetic datasets as fanout increases.**

and high misprediction distances (37.4).

To understand where time goes during the machine-learning-accelerated search, we breakdown the machine-learning-accelerated search time into five components: (i) model table access, (ii) prediction, (iii) mapping table access, (iv) binary search inside node and (v) search for the correct node. Figure 6 shows the results.

As the figure shows, for uniform distribution, majority of the time goes to the binary search inside the leaf node. For skewed (0.25) distribution, majority of the time is equally divided among the binary search inside the node and prediction. For uniform distribution, prediction is simply multiplying the weight with the key and adding the intercept to it. Hence, prediction is merely a multiplication followed by an addition, and takes a negligible amount of time. However, for skewed distributions, prediction requires taking the logarithm of the key and exponential of the successive predictions at each level. As we use logarithm of the node number during training, we need to take the exponential of the prediction to obtain back the predicted node number. However, taking exponential of the prediction at each level of the tree requires a serialized multiplication operations, which in the end, takes a significant portion of the accelerated search time.

For the skewed (1.0) distribution, the binary search inside the node and prediction consumes the majority of the search time for the fanout of 50. For the fanout of 100, search for the correct node becomes one of the major components in the search time due to the increased misprediction rate and misprediction distances. Lastly, for the fanout of 500, search for the correct node almost completely dominates the search time due to low prediction accuracies and high misprediction distances.

Systems usually tune their B+tree fanout values to minimize the search time. Our results showed that the search time for the regular B+tree that we use as the baseline is 23ms, 22ms and 20ms for fanouts of 50, 100 and 500. Hence, the performance difference is less than 15% for the range of fanouts we test. However, our proposed method works significantly better for the fanout of 50 than the fanouts of 100 and 500 for different types of distributions. Hence, our proposed method brings a new type of fanout tuning mechanism that requires incorporating the machine-learning-accelerated search time into the tuning process. As we obtain consistently good results for the fanout of 50, we use fanout 50 for the rest of the paper.

In summary, while machine-learning-accelerated search significantly improves the regular B+tree search time (by ∼57%) for simple distributions like uniform distribution, it improves the search time more modestly (by ∼33%) for more complex distributions like skewed Zipfian distribution due to the complexity of feature calculation. Moreover, B+tree fanout should be carefully chosen by incorporating the machine-learning-accelerated search time into the fanout tuning process.

### 6.1.3 Adaptivity

This section measures the adaptivity of the adaptive training algorithm we proposed in Section 4, Algorithm 2. To do that, we do 100 thousands of searches and inserts in an alternating order, and we measure the machine-learning-accelerated search time and the average distance over the time. Our goal is to see (i) how robust the learned models are against newly inserted keys and nodes, and (ii) how fast the adaptive training model can adapt to the newly inserted keys and nodes.

Figure 7 show the search time results. The algorithm starts with a B+tree with 300 million keys. Each point in the curves (except the first point) marks the time where 50 million inserts are finished. Hence, Figure 7 shows that the search time increases steadily up until 400 million inserts, after which the algorithm decides going into retraining phase. From the beginning until the end of the 400 million inserts the search time is increased by ∼50% for the uniform distribution, and by ∼33% for the skewed distribution, while the data size is more than doubled. Hence, the search time is increased sub-linearly indicating that the learned models are robust against the inserts. After the retraining, Figure 7 shows that search time increases even more steadily, once again, indicating the robustness of the learned models.

