A Dynamic Characteristic Aware Index Structure Optimized for Real-world Datasets

JIN YANG, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of)
HEEJIN YOON, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of)
GYEONGCHAN YUN, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of)
SAM H. NOH, Department of Computer Science, Virginia Tech, Arlington, United States
YOUNG-RI CHOI, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of)

Many datasets in real life are complex and dynamic, that is, their key densities are varied over the whole key space and their key distributions change over time. It is challenging for an index structure to efficiently support all key operations for data management, in particular, search, insert, and scan, for such dynamic datasets. In this article, we present DyTIS (Dynamic dataset Targeted Index Structure), an index that targets dynamic datasets. DyTIS, although based on the structure of Extendible hashing, leverages the CDF of the key distribution of a dataset, and learns and adjusts its structure as the dataset grows. The key novelty behind DyTIS is to group keys by the natural key order and maintain keys in sorted order in each bucket to support scan operations within a hash index. We also define what we refer to as a dynamic dataset and propose a means to quantify its dynamic characteristics. Our experimental results show that DyTIS provides higher performance than the state-of-the-art learned index for the dynamic datasets considered. We also analyze the effects of the dynamic characteristics of datasets, including sequential datasets, as well as the effect of multiple threads on the performance of the indexes.

CCS Concepts: • **Information systems** → **Data structures**; **Data management systems;**

Additional Key Words and Phrases: Dynamic datasets, index structure, key distribution

ACM Reference Format:
Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. 2025. A Dynamic Characteristic Aware Index Structure Optimized for Real-world Datasets. *ACM Trans. Storage* 21, 2, Article 15 (February 2025), 32 pages. https://doi.org/10.1145/3707642

This research was partly supported by the Ministry of Science and ICT (MSIT), Korea, under the Information Technology Research Center support program (IITP-2024-2021-0-01817) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP), the National Research Foundation of Korea (NRF) (NRF-2023R1A2C2006432), IITP grants funded by the Korea government (MSIT) (RS-2024-00438729, Development of Full Lifecycle Privacy-Preserving Techniques using Anonymized Confidential Computing, and RS-2024-00459026, Energy-Aware Operating System for Disaggregated System), an NSF grant (2312785), and Samsung Electronics Co., Ltd.
Authors' Contact Information: Jin Yang, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of); e-mail: yangjin@unist.ac.kr; Heejin Yoon, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of); e-mail: heejin5178@unist.ac.kr; Gyeongchan Yun, Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of); e-mail: rugyoon@unist.ac.kr