CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm

Jiaoyi Zhang
Tsinghua University
Beijing, China
jy-zhang20@mails.tsinghua.edu.cn

Yihan Gao
Tsinghua University
Beijing, China
gaoyihan@mail.tsinghua.edu.cn

ABSTRACT

Learned indexes, which use machine learning models to replace traditional index structures, have shown promising results in recent studies. However, existing learned indexes exhibit a performance gap between synthetic and real-world datasets, making them far from practical indexes.

In this paper, we identify that ignoring the importance of data partitioning during model training is the main reason for this problem. Thus, we explicitly apply data partitioning to index construction and propose a new efficient and updatable cache-aware RMI framework, called CARMI. Specifically, we introduce entropy as a metric to quantify and characterize the effectiveness of data partitioning of tree nodes in learned indexes and propose a novel cost model, laying a new theoretical foundation for future research. Then, based on our novel cost model, CARMI can automatically determine tree structures and model types under various datasets and workloads by a hybrid construction algorithm without any manual tuning. Furthermore, since memory accesses limit the performance of RMIs, a new cache-aware design is also applied in CARMI, which makes full use of the characteristics of the CPU cache to effectively reduce the number of memory accesses. Our experimental study shows that CARMI performs better than baselines, achieving an average of 2.2×/1.9× speedup compared to B+ Tree/ALEX, while using only about 0.77× memory space of B+ Tree. On the SOSD platform, CARMI outperforms all baselines, with an average speedup of 1.2× over the nearest competitor RMI, which has been carefully tuned for each dataset in advance.

PVLDB Reference Format:
Jiaoyi Zhang and Yihan Gao. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. PVLDB, 15(11): 2679 - 2691, 2022.
doi:10.14778/3551793.3551823

1 INTRODUCTION

As an indispensable access method of database systems, indexes provide fast data accesses by avoiding expensive table scans. Traditional index structures are general-purpose, in the sense that they organize data according to fixed rules without taking advantage of the characteristics of underlying data distribution. Recently, Kraska et al. [24] pioneered a line of research where indexes are constructed using machine learning models. Specifically, they proposed a structure called the Recursive Model Index (RMI). In RMI, index nodes themselves are ML models, and they are connected hierarchically. To perform a lookup, we traverse the tree-like structure using the ML model in each node to determine the child branch to continue. Upon arriving at a position in the data array (i.e., leaf nodes), we perform the "last-mile" search within a range to correct the model prediction error. [24] demonstrates that the RMI and learned indexes generally exhibit smaller memory consumption and superior search performance compared to B+trees.

Despite the promising results shown in the latest research [13, 15, 16, 22, 24, 33, 41], few real database systems choose to adopt learned indexes. An outstanding reason, as indicated in the SOSD benchmark paper [21], is that the performance of learned indexes (represented by the RMIs) drops significantly when moving from synthetic datasets to real-world datasets. The average latency of an index lookup is 2.92× larger on real-world datasets than that on the same-sized synthetic ones, as shown in [21].

We argue that the main reason behind such a performance gap lies in the fact that existing RMI designs overlook the importance of data partitioning during model training. Many current RMIs, including the original proposal [24], typically emphasize the "model-fitting" aspect during index construction. Since the fanout and depth of the index are predetermined (and manually tuned), each leaf node is assigned a corresponding subset of the data and tries to train a model to fit the cumulative distribution function (CDF) of the assigned data as accurately as possible.

The consequence of such a rigid data partitioning strategy contributes to the large performance gap between synthetic and real-world datasets for RMIs. For synthetic datasets, the local data distribution at each leaf node is relatively "smooth" so that the expected prediction error is small, leading to a fast last-mile search. Real-world datasets, however, are much "bumpier", and the degree of "bumpiness" varies across data ranges. Such irregularity in data distribution makes it difficult for simple models (e.g., linear regression) to achieve accurate predictions under predetermined or manually-tuned data partitions. Larger prediction errors require more memory accesses to correct and, therefore, hurt the index performance. Prior work such as [13, 16, 45] has attempted to automate partitioning during index construction. Their approaches, however, are heuristic-based and only work on a fixed model type, i.e., linear model.

In this paper, we propose to address the issue by explicitly incorporating data partitioning into the RMI construction process. First, we propose a new cost model that considers the data partitioning aspect for RMI training. The effectiveness of data partitioning is quantified using the entropy [18] over the number of data points in

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551823

each partition. The intuition is that larger entropy indicates smaller partition sizes and a more even size distribution. Smaller partitions are preferable because they facilitate leaf-node training and produce more accurate models, especially on non-linear datasets. In addition, smaller child partitions flatten the hierarchical structure of RMI, and thus reduce cache misses. Meanwhile, more even child partitions lead to a more balanced tree structure.

Based on the new cost model, we formalize the index construction problem as an optimization problem and solve it using an algorithm combining the greedy and the dynamic programming approaches. Unlike CDFShop [33], where the node types for each layer are determined ahead of time, our algorithm selects the best model and the best partition fanout for each node automatically at construction time without the need for recompilation.

Finally, since memory accesses dominate the lookup performance for RMIs [31], we design the memory layout for each node in a cache-aware manner. Specifically, we require the size for every node with a model to be exactly the cache line size (i.e., 64 bytes) so that each node visit (or model inference) incurs at most one cache miss. For leaf nodes containing data points across multiple cache lines, we adopt a two-level B+tree design, consisting of a 64-byte root and multiple 256-byte data blocks. Such a cache-aware layout can effectively reduce the number of memory accesses during the last-mile search as in existing solutions [13, 22, 24, 33], especially on real-world datasets, with only a small space overhead. In addition, fixed-sized nodes facilitate memory prefetching where memory accesses are parallelized to further reduce the access latency.

We present CARMI, a novel **C**ache-**A**ware **RMI** framework that implements the new ideas introduced above with six example tree node (model) types. Our experimental study shows that CARMI outperforms all baselines on both our microbenchmark and the SOSD benchmark. In our microbenchmark, CARMI achieves an average speedup of 2.2× (up to 4.2×) and 1.9× (up to 7.2×) compared to B+tree and ALEX, respectively, while only using about 0.77× memory space of B+tree. On the SOSD benchmark, CARMI achieves an average speedup of 2.5× (up to 3.0×)/1.5× (up to 2.3×) compared to B+tree/ALEX, respectively. Compared to its closest competitor RMI, which has been carefully tuned for each dataset in advance, CARMI is still 1.2× faster on average (up to 1.5×).

We make the following contributions:

- We identify that the inflexibility of data partitioning for learned indexes is one of the key reasons why there is a significant performance gap when applying them to synthetic and real-world datasets.
- We propose a new cost model for RMI training, which uses entropy across partition sizes to measure the effect of data partitioning on index performance.
- We formalize the index construction problem as an optimization problem and propose an algorithm to solve it efficiently and automatically.
- We propose CARMI, a novel RMI framework incorporating the new cost model and automatic node selection algorithm. CARMI also uses a new memory layout that is more cache-friendly, especially for the last-mile search.
- We conduct a series of experiments to demonstrate the superior performance and robustness of our framework.

The remainder of this paper is outlined as follows: In Section 2, we review the RMI framework and discuss the two ways of viewing RMI: model fitting vs. data partitioning. In Section 3, we derive a cost model for the entire index structure, and introduce entropy as a metric for characterizing the node performance. Section 4 discusses the cost-based hybrid index construction algorithm, which is used to choose different node settings flexibly to construct the optimal index structure during runtime. In Section 5, we explain the cache-aware designs of CARMI, including the new memory layout and a prefetching mechanism. The experimental setup and results are shown in Section 6. We discuss the possible extension directions and future works in Section 7. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 MOTIVATION AND CARMI

2.1 RMI and Data Partitioning

Figure 1 shows the structure of the Recursive Model Index (RMI), an ML-based index framework. Each inner node in the RMI represents an order-preserving regression model: the model $f : k \rightarrow idx$ takes a key k as input, and outputs an integer $idx \in \{1, 2, \ldots, c\}$, where c is the number of child nodes of this inner node. The order-preserving property guarantees $k_1 \leq k_2 \Rightarrow f(k_1) \leq f(k_2)$ so that an RMI can answer range queries correctly. At the bottom layer, leaf nodes are trained using linear models to fit the underlying data points. Searching the RMI given k proceeds as follows: starting from the root, we evaluate the model to determine which child node to visit for the next step. This process is repeated until a leaf node is reached. Finally, we perform a binary search (bounded by the maximum error) to retrieve the matching records.