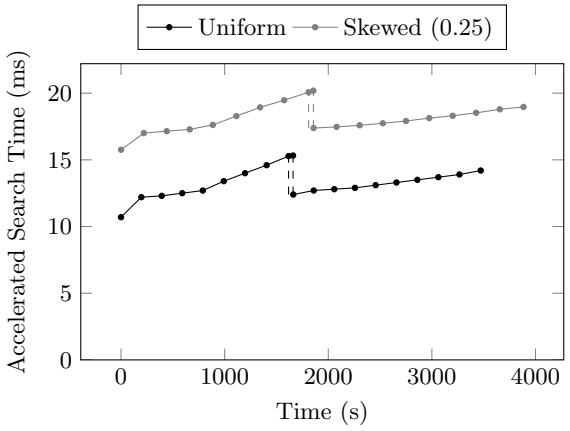


**Figure 7: Accelerated search time in the presence of inserts for synthetic datasets. Fanout of 50 is used.**

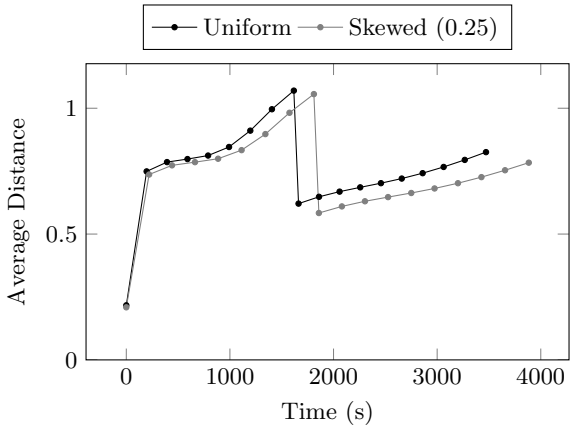


**Figure 8: Average distance in the presence of inserts for synthetic datasets. Fanout of 50 is used.**

The two consecutive dashed lines in Figure 7 marks the points where retraining starts and ends. In addition, there is a point in between the start and end of retraining showing the search time during retraining. As can be seen, retraining takes relatively a small amount of time (∼43 seconds) compared to the pace of increase in the search time. In addition, the search time during the retraining is more or less the same as the search time before retraining, which shows that the parallel retraining thread does not interfere with the actual execution of the search and insert queries.

Figure 8 shows the average distance results. Note that the graph shows the average distance results rather than the average misprediction distance. Hence, the values start close to zero and then gradually increase. Figure 8 follows a similar trend to Figure 7, and hence supports the conclusions.

On the other hand, both Figure 7 and 8 have a steep increase right after the initial 50 million inserts, and after 200 million inserts. The reason for those is that right at the beginning the nodes are all full due to bulkloading. Similarly, after 200 million inserts, nodes get full enough to start frequent node splits. As a result, there are more inserted nodes right the first 50 million and 200 million inserts, and hence, more steep increase in the average distance and search time.

In summary, inserts increase the search time and average distance sub-linearly, showing that the one linear regression model per node scheme and the adaptive training algorithm that we develop on top of this scheme is robust against inserts. In addition, retraining time is small compared to the pace of increase in the search time, and hence the algorithm can adapt relatively fast to the newly inserted keys and nodes.

## 6.2 Real-world Dataset

This section presents the experimental evaluation of the proposed method on the real-world dataset of IMDB movie titles. We evaluate (i) accuracy and (ii) performance aspects when running the proposed algorithms.

### 6.2.1 Accuracy

Figure 9 and 10 show the accuracy and average misprediction distance results for the IMDB dataset using three different fanouts. As can be seen, accuracy ranges from 0.86 to 0.96 for fanouts of 50, 100 and 500. This means that, $86\% - 96\%$ of the time, the models correctly predict the leaf

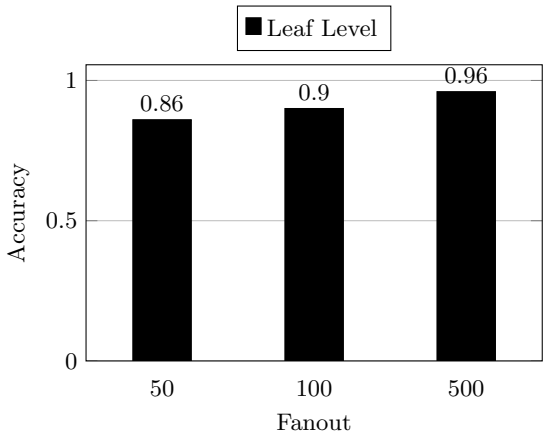


**Figure 9: Prediction accuracy for leaf level for the real-world IMDB dataset as fanout increases.**

node. For the remaining $4\% - 14\%$ of the time, the models mispredict the leaf node. For those mispredicted leaf nodes, Figure 2 shows that the average misprediction distance is always 1. The standard deviation among these mispredictions is always 0 (not shown). Hence, all the mispredictions are off only by 1 node from the correct leaf node. Hence, the infrequent case of mispredictions minimally affect the overall performance.

Unlike the synthetic data experiments where accuracy drops as fanout increases, Figure 9 shows that accuracy grows as fanout increases. The reason for that is the B+tree built on top of the IMDB dataset is much smaller than the B+tree built on the synthetic datasets. While the B+tree built on top of the synthetic datasets is in the orders of 100s of thousands of leaf nodes to a few millions of leaf nodes, the B+tree built on top of the IMDB dataset is in the orders of thousands to tens of thousands. Hence, higher fanouts for the small IMDB dataset make the prediction task significantly easier resulting in increased accuracies.

### 6.2.2 Performance

This section measures how much the machine-learning-accelerated B+tree improves the B+tree search time in the

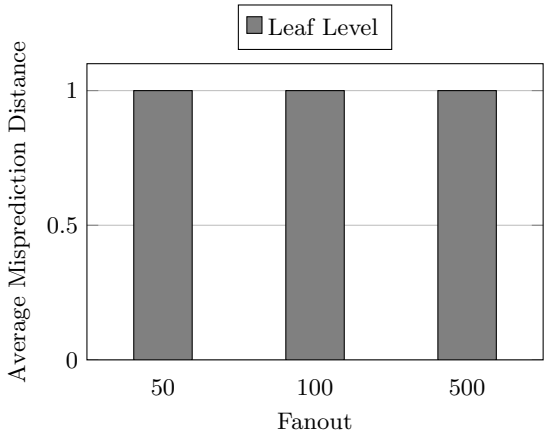


**Figure 10: Average misprediction distance for leaf level for the real-world IMDB dataset as fanout increases.**

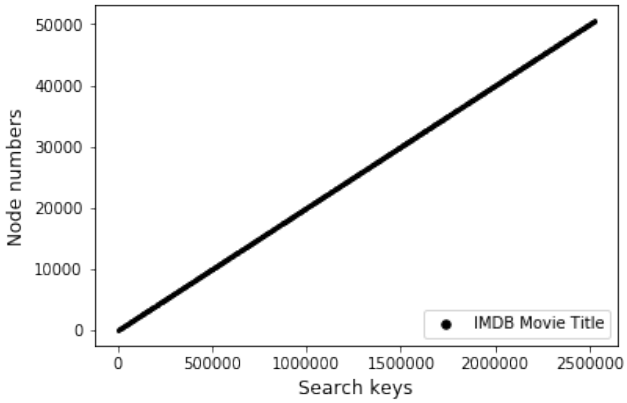


**Figure 11: Key to node number mapping for the real-world IMDB dataset.**

IMDB dataset. To do that, we again compare the search time of a regular B+tree with that of the machine-learning-accelerated B+tree. Figure 12 shows the results. It shows the machine-learning-accelerated search time normalized to regular B+tree search.

The figure shows that machine-learning-accelerated search improves the regular B+tree search by 42% to 53%. To understand where time goes during the machine-learning-accelerated search, we again break down the machine-learning-accelerated search time into five components. Figure 13 shows the results.

The figure shows that the majority of the search time goes to the binary search inside the leaf node. As expected, this time increases with the increase in the fanout. However, the overall time to perform the search overall only very slightly changes. The reason for that is the decreased model and mapping table access time. Using bigger fanouts results in smaller mapping and model tables. As a result, access time to the model and mapping tables decreases as fanout increases. Hence, the increase in the binary search time is compensated by the decrease in model and table access time, and results in having similar overall search time for different fanouts.

To understand the success of the linear regression models on the real-world IMDB dataset, we again plot the key to node number mapping for the B+tree built based on the IMDB dataset. Figure 11 shows the graph. As can be seen, the relationship between the key and node number is almost linear as is the case for the uniform distribution. As a result, linear regression successfully models the key to node number mapping, and can be used to accelerate search time by 42% to 53%.

# 7. DISCUSSION

This section presents a discussion on the set of topics that relates to our design choices for our method.

**One model per node scheme:** We use one linear regression model per node. While the results show that it works well, one model per node can cause explosion in the number of models as tree size grows. We observed that most models look like each other. Hence, in case of a large number of nodes, using less number of models per level can provide a similar level performance with less number of models. We tried one model per level, which worked well only at the upper levels.

**Leaf-node training:** We can further improve performance by predicting the exact location of the record inside the leaf node, rather than doing a binary search inside the node. This, however, would require exponentially more models as there are exponentially more number of leaf nodes than the index nodes. Moreover, leaf nodes are more prone to change in the presence of insertions making the leaf-node models more fragile against inserts.

**Adapting to changing distribution:** Adapting to changing distribution is out of the scope of this paper. However, the adaptive training algorithm is suitable to integrate such a change. A heuristic can decide on whether the used features should be log-transformed or not during the retraining, and can easily adapt to a changing distribution.

# 8. RELATED WORK

There is a large body of in-memory index works. Bw-tree proposes avoiding in-place updates to avoid cacheline invalidations. Instead, it relies on delta pages that are gradually consolidated into a single value [6]. Masstree proposes optimizes for common prefixes on dataset such as website links. In addition, it uses software prefetching on accessing wide B+tree nodes [7]. Adaptive Radix Tree extends traditional radix tree structure by supporting variable-sized nodes, and hence, mitigating high space consumption of radix trees [4].

Our proposed method is applicable to all types of in-memory indices that require search inside index nodes. Hence, all the three referred, popular in-memory index structures can benefit from our proposed machine-learning-based index search acceleration. Traditional radix trees use key as to decide which nodes to traverse. Hence, there is no need for traditional radix tree to predict any node in the search path. However, radix trees are usually used for string keys, and consume large amount of space [4].