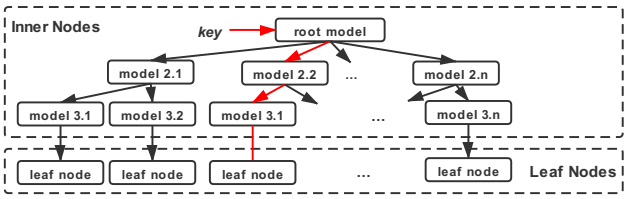

Figure 1: Learned Index

Existing RMI designs [13, 15, 16, 22, 24, 33, 41] view the index construction problem from the perspective of model fitting. In this view, models are trained to minimize a loss function (e.g., squared loss function) to best fit the CDF of a given dataset. Specifically, the root model aims to fit the CDF of the entire dataset, while the leaf-node models try to fit local distributions. Under such a problem setting, however, the index fanout and depth are typically predetermined before model training, thus wasting opportunities to improve model accuracies through more flexible data partitioning.

Approaching RMI construction via a "model-fitting" view can lead to a significant performance drop when shifting from synthetic datasets to real-world datasets. Because real-world datasets are often non-smooth and non-linear, as shown in Section 6.3, simple models such as linear regression cannot fit well in certain CDF ranges if the partitioning is too coarse-grained. Inaccurate predictions from the models then require additional procedures such

as large range scans to finish the "last-mile" search. These large-range searches span many cache lines and require multiple memory accesses, resulting in performance degradation.

In this paper, we propose to look at RMI construction from a "data-partitioning" view, where the models aim to partition a given dataset more evenly into smaller chunks to form a flatter and more balanced tree. Moreover, larger fanouts (i.e., more child partitions) of the upper-level models are preferable in terms of the overall prediction accuracy because lower-level nodes now can work on smaller local datasets to reduce their maximum model prediction errors. However, more partitions could lead to space overhead and performance degradation because of potentially more complex models. Therefore, we include the effectiveness of data partitioning in our new cost model along with search time and space, and develop an algorithm to train an RMI automatically using an objective function derived from the cost model.

To quantify the effectiveness of data partitioning, we propose to use entropy, an information-theoretic metric [18]. Suppose a node M distributes a total of n data points into c different child nodes[1], then the entropy $H(M)$ is defined as: $H(M) = -\sum_{i=1}^{c} p_i \log_2 p_i$, where $p_i = \frac{n_i}{n}$ and n_i is the number of data points allocated to the i-th child node. As mentioned in the introduction, larger entropies mean that datasets are divided more evenly into smaller subsets, which is more desirable as discussed above. Further, the entropy also helps establish the notion of local node efficiency, as described in Section 3.3.1, which combines the time and space cost of a single node and its dataset partitioning utility into a single metric. With this metric, we can effectively compare models locally without global information, thus speeding up index construction.

2.2 Overview of CARMI

In this paper, we extend the RMI framework and propose a refined general RMI framework that can automatically build suitable and updatable indexes for given datasets, called Cache-Aware RMI (CARMI). CARMI retains RMI's core idea of replacing traditional indexes with ML models and has similar procedures for querying data points. There are two types of nodes in CARMI: inner nodes and leaf nodes. Inner nodes use the trained models to navigate in the tree structure, while leaf nodes store 8-byte pointers to the data records (similar to leaf nodes in a B+tree). Differently from the original RMI, we use a new memory layout (§ 5) where all tree nodes are limited to 64 bytes and are stored in a single array. In addition, CARMI allows nodes to use different types of models for different sub-ranges in the dataset to maximize performance and compression.

For index construction, CARMI uses a hybrid algorithm (§ 4) whose optimization objective is to minimize the weighted sum of time and space costs (§ 3). With this algorithm, CARMI can automatically construct indexes with good performance at runtime.

We show an example of CARMI below:

Example 1. *Consider a dataset consisting of 500 data points* $D = \{0, 1, \cdots 199, 200, 202, \cdots, 798\}$. *As shown in Figure 2, an example CARMI has four layers. The root node adopts a linear regression model* ($idx = \lfloor 0.005 \times key \rfloor$) *to index its children. The LR inner node* M_0 *manages the first 200 data points, which are further indexed by two leaf nodes* M_4 *and* M_5. *The remaining 300 data points are directly indexed by leaf nodes* ($M_1 - M_3$), *each of which is associated with 7 data blocks.*

Suppose we have a point query with key = 592. *We first access the root node and use its model to calculate the next index* ($idx = \lfloor 0.005 \times 592 \rfloor = 2$). *After fetching the content of* M_2, *we use its strategy to get the index of the data block* (L_{27}), *where we perform a binary search for key* 592. *Because each node in CARMI occupies exactly one cache line, assuming that the root is already in the CPU cache, the above query only requires two random memory accesses: one for leaf node* M_2 *and the other for data block* L_{27}.

[1]Note that the number of child nodes for an inner node is part of the model configurations, and we need to determine its optimal value when constructing the index.

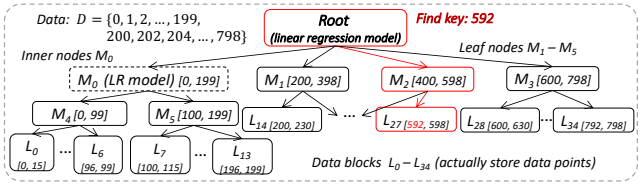

Figure 2: A Simple Example of CARMI

3 COST MODEL OF CARMI

In this section, we describe our new cost model for index construction in CARMI. To quantify the performance of tree nodes in a standalone manner, we first characterize each node in terms of time cost, space cost, and entropy, and then derive a cost model for the entire index structure. Since the analysis is generic, CARMI can support any new node types as long as they can be represented accordingly. Then, we use the cost model as an optimization objective to find the suitable node design and tree structure at runtime. We also briefly describe some example node designs available in our open-sourced CARMI [1] and used in our experiments in Section 6.

The rest of the section is organized as follows. We analyze inner/leaf nodes from three aspects in Sections 3.1 and 3.2, respectively. The entire cost model and the formulation of the index construction problem are described in Section 3.3. Finally, we briefly discuss a few specific tree node designs in Section 3.4.

3.1 Inner Nodes

The main functionality of inner nodes is determining which branch to go through, so that we can quickly map a given key to its corresponding leaf node. In the following, we discuss three separate dimensions for characterizing inner nodes: the time required for determining the next branch, the space cost of the node, and the data partitioning effectiveness of the node.

3.1.1 Time. The time cost of an inner node includes two parts: the access time and the computation time, denoted as $TCost_{access}$ and $TCost_{CPU}$, respectively. $TCost_{access}$ refers to the time to read the node content, which is equal to the latency of the main memory due to our cache-aware design in Section 5. $TCost_{CPU}$ is the time required for a model to compute the index of the next node, which only depends on the model type. Then, the total time required for an inner node M to predict the next branch is: $TCost(M) = TCost_{CPU} + TCost_{access}$.

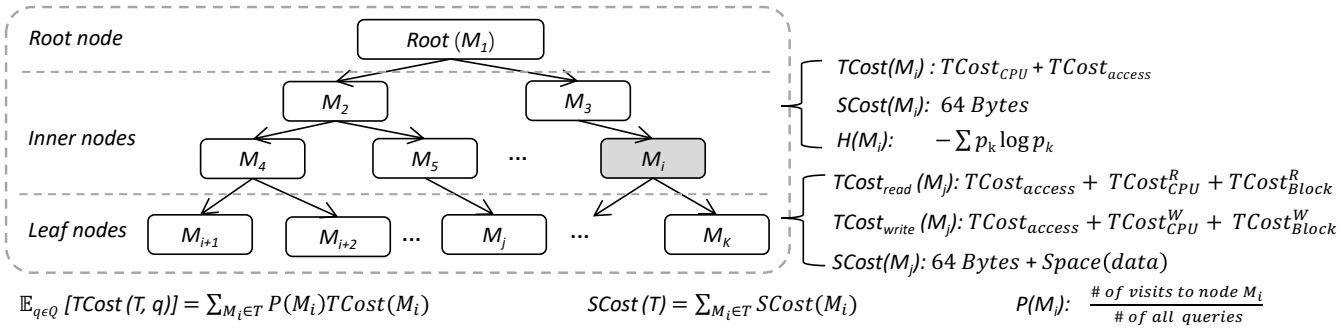

Figure 3: Cost Model of CARMI

3.1.2 Space. The space cost of an inner node M is the total amount of space in bytes, which is denoted as $SCost(M)$.

3.1.3 Entropy. We use the entropy metric to characterize the ability of an inner node to partition a dataset evenly. The entropy of an inner node M is: $H(M) = -\sum_{i=1}^{c} p_i \log_2 p_i$ (details in Section 2.1).

3.1.4 Root Node. The root node in CARMI is handled differently compared to inner nodes. Since the root node is always accessed during lookup, we can assume that it is in the cache memory. As a result, its $TCost_{access}$ is equal to the latency of the cache memory.

3.2 Leaf Nodes

Leaf nodes are used to manage the actual data points and can also be characterized in terms of time and space cost. The third dimension is the capacity for storing data points (i.e., how many data points can be stored in the leaf node).

In CARMI, we design a new type of leaf node similar to a two-level B+tree node. Its root is 64 bytes and contains pointers to multiple 256-byte data blocks that store data points.

3.2.1 Time. For leaf nodes, the time cost of two specific operations is analyzed: insert a new data point and lookup a data point[2].

Similar to inner nodes, the time to access the node ($TCost_{access}$) is equal to the main memory access latency. Due to the new leaf node, finding a data point requires first finding the index of the data block, and then locating the data point within the block. Their time costs are denoted as $TCost^R_{CPU}$ and $TCost^R_{Block}$, respectively.

As for the insert operation, the content of both leaf node and data blocks might need to be changed accordingly. To reflect this, we use a different superscript ($TCost^W_{CPU}$ and $TCost^W_{Block}$).

Overall, the time cost of a leaf node is modeled as:

$$\begin{aligned} TCost_{read}(M) &= TCost_{access} + TCost^R_{CPU} + TCost^R_{Block} \\ TCost_{write}(M) &= TCost_{access} + TCost^W_{CPU} + TCost^W_{Block} \end{aligned} \tag{1}$$

3.2.2 Space. The space cost of a leaf node M, consists of the bytes of metadata and data blocks. Then the total space cost is:

$$SCost(M) = SCost_{leaf} + Space(data) \tag{2}$$

where $SCost_{leaf}$ is 64 bytes, and $Space(data)$ is the total amount of space occupied by the data blocks.

[2]Deletion and update operations are not discussed since they are similar to the read access operations (we adopt a lazy deletion approach).

3.2.3 Capacity of Leaf Nodes. The capacity of a leaf node refers to the number of data points stored in it, and it depends on the total amount of space allocated for data points and the way they are arranged. For example, if we need to make room for future inserts to reduce the latency of insert operations, then the capacity of the leaf node will be reduced accordingly.

3.3 The Optimization Problem

Based on the above analysis, we can define a cost model for the entire index structure. The time cost of queries can be estimated by utilizing the above analysis results: For any query q and index structure T, let the traversal path of q in T be $M_1(root) \rightarrow M_2 \rightarrow \ldots \rightarrow M_k(leaf)$, then the time cost of q can be approximated as:

$$TCost(T, q) = \sum_{i=1}^{k} TCost(M_i) \tag{3}$$

The space cost of the index structure is simply the sum of the space cost of all inner nodes and leaf nodes:

$$SCost(T) = \sum_{M_i \in T} SCost(M_i) \tag{4}$$

Now we can formalize the index construction problem as an optimization problem: We would like to find the optimal index structure that minimizes the average time cost of each query under a given space cost budget. Here the average time cost is evaluated with respect to a fixed query workload known in advance, which can be obtained from users' recent history queries. In most cases, recent history queries will faithfully reflect the characteristics of future queries. If no history queries are available, we can use a uniform access workload where each data point is accessed once.

The problem of finding an optimal tree structure is formulated as follows:

Problem 1. *Let $Q = \{q_1, \ldots, q_m\}$ be a collection of queries, and $D = \{d_1, \ldots, d_n\}$ be the collection of keys to be maintained in the index structure. Find the optimal index tree structure T such that $\mathbb{E}_{q\in Q}[TCost(T, q)]$ is minimized, under the constraint that the total space cost of T does not exceed a fixed budget B: $SCost(T) \leq B$.*

Problem 1 is a constrained optimization problem, and one common strategy for solving it is to convert it to an unconstrained

problem using the Lagrange multiplier method [9]. Then, Problem 1 can be transformed into a roughly equivalent form with a linear combination of time and space cost as the objective:

Problem 2. *Let $Q = \{q_1, \ldots, q_m\}$ be a collection of queries, $D = \{d_1, \ldots, d_n\}$ be the collection of keys maintained in the index, and λ be a positive constant parameter. Find the optimal index tree structure T such that $\mathbb{E}_{q \in Q}[TCost(T, q)] + \lambda SCost(T)$ is minimized.*

Problem 2 suggests that we ultimately want to minimize a weighted sum of the time and space cost of the index, and we will describe an algorithm for solving it in Section 4.

Let $P(M_i)$ be the fraction of history queries passing through a tree node M_i. Then the expression $\mathbb{E}_{q \in Q}[TCost(T, q)]$ can be rearranged into an alternative form:

$$\mathbb{E}_{q \in Q}[TCost(T, q)] = \sum_{M_i \in T} P(M_i) TCost(M_i) \tag{5}$$

Finally, a summarization of the cost model of CARMI can be found in Figure 3 for fast reference.

3.3.1 Theoretical Analysis. In the following analysis, we assume the history queries to be a uniform access of data points for simplicity. Then, the value of $P(M_i)$ is the same as the total fraction of data points in the subtree of M_i. With this assumption, the following theorem establishes a connection between $\sum_i P(M_i)H(M_i)$ and the total number of data points n.

Theorem 3.1. *Let $T = \{M_1, \ldots, M_K\}$ be an index structure with K nodes in total. $P(M_i)$ represents the ratio of data points in node M_i relative to the total number n. Then we have: $\sum_{i=1}^{K} P(M_i)H(M_i) = \log_2 n$, where for leaf nodes, $H(M_i)$ is defined as $\log_2(Capacity(M_i))$ and $Capacity(M_i)$ is the capacity of leaf nodes (§ 3.2.3).*

The proof of this theorem can be found in our technical report [47]. Essentially, Theorem 3.1 states that the weighted sum of the entropy of all tree nodes is always a constant value. In other words, entropy characterizes the "contribution" of each tree node to the index structure: the higher entropy each node contributes, the less the overall number of tree nodes we need in the index structure.

It is also of interest to compare Theorem 3.1 with the optimization objective of Problem 2. We can rewrite the objective using Equation 5:

$$\text{Objective} = \sum_{i=1}^{K} [P(M_i)TCost(M_i) + \lambda SCost(M_i)]$$

Comparing with Theorem 3.1, we see that intuitively each tree node M contributes $P(M)H(M)$ entropy-wise to the index structure while incurring $P(M)TCost(M) + \lambda SCost(M)$ cost to the overall objective. Thus the ratio of these two terms can be used to quantify the local efficiency of node M:

$$\text{cost-ratio(M)} = \frac{P(M)TCost(M) + \lambda(SCost(M))}{P(M)H(M)} \tag{6}$$

Generally, we want to minimize the cost-ratio of all tree nodes, especially the ones with a large value of $P(M)H(M)$. For example, if all tree nodes have a cost-ratio less than c, then the optimization objective would be bounded by $c \log_2 n$.

Table 1: The Mechanism of Various Types of Nodes

Node	Mechanism
LR	Use a LR model to determine the corresponding branch
P. LR	The model is a piecewise linear regression model
Hist	Determine the branch through a histogram lookup table
BS	Use binary search to determine the branch, similar to B+tree nodes
CF Array	A two-layer cache-friendly structure similar to a B+tree
Ext. Array	Only store meta data, and data points are stored in an external location

3.4 Specific Implementation

We have implemented four types of inner nodes and two types of leaf nodes in CARMI. For inner nodes, they use either linear regression, piecewise linear regression, binary search or histogram models to predict the next branch. We have implemented two different types of leaf nodes: cache-friendly array (CF array) and external array. CF array leaf nodes store data points compactly in data blocks in a sequential manner, and the leaf node itself stores the minimum key values of data blocks. External array leaf nodes are used for primary index structures, where the original data points are already sequentially stored in an external location. In such a case, we only need to store pointers to external locations in the leaf node.

Note that the choice of inner/leaf nodes can be flexibly determined at runtime and do not need to agree on a single one throughout the tree structure. The mechanism of each type is outlined in Table 1. The specific implementation details can be found in our technical report [47], along with the empirical performance, entropy, and local efficiency of these nodes.

4 INDEX CONSTRUCTION ALGORITHM

In Section 3.3, we have shown that the optimal index tree structure can be constructed by minimizing the weighted sum of the time and space cost of the index structure (see Problem 2). In this section, we describe an algorithm for solving it.

First, let us rearrange the optimization objective as follows:

$$\begin{aligned}\mathbb{E}_{q \in Q}[TCost(T, q)] + \lambda SCost(T) &= TCost(root) + \lambda SCost(root) \\ &+ \sum_{i=1}^{c} [\frac{|Q_{T_i}|}{|Q|} \mathbb{E}_{q \in Q_{T_i}}[TCost(T_i, q)] + \lambda SCost(T_i)]\end{aligned} \tag{7}$$

where c represents the number of child nodes of the root node, T_i is the sub-index tree of the i-th child node, and Q_{T_i} is the collection of queries that access T_i. Note that the number of child nodes is chosen from an exponentially increasing sequence to reduce training time, such as the powers of 2.

As we can see, the terms inside the square brackets have a very similar form compared to the original objective. In other words, once the root node of this subtree is fixed, the original optimization problem can be broken down into several independent subproblems with similar forms, and each sub-problem can be solved independently. We have two algorithms to solve each sub-problem:

- **Node selection algorithm:** A greedy algorithm that only considers local information to construct nodes, which is used when the subtree manages a large dataset (e.g., the root node).
- **Dynamic programming (DP) algorithm:** This algorithm is guaranteed to find the optimal sub-structure but is slower, and we only use it when the sub-dataset is small.

Our hybrid construction algorithm uses these two algorithms in combination, and the overall workflow for constructing the index structure for a given dataset is:

- First, we choose a root node setting using the greedy node selection algorithm.
- Next, we use the setting obtained in the previous step to assign the dataset to the child nodes of the root, and then construct a sub-index tree over each assigned sub-dataset.
- Each child node chooses the appropriate algorithm to build the sub-index structure according to the size of the sub-dataset.
- Finally, all the sub-structures are merged together and linked to the root node to form the complete index tree.

The rest of this section is organized as follows: The greedy node selection step is described in Section 4.1, and the DP algorithm is described in Section 4.2.

4.1 Node Selection Algorithm

For nodes that manage a large dataset, such as the root node, we hope to quickly select a good node design using only local information. Below, we develop a greedy node selection algorithm to find the locally optimal solution without considering the design of the lower-level nodes:

- For the current node, this algorithm enumerates various possible node types and considers several choices for the number of child nodes. For each setting, we calculate the time cost, space cost and entropy of the node.
- Then, we calculate the cost ratio of each setting using Equation 6 in Section 3.3, and select the one with the minimum cost ratio as the current node.
- Using the setting obtained in the previous step, we assign each data point to one of the child nodes of the current node.
- Finally, the sub-index tree of each child node is constructed recursively using either this algorithm or the DP algorithm.

4.2 Dynamic Programming Algorithm

As shown in Equation 7, the original optimization problem can be decomposed into similar sub-tasks once the root node is fixed. Based on this intuition, we can use a dynamic programming algorithm to solve this problem:

- **State:** The state of each step is the dataset handled by the current node, denoted as $D_{l,r}$, where l and r are the left and right indexes of the entire dataset. Let $Q_{l,r}$ be the collection of queries that access $D_{l,r}$, and $T_{l,r}$ be the sub-index tree for $D_{l,r}$. Then the corresponding entry $cost[l,r]$ in the DP table is the value of the following expression:

$$cost[l,r] = \min_{T_{l,r}}\{\frac{|Q_{l,r}|}{|Q|}\mathbb{E}_{q\in Q_{l,r}}[TCost(T_{l,r},q)] + \lambda SCost(T_{l,r})\} \quad (8)$$

- **State transition equation:** Let S be the collection of all possible settings for the root node of the sub-index tree, including the type of the root node and the number of the child nodes. Then according to Equations 7 and 8, the state transition equation is:

$$cost[l,r] = \min_{(M,c)\in S}\{\frac{|Q_{l,r}|}{|Q|}TCost(M)+\lambda SCost(M)+\sum_{j=1}^{c} cost[l_j,r_j]\}$$

where M and c are the node type and the number of child nodes (if the node is a leaf node, c is 0), respectively, l_j and r_j are the left and right indexes of the sub-dataset that belongs to the child node j. If the dataset is smaller than a certain threshold, then we only consider the leaf node settings in the state transition equation.

Specifically, given a sub-dataset $D_{l,r}$, we need to enumerate all its possible settings for the root node, and compute the cost of each setting separately. Then, the setting with minimum cost is selected to construct the current node, and the cost is stored in the entry $cost[l,r]$. For each setting, we first look up the time/space cost of the root node from our cost model, and use the sub-dataset $D_{l,r}$ to train the root node model M. Then, we use the trained model M to assign the dataset to c child nodes and obtain their cost via a memorized search approach: for each subtask of computing $cost[l_j,r_j]$, the algorithm first checks whether it has been solved before. Then according to the check result, it either returns the minimum cost directly from the DP table or recursively calls the process of the DP algorithm. When the number of data points in the subtree is less than a pre-specified threshold, the algorithm will directly construct a leaf node as the current node. Otherwise, the algorithm considers two cases (leaf nodes or subtrees with inner nodes) and chooses the optimal design according to the transition equation. The pseudocode can be found in our technical report [47], as well as a simple example to illustrate our construction algorithm.

Although, in principle, the dynamic programming algorithm can be used to construct the optimal index structure for the entire dataset, in practice, it is too slow to handle large datasets. Therefore, we only use it to solve sub-problem that are small enough.

5 CACHE-AWARE DESIGN

In this section, we describe the cache-aware design of CARMI with a new memory layout in detail, use some examples to explain the intuition and benefits of such a design, and also discuss the potential opportunities for using memory prefetching instructions to further speed up queries.

The rest of this section is organized as follows: Section 5.1 discusses the details of cache-aware design, and Section 5.2 describes our new memory layout and the basic lookup and insert operations. The use of memory prefetch will be discussed in Section 5.3.

5.1 Details of Cache-Aware Design

B+tree with tree nodes occupying exactly a cache line size can outperform standard binary search trees [40]. Specifically, each memory access always copies a fixed-size memory content (i.e., cache line size, usually 64 bytes), then indexes that utilize the cache and do not "waste" any data retrieved into cache memory can generally outperform standard data structure by a large margin.

In CARMI, we employ the same design decision as in B+tree and enforce all tree nodes to have a fixed size of 64 bytes. To understand the intuition behind such a design, we briefly analyze the performance bottleneck of the lookup procedure in RMI. For an index structure within the RMI framework (shown in Figure 1), the process for accessing a data point is as follows:

- We first access the content of the root node, and then use its model to choose a child node of the root.
- Subsequently, in each layer, we visit a node chosen by the node in the previous layer, and then use the model of this node to choose one of its child nodes. We repeat this step until we have reached a leaf node.

In the above procedure, for each tree node, we need at least one memory access to get its content before performing the model computation. For simple models (e.g., linear models), the computation time is usually less than 20 ns, while a memory access takes about 70-100 ns if the node content is not cached. Therefore, the time required for memory access would actually take up most of the time we spent on data lookup. In other words, the performance bottleneck of RMI is memory access.

Based on the above analysis, we hope to minimize the number of memory accesses and fully utilize each access during data lookup. In CARMI, we achieve them from the following two aspects.

First, we enforce each node to have a size of exactly 64 bytes to align the cache line. This design allows the node to be fetched with only one memory access. Furthermore, the large node size allows it to store rich information, which helps to reduce the average tree depth, thus reducing the number of needed memory accesses.

To further reduce the number of memory accesses in the last mile, we design a new leaf node called Cache-Friendly array leaf node, which is conceptually similar to a two-layer B+tree node. The first layer is a 64-byte root that stores pointers to several 256-byte data blocks in the second layer and the minimum key values of each block. With such a design, we only need one memory access to obtain metadata to determine the next data block and narrow down the last-mile search range to 256 bytes, effectively reducing memory accesses on real-world datasets.

5.2 Memory Layout

For the memory layout of CARMI, we have two main arrays, *data* and *node*, to assist in implementing our cache-aware design. These two arrays are used to store data points and tree nodes, respectively, as shown in Figure 4, with details as follows:

- **Data array:** The *data* array is used to store data points in CARMI. It is a large array containing many small data blocks, each of which has a fixed size, represented by a parameter *kBlockSize* with a default value of 256. Data points are stored in these data blocks. In the *data* array, data blocks managed by the same leaf node are stored in adjacent locations.
- **Node array:** All tree nodes, including inner nodes and leaf nodes, are stored in the *node* array. Each tree node occupies a total of 64 bytes: the first byte is always the node type identifier, and the next three bytes are used to store the number of child nodes (the number of data blocks for leaf nodes). For inner nodes, the following 4 bytes represent the starting index of the child nodes in the *node* array. For leaf nodes, they store the starting index of data blocks in the *data* array instead. The remaining 56 bytes store additional information depending on the tree node type.

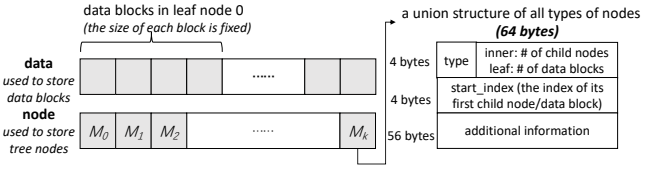

Figure 4: Memory Layout

With the help of the above two arrays, we can perform operations such as lookup and insert. The way we use them in lookup and insert operations is described below.

5.2.1 Lookup Operation. When accessing a data point, we first use the root node model to compute the index of the node in the next layer. Next, we access the tree node according to the index value and use its model to update the value of the index variable. This process is repeated iteratively until a leaf node is visited. Finally, we search within this leaf node to get the corresponding data record.

5.2.2 Insert Operation. The basic process of the insert operation is similar to the lookup operation. After finding the correct data block for insertion, we insert the data point into it. In addition, there are two mechanisms that can be initiated by the leaf node under certain situations:

- **Expand:** When a leaf node needs more space, it can initiate an expand operation to get more data blocks. We first collect all the data points stored in it, then construct a new leaf node at a new location with more space, as shown in Figure 5. The new leaf node then replaces the original one to complete the process.
- **Split:** If we can no longer use a single leaf node to efficiently manage all the data points, it needs to be split. The leaf node will be replaced with a subtree consisting of a new inner node and several new leaf nodes, as shown in Figure 5. The number of leaf nodes in the subtree depends on the trained model of the new inner node.

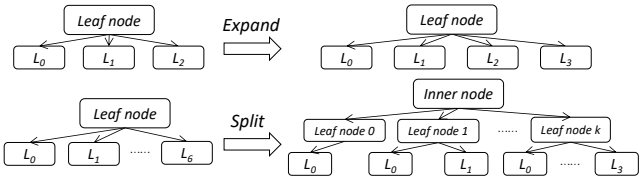

Figure 5: Expand and Split Mechanism

5.3 Prefetch

5.3.1 Prefetch Mechanism. In some cases, when the key value distribution in the dataset is very regular (e.g., uniform distribution), the index of the data block can be directly predicted from the input key value. In such cases, we can further reduce the data access latency by utilizing memory prefetching. Specifically, we add an additional prefetch model to the root node in order to predict the

data block in which a given key value may be stored. Therefore, during the access process, the predicted data block is prefetched at the root node. This prefetching operation can be executed in parallel with other memory accesses in the normal process. If the prediction is correct, the data block will be available in the cache when we need to access it. In this way, we can speed up the access for datasets with regular data distributions. We demonstrate the prefetch mechanism via the following example:

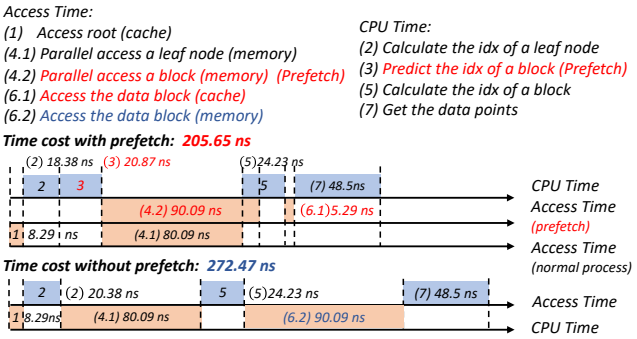

Figure 6: Memory Prefetch in CARMI

Example 2. *We use the same settings as in Example 1 to show the effect of prefetching. The time cost of accessing a data point in CARMI can be divided into CPU time and access time. In this example, we need to access: the root node (8.29 ns), the leaf node (80.09 ns), and the data block (90.09ns). The CPU time consists of model calculations in the root node (20.38 ns), the leaf node (24.23 ns), and the search process (48.5 ns) in the data block. Note that the CPU calculations cannot be processed until the corresponding cache/memory access is complete. Without prefetching, CPU calculations and access operations are executed sequentially and do not overlap, resulting in a total time cost of 272.47 ns.*

To utilize the prefetch mechanism, an additional 28.87 ns of CPU time are required to predict the index of the data block in order to prefetch it in advance at the root node. The advantage of such prefetching step is that when we need the data block later in the process, it will already be in the cache and can be fetched rapidly, hence reducing the time required for a memory access. In this case, a lookup operation with prefetching requires only 205.65 ns, a reduction of 67 ns compared to the situation without prefetching.

5.3.2 Prefetch Support. To support the prefetching design, we need to make a few changes to the construction algorithm: we need to add an additional model at the root node to predict the index of the data block, and we also want to store data points in data blocks according to the prefetch prediction model whenever possible. Details of these changes can be found in our technical report [47], together with the pseudocode of the construction algorithm.

6 EXPERIMENTS

In this section, we conduct experiments on various datasets and workloads to evaluate the performance of CARMI and delve into CARMI from different aspects through several auxiliary experiments. Due to page limitations, some contents are described in our technical report [47].

6.1 Experimental Setup

We conducted all the single-threaded experiments on an Ubuntu Linux machine equipped with an AMD Ryzen 3700X 8-Core Processor and 32GB RAM.

6.1.1 Datasets. We use 7 datasets in our experiments. Synthetic datasets are stored as key-value pairs of <*double*, *double*>, while real-world datasets are <*uint*64, *uint*64>. Each dataset is 1GB (i.e., the number of records is 67,108,864), and the details are as follows:

- **Four synthetic datasets:** These datasets are generated from 4 distributions: (a) **lognormal**: $log(key) \sim \mathcal{N}(0, 1)$; (b) **uniform**: $key \sim \mathcal{U}(0, 1)$; (c) **normal**: $key \sim \mathcal{N}(0, 1)$; (d) **exponential**: $key \sim \text{Exp}(0.25)$. Key values are multiplied by 10^8.
- **YCSB dataset**: YCSB represents the IDs from the YCSB benchmark [12], which are uniformly distributed.
- **OSMC dataset:** This dataset is generated from a public dataset of Open Street Maps [2], which includes the latitude and longitude of points worldwide.
- **Facebook dataset:** This dataset is generated from the IDs of Facebook users [21, 42].

We have also ported CARMI into the SOSD [31] platform to test its performance, and the details are shown in Section 6.3.

6.1.2 Evaluation Workloads. In our experiments, we use 4 different query workloads. The first three workloads are similar to the workloads in the YCSB benchmark [12]: (a) a write-heavy workload with a mix of 50% reads and 50% inserts; (b) a read-heavy workload with a mix of 95% reads and 5% inserts; (c) a read-only workload. Finally, we include a range scan workload with a mix of 95% range scan and 5% inserts as in [13].

For each workload, we execute 100,000 operations and measure the average time used by each operation. The lookup keys are selected randomly from the existing keys in the index. We consider two access patterns for generating queries: Zipfian distribution (the normalized frequency of the x-th element is: $f(x) = \frac{1}{x^\alpha} / \sum_{i=1}^{N} \frac{1}{i^\alpha}$, $\alpha = 0.99$ as in [13]) and uniform distribution. In workloads involving insert operations, read and insert operations alternate proportionally. For example, in read-heavy workloads, we execute 19 lookups followed by 1 insert operation. For range scan workloads, the length of each range scan is uniformly sampled from [1, 100].

The queries for the YCSB dataset are handled differently: read queries are generated from the Zipfian distribution, but insert keys are monotonically increasing to align with the YCSB benchmark.

6.1.3 Implementation and Baseline. We compare against the following baselines: STX B+tree [3] and ALEX [13]. The node size of B+tree is 512 bytes, which is optimal for in-memory queries [46]. All other parameters use the default values in the source code. SOSD benchmark [21] also includes other indexes as baselines in Section 6.3. The source code for CARMI is available at [1]. In our experiments, we only tune one parameter λ, which is sensitive to data distribution and dataset size. All other parameters do not need to be tuned and use the default values unless otherwise stated. Among them, the size of each data block is 256 bytes, and the maximum capacity of external leaf nodes is 512. The default value of *kDPThreshold* is 512, which is used to switch between the DP and greedy algorithms. If the size of a sub-dataset is less than 90, we

workloads	indexes	uniform dataset			lognormal dataset			YCSB dataset		OSMC dataset			Face dataset		
		space /MB	time /ns		space /MB	time /ns		space /MB	time /ns	space /MB	time /ns		space /MB	time /ns	
			uniform	zipfian		uniform	zipfian				uniform	zipfian		uniform	zipfian
read-only	**CARMI**	**1394.3**	**114.3**	**85.1**	**1475.7**	**129.2**	**92.1**	**11.4***	**93.2**	**1771.5**	**287.5**	**160.1**	**1812.6**	**248.5**	**166.4**
	ALEX	1474.3	112.5	83.3	1476.7	115.5	89.3	1474.3	97.6	1522.0	421.9	221.4	1522.0	404.0	228.1
	B-Tree	2276.0	482.6	267.8	2276.0	480.6	268.1	2276.0	269.7	2276.0	479.9	262.5	2276.0	480.6	261.5
write-heavy	**CARMI**	**1509.9**	**168.4**	**154.5**	**2127.4**	**190.3**	**168.6**	**11.4***	**161.2**	**1772.5**	**357.9**	**246.0**	**1882.7**	**316.9**	**246.2**
	ALEX	1474.3	164.1	147.0	1476.7	294.0	262.9	1474.3	317.3	1522.0	1109.1	893.3	1522.0	1801.7	1777.6
	B-Tree	2276.0	500.3	419.9	2276.0	522.4	421.9	2276.0	401.2	2276.0	503.5	421.3	2276.0	498.7	432.9
read-heavy	**CARMI**	**1509.4**	**135.8**	**102.1**	**2077.5**	**155.0**	**104.2**	**11.4***	**104.4**	**1771.9**	**330.2**	**172.5**	**1812.9**	**278.3**	**174.2**
	ALEX	1474.3	116.2	89.2	1476.7	121.7	95.4	1474.3	234.7	1522.0	651.7	420.4	1522.0	993.8	829.0
	B-Tree	2276.0	488.9	286.8	2276.0	483.4	292.5	2276.0	280.4	2276.0	483.1	284.5	2276.0	477.6	280.1
range scan	**CARMI**	**1509.4**	**402.4**	**256.4**	**2077.5**	**434.3**	**271.3**	**11.4***	**266.4**	**1771.9**	**585.6**	**305.3**	**1812.9**	**600.9**	**323.9**
	ALEX	1474.3	375.0	233.6	1476.7	383.6	240.3	1474.3	359.3	1522.0	935.3	598.2	1522.0	1246.4	990.5
	B-Tree	2276.0	704.5	416.5	2276.0	728.5	414.7	2276.0	393.1	2276.0	696.4	417.1	2276.0	698.7	415.3

* Does not include the space of external array.

Figure 7: CARMI vs. Baselines: Time and Space Usage Comparison.

will directly construct a leaf node instead of choosing a better one from leaf/inner nodes. When a leaf node needs to split, we replace it with an inner node and 16 leaf nodes.

6.1.4 Training Queries for Index Construction in CARMI. In our experiments, the training query workload for index construction is specified as a uniform read access over all data points, and a uniformly sampled subset of data points to serve as key values for insert queries (except the YCSB dataset). The ratio of read/insert queries is the same as in the target query workload.

6.2 General Efficiency Comparison

In this section, we evaluate the performance of CARMI[3] and compare it with the baselines. We consider 4 query workloads and 7 different datasets, as explained in Section 6.1. We consider two access patterns for datasets other than YCSB. These result in a total of 52 possible configurations. We show 36 of the results in Figure 7. Normal and exponential datasets have similar results to the uniform dataset, so we omit them.

In general, the time efficiency of both learned indexes is significantly better than B+tree. Comparing CARMI with ALEX, which is also a learned index, we see that these two learned indexes perform roughly the same over synthetic datasets. However, CARMI significantly outperforms on these real-world datasets, achieving an average speedup of 1.2×/2.2× on read-only/read-write workloads. This demonstrates the effectiveness of data partitioning and cache-aware design over real-world datasets, in which data location is much harder to predict.

6.2.1 Read-only Workload. For read-only workloads, the lookup speed of CARMI is about 1.6-4.2× faster than B+tree and 1.2× faster on average than ALEX. Meanwhile, CARMI uses only 0.7× of memory space compared to B+tree (excluding YCSB[4] dataset).

It is interesting to compare CARMI with B+tree since their difference is only on the upper level. The significant speed gain of CARMI is mainly due to the following reasons: (a) CARMI uses fast model prediction instead of binary search. (b) the larger fanout of the nodes in CARMI reduces the depth of the index. Then, most data points are managed by leaf nodes directly under the root node, making the index structure flatter. To verify this, we also measure the average tree depth with respect to keys, where a single root node has a depth of 1. CARMI has an average tree depth of 2.1, while ALEX and B+tree have average tree depths of 2.5 and 7.3.

Comparing the two learned indexes, we find that the main difference is in the performance over OSMC/Face, in which CARMI achieves an average speedup of 1.5×. These two real-world datasets have highly non-linear local distribution, as shown in Figure 8, which invalidates ALEX's strategy of storing data points according to the predicted location and causes the index to grow deeper. Meanwhile, our construction algorithm can automatically select suitable nodes to obtain good performance. Section 6.3 shows similar results on other real-world datasets, indicating that this is not a coincidence.

In summary, CARMI outperforms B+tree and achieves similar or better performance than ALEX under read-only workloads, and the gain is more evident on real-world datasets.

6.2.2 Read-Write Workloads. For read-write workloads, CARMI achieves an average speedup of 2.0×/2.2× compared to B+tree and ALEX[5] while using 0.8×/1.2× space on average.

It is generally difficult for learned indexes to partition non-linear datasets evenly, leading to a large gap between the numbers of data points in each node. This results in additional expansion costs in ALEX during the insert operations. On the other hand, CARMI handles well even under such scenarios, using only 0.42× time of ALEX on lognormal/real-world datasets.

In summary, CARMI has a better time efficiency, uses a similar amount of space compared to ALEX, and has more significant advantages on non-linear and real-world datasets.

[3]We also provide an extended version of CARMI, the results of which are presented in our technical report [47].

[4]For the YCSB dataset, since CARMI uses the external leaf nodes, which only store a pointer of the location of data points, the space cost of 11.4 MB only counts the space of the tree structure itself and does not include the space used by external data.

[5]The insert of the YCSB protocol is to continuously insert new data points at the end of the dataset, and ALEX does not strictly follow this mode for insert in their paper. Thus, our reported number differs from the values in their paper.

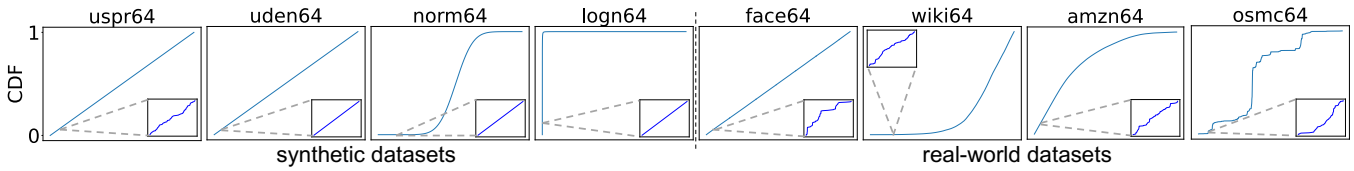

Figure 8: The CDFs of Datasets in SOSD [21].

6.3 SOSD Results

To further evaluate the performance of CARMI on practical datasets, we integrate CARMI into the SOSD benchmark [21, 31], a platform for testing learned indexes. We use all real-world datasets in it, each consisting of 200 million unsigned 32-bit/64-bit integers: **amzn** is book sale popularity data [4], **face** is the user IDs of Facebook [42], **wiki** is Wikipedia article edit timestamps [5], and **osmc** is derived from Open Street Maps [2]. SOSD performs 10 million lookups on each dataset, where lookup keys are uniformly chosen from the set of keys, and computes the average latency per lookup.

SOSD includes ten baselines: (a) **three learned indexes**: RMI, RS, and ALEX; (b) **three traditional indexes**: B+tree, FAST, and ART (FAST and ART do not support all the datasets as explained in SOSD [21]); (c) **three on-the-fly** algorithms that directly operate on a sorted array: BS, IS and TIP; (d) **one auxiliary index** that uses small auxiliary structures: RBS. Table 2 shows the average lookup time of indexes, where the results with the shortest latency and those slightly slower (within 20ns) are bolded.

Table 2: The Average Lookup Time (ns) on SOSD Platform.

	uint32		uint64			
(ns)	amzn	face	amzn	face	wiki	osmc
CARMI	**192.84**	**187.43**	**201.80**	**334.41**	**217.50**	**368.65**
RMI	265.43	274.53	266.54	**334.54**	**222.43**	402.32
RS	280.03	362.54	296.61	436.63	**218.43**	412.39
ALEX	**210.99**	434.69	251.32	496.48	289.81	499.04
RBS	325.43	312.39	385.96	**334.73**	335.75	529.06
FAST	246.03	228.79	N/A	N/A	N/A	N/A
ART	N/A	**182.36**	N/A	391.76	N/A	N/A
B+tree	529.43	524.54	601.23	592.43	608.42	599.43
BS	1014.5	983.49	1015.1	961.93	1002.3	987.24
TIP	731.97	880.12	750.89	1124.6	942.84	4773.8
IS	3852.7	1007.7	4103.9	1494.8	6836.4	66474

We also examine the distribution of datasets. As shown in Figure 8, a major difference between synthetic and real-world datasets is that synthetic datasets all have locally linear CDF, which is rare in real-world datasets. The highly non-linear local CDFs of real-world datasets make it difficult for prior solutions to accurately predict the location of individual records, leading to large-range searches in the last mile. In contrast, CARMI is designed based on the data partitioning view with a more cache-friendly leaf node layout. Thus, it is less penalized by these highly non-linear parts and thus performs better in practical datasets: CARMI can achieve an average speedup of 1.21× even compared to a well-tuned RMI.

Interestingly, most learned indexes do not outperform traditional indexes on particular real-world datasets, as shown in Table 2. Specifically, FAST takes only 237.41 ns on average on amzn32 and face32, while RMI, RS, and ALEX take 1.14×/1.35×/1.36× average lookup time, respectively. ART takes an average of 287.06 ns on face32 and face64, while RMI, RS, and ALEX take 1.06×/1.39×/1.62×, respectively. Despite this, CARMI's lookup latency is either the shortest or very near to the shortest of all indexes for all datasets.

We also report the average space usage of each index in Table 3. The space required by CARMI is comparable to that of traditional indexes with good performance, such as FAST and ART. Note that users can adjust the space cost of CARMI.

Table 3: Average Space Usage (MB) on SOSD.

	CARMI	RMI	RS	ALEX	RBS	
uint32	3918.30	2471.65	2299.63	3336.74	2289.82	
uint64	5481.44	3143.31	3064.33	4430.83	3052.76	
	FAST	ART	B+tree	BS	TIP	IS
uint32	5120.00	4596.52	2657.31	2288.82	2288.82	2288.82
uint64	N/A	5280.92	3540.52	3051.76	3051.76	3051.76

6.4 Tradeoff between Time and Space

Another advantage of our cost-based index construction algorithm is that one can now flexibly choose the balance point between time and space cost. As shown in Figure 9, by tuning the value of parameter λ in Problem 2, one can reduce the memory usage of indexes at the cost of increased read access latency. The curves of real-world datasets exhibit the characteristics of convex functions, but the optimal solution for each dataset depends on the actual data distributions. Therefore, we need to carefully tune λ according to practical scenarios to find the optimal solution, so as to obtain a more desired tradeoff between time and space.

6.5 Cost of Construction

We compare the time to construct each index for 1GB datasets using different values of the parameter *kDPThreshold*. We use this parameter to adjust the threshold for the size of the sub-datasets using the DP algorithm, thus obtaining a tradeoff between the construction time and the average lookup latency. As shown in Figure 10, the construction of CARMI with different settings can be finished within 0.3-3.2 minutes, while B+tree and ALEX take 10s and 20s on average, respectively. Although CARMI takes longer to build indexes than baselines, it is generally acceptable for most

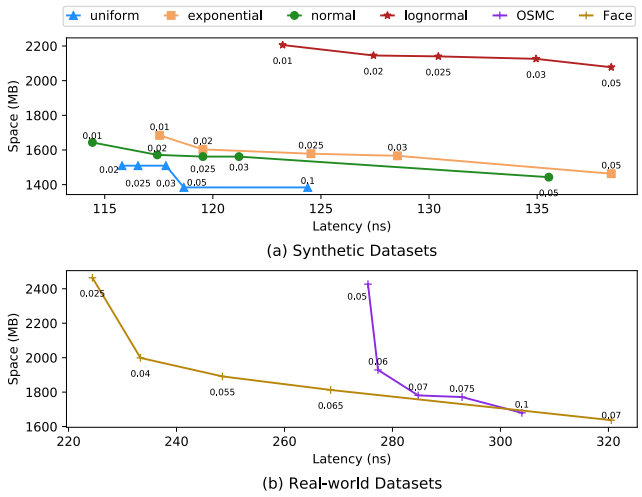

Figure 9: The Tradeoff Between Time and Space Cost

practical scenarios. Moreover, *kDPThreshold* has a slight impact on the average lookup latency, with the difference between 96 and 1024 being around 10 ns.

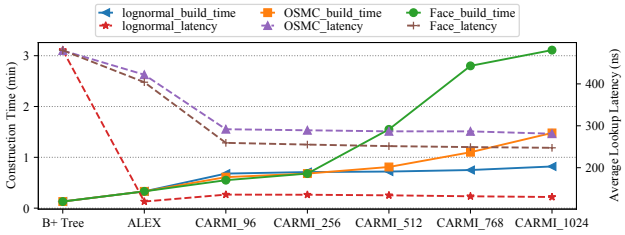

Figure 10: Construction/Lookup Time of Indexes.

To conclude, in scenarios where construction time is important, *kDPThreshold* can be tuned to reduce the construction time. While in most practical scenarios, the default value can be used directly.

6.6 Performance Breakdown

In this section, we investigate the contribution of each idea to the overall performance. We implement 4 variants of CARMI to simulate indexes with different ideas and perform read-only workloads on 5 datasets, including 3 synthetic datasets and 2 real-world datasets. A brief introduction of these variants is as follows:

- **RMI**: a two-layer RMI with an LR root node and 131072 external leaf nodes.
- **Greedy**: a dynamic index constructed by the greedy algorithm.
- **Greedy cache-aware**: a dynamic index equipped with the cache-aware design and greedy algorithm.
- **CARMI**: a complete CARMI equipped with all proposed ideas.

As shown in Figure 11, most of the benefits of the CARMI framework come from the data partitioning view, which results in an average speedup of 1.84× for the greedy index compared to RMI. Specifically, the greedy index only uses the greedy algorithm, so

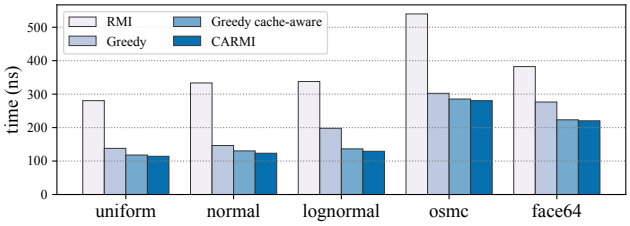

Figure 11: Latency of Four CARMIs.

each node is determined by the entropy-based notion of local efficiency. This means that larger fanouts can effectively reduce large-range searches in RMI, and four inner nodes can be mixed to handle different situations well to improve data partitioning effectiveness.

Next, in the greedy cache-aware index, we replace leaf nodes with CF array leaf nodes and cooperate with the prefetching mechanism to examine the effect of our cache-aware design. Due to the influence of data distributions and local partitions, the cache-aware design shows different effects. Among them, the speedup for lognormal and face64 datasets is more significant. In summary, this index still obtains an average speedup of 1.21× compared to the greedy index, which is the contribution of cache-aware design.

If sufficient resources are available, users can use the hybrid construction algorithm comprised of the greedy algorithm and DP algorithm to build better indexes. According to the average access time when the parameter *kDPThreshold* is 1024, as shown in Figure 11, the complete CARMI can reduce the average access time by about 10ns.

6.7 Robustness of CARMI

In this section, we simulate three groups of workload and data distribution shifts to examine the robustness of CARMI. First, we build indexes on the OSMC/Face datasets with history queries that obey a uniform distribution. Then we query them with eight access patterns: seven Zipfian distributions and one uniform distribution to simulate the situation where new sets of keys are queried. To simulate read-write workload shifts, we construct indexes with history queries under read-heavy and write-heavy workloads, respectively, and test them with write-heavy workloads. Next, we use the uniform dataset to build the index and perform a write-heavy workload by inserting the keys from the OSMC dataset to simulate the data distribution shifts. As shown in Figure 12, CARMI can still maintain good performance in these situations and is robust to workload and data distribution shifts.

7 DISCUSSION AND FUTURE WORK

7.1 CARMI in Disk/NVM

In our current implementation of CARMI, both inner nodes and leaf nodes are stored in memory. We can easily extend CARMI to involve disk operations. If we keep records on disk, we need to design a new type of leaf node that accesses a disk page instead of an in-memory data block. The corresponding time and space costs of this new type of node should also reflect the latency of

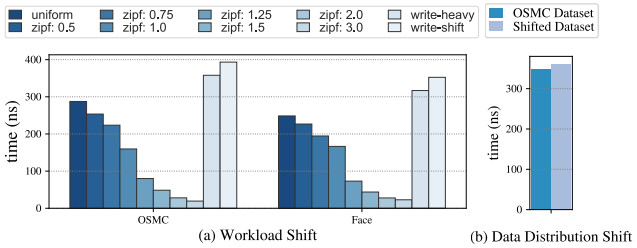

Figure 12: Robustness of Indexes in Different Situations.

disk operations. The current hybrid construction algorithm can be directly used without any change.

Moreover, with the rapid development of non-volatile memory (NVM) [26], memory capacity can be vastly expanded while retaining decently high access speed. CARMI can adapt to different storage devices by modifying the cost model.

7.2 Other Data Types

In our implementation of CARMI, we only support numerical key types. It is possible to extend CARMI to handle other types as well. Generally speaking, the existing RMI framework cannot readily handle strings since the distribution of strings is hard to fit well by their current models. Our perspective based on data partition can potentially address this difficulty. It is necessary to design new nodes for strings and make specific changes to the storage method. The hybrid construction algorithm and the cost model require slight modifications, but the main ideas and processes of this algorithm can still be applied.

8 RELATED WORK

In this section, we introduce some related works in the field of database index. Due to page limitation, some related works, including researches in the field of database that utilize ML techniques [17, 20, 23, 25, 32, 36, 44], are not discussed here and can be found in our technical report [47].

Traditional Indexes: Many researchers have optimized index structures in the past decades and have proposed many indexes to achieve good performance, such as B-tree, B+tree [7, 8], T tree [27], balanced B-tree [6], and red-black tree [10], etc. Since most indexes are stored in the main memory, CSS-tree [39] restricts node size to the cache line size and eliminates child nodes' pointers to utilize the cache, and CSB+ tree [40] is then proposed to support updates. ART [28] is an adaptive cardinality tree designed to reduce the number of cache misses. For further acceleration, pB+-tree [11] and FAST [19] use prefetching instructions and SIMD instructions, respectively. Masstree [30] effectively handles possible binary keys of any length, including keys with long shared prefixes.

Learned Indexes: Kraska et al. [24] propose a recursive model index (RMI), which uses ML models to build index structures. However, RMI does not support inserts. FITing-tree [16] uses linear models to replace leaf nodes of B-trees to compress the index. PGM-index [15], a compressed learned index with provable worst-case bounds, extends FITing-tree and provides an optimal method to find the piecewise linear models. These two indexes support inserts by additional buffers, and the performance can be improved. To support writes, ALEX [13] uses model-based inserts and gapped array leaf nodes to make room for future inserts. However, it requires large-range searches to accurately locate records, resulting in performance degradation on real-world datasets. In this paper, we adopt a data partitioning view and use a cost-based construction algorithm to select suitable node settings to build an updatable index. CARMI uses various types of nodes to handle different situations, thus maintaining good performance on real-world datasets.

CDFShop [33] uses Pareto analysis to find Pareto optimal configuration for the static two-layer RMI. They use a parameter search strategy, which is unsuitable for dynamic structures due to the exponentially large number of configurations in the search space. On the other hand, CARMI can automatically tune tree structures and model types at runtime through a new dynamic architecture and the cost-based algorithm.

Besides, RadixSpline [22] focuses more on building indexes in a single pass, and PLEX [41] is built on RS and retains only one hyperparameter for more convenient use. LIPP[45] uses entry types and a conflict degree metric to decide tree node layout. Mitzenmacher et al. [34] build a learning bloom filter that uses neural network models to predict whether keywords belong to a specific set. In addition, several other works apply the idea of the learned index to multi-dimensional indexes [14, 35] and spatial query processing [29, 37, 38, 43] to improve performance.

9 CONCLUSION

This paper conducts in-depth research on the basic framework of learned indexes (RMI), and argues that the inflexibility of data partitioning is an important reason for the performance degradation of RMIs in practical scenarios. To address this issue, we propose to view RMI construction from a data partitioning view and propose a general cache-aware RMI framework called CARMI. Specifically, we use the entropy metric to quantify the data partitioning effectiveness and propose a new cost model to characterize the performance of individual tree nodes, which helps to design more robust indexes. Furthermore, CARMI is equipped with a new memory layout that is more cache-friendly and uses a hybrid algorithm to automatically construct index structures for different datasets and workloads without manual tuning. Experimental results show that CARMI has an outstanding performance under various datasets and workloads, achieving an average of 2.2×/1.9× speedup compared to B+tree/ALEX, respectively, while using only 0.77× the memory space on average. This paper demonstrates that data partitioning is important for learned indexes during construction and that the new cost model can well characterize the performance of a single node, which is beneficial to improving the performance and flexibility of indexes. Finally, the CARMI framework is highly extensible and robust and can be applied to a wider range of scenarios if we design different types of nodes for it.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. We also thank Huanchen Zhang for helping revise the early sections.

REFERENCES

[1] [n.d.]. https://github.com/JiaoYiZhang/learned_index.
[2] [n.d.]. https://registry.opendata.aws/osm/.
[3] [n.d.]. https://panthema.net/2007/stx-btree/.
[4] [n.d.]. https://www.kaggle.com/ucffool/amazon-sales-rank-data-for-print-and-kindle-books.
[5] [n.d.]. http://dumps.wikimedia.org.
[6] Rudolf Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica* 1, 4 (1972), 290–306.
[7] R Bayer and E McCreight. 1970. ORGANIZATION AND MAINTENANCE OF LARGE. (1970).
[8] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software pioneers*. Springer, 245–262.
[9] Brian Beavis and Ian Dobbs. 1990. *Optimisation and stability theory for economic analysis*. Cambridge university press.
[10] Joan Boyar and Kim S Larsen. 1994. Efficient rebalancing of chromatic search trees. *J. Comput. System Sci.* 49, 3 (1994), 667–682.
[11] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. 2001. Improving index performance through prefetching. *ACM SIGMOD Record* 30, 2 (2001), 235–246.
[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
[13] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.
[14] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282* (2020).
[15] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.
[16] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.
[17] Abdullah Gani, Aisha Siddiqa, Shahaboddin Shamshirband, and Fariza Hanum. 2016. A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowledge and information systems* 46, 2 (2016), 241–284.
[18] Robert M Gray. 2011. *Entropy and information theory*. Springer Science & Business Media.
[19] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 339–350.
[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).
[21] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).
[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*. 5:1–5:5. https://doi.org/10.1145/3401071.3401659
[23] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. (2019).
[24] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.
[25] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).
[26] Martijn HR Lankhorst, Bas WSMM Ketelaars, and Robertus AM Wolters. 2005. Low-cost and nanoscale non-volatile memory concept for future silicon chips. *Nature materials* 4, 4 (2005), 347–352.
[27] Tobin J Lehman and Michael J Carey. 1985. *A study of index structures for main memory database management systems*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[28] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.
[29] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2119–2133.
[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.
[31] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.
[32] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212* (2018).
[33] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. Cdfshop: Exploring and optimizing learned index structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2789–2792.
[34] Michael Mitzenmacher. 2018. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884* (2018).
[35] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.
[36] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.
[37] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The case for learned spatial indexes. *arXiv preprint arXiv:2008.10349* (2020).
[38] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.
[39] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 78–89.
[40] Jun Rao and Kenneth A Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 475–486.
[41] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *arXiv preprint arXiv:2108.05117* (2021).
[42] Peter Van Sandt, Yannis Chronis, and Jignesh M Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data*. 36–53.
[43] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 569–574.
[44] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record* 45, 2 (2016), 17–22.
[45] Jiacheng Wu, Yong Zhang, and Shimin Chen. 2021. Updatable Learned Index with Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.
[46] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*. 1567–1581.
[47] Jiaoyi Zhang and Yihan Gao. 2021. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. arXiv:2103.00858 [cs.DB]