Kraska et al., 2018 [3] has recently proposed replacing index structures with a graph of recursively connected com-

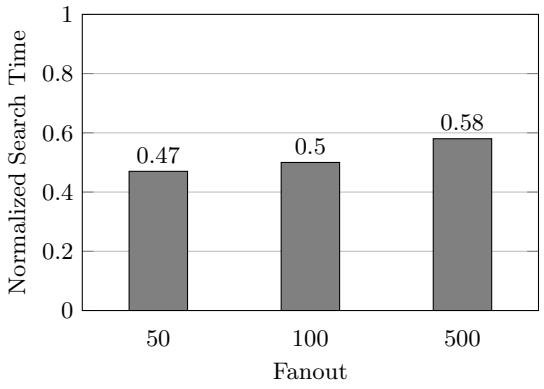


**Figure 12: Normalized B+tree search time for the real-world IMDB dataset as fanout increases.**

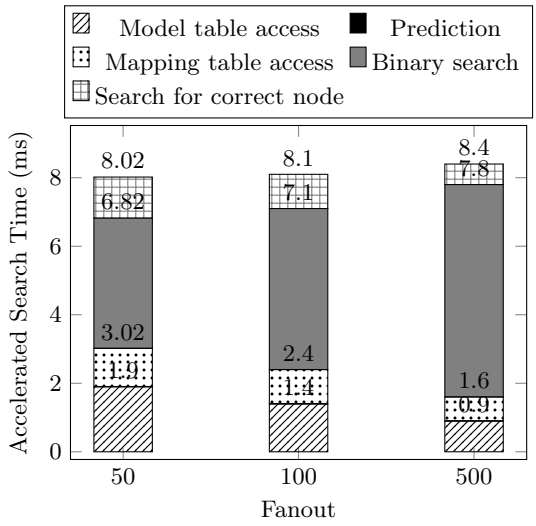


**Figure 13: Accelerated search time breakdown for the real-world IMDB dataset as fanout increases.**

plex neural network models, i.e., Recursive Model Index (RMI) for read-only workloads, and clustered index organizations where data is stored in contiguous memory locations. Similarly, Kraska et al., 2019 [2] has proposed a system architecture that learns different DBMS components by taking data, hardware and workload into account. In our paper, we discuss integrating simple linear regression models into an existing B+tree index that supports inserts, and that stores data in non-contiguous memory locations.

Moreover, we propose using a single linear regression model per B+tree node rather than finding out a separate recursive machine learning model structure through a parameter tuning process, which can be a costly process that is needed at every retraining phase. We show that using a single model per node is successful to find out the correct leaf node.

Galakatos et al., 2018 [1] has recently proposed A-Tree, which is an approximate index structure that uses a piecewise linear function for modeling the key distribution. In our work, we use a single linear function per node. Using a piecewise linear regression for modeling the whole set of keys is an interesting avenue of future work that would reduce the number of linear regression models our method requires. Our experiments showed that the access time to the table where we keep the linear regression models, i.e., the model table, does not constitute a big portion of the overall search time (see Section 6.1.2). However, this result can change for bigger sizes of B+trees, and using a piece-wise linear function can mitigate this problem. We discuss this issue further in Section 7.

# 9. CONCLUSION

In this paper, we show that simple machine learning techniques such as linear regression are successful in predicting B+tree leaf nodes with 80% to 96% accuracy. We develop an algorithm that integrates learned models into B+tree search operation, and adaptively updates the models in the presence of inserts. The results show that the proposed algorithm can reduce B+tree search time by 33% to 57%, and is robust against inserts. Moreover, the proposed method can update the machine learning models fast enough to account for the newly inserted keys and nodes.

# 10. ACKNOWLEDGMENTS

We thank the AIDB reviewers and the members of the DIAS laboratory for their constructive feedback and support throughout this work. We thank Timo Kersten and Panos Sioulas for their useful comments. We thank Thomas Neumann for supporting our cross-university collaboration. This project has received funding from the DIAS Lab, IC Faculty, EPFL - Ecole Polytechnique Federale de Lausanne, Switzerland.

# 11. REFERENCES

[1] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. A-Tree: A Bounded Approximate Index Structure. *CoRR*, abs/1801.10207, 2018.

[2] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A Learned Database System. In *CIDR*, 2019.

[3] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, pages 489–504, 2018.

[4] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*, pages 38–49, 2013.

[5] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal*, 27(5):643–668, 2018.

[6] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, pages 302–313, 2013.

[7] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*, pages 183–196, 2012.

[8] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*, pages 387–402, 2016.

[9] U. Sirin, A. Yasin, and A. Ailamaki. A Methodology for OLTP Micro-architectural Analysis. In *DAMON*, pages 1:1–1:10, 2017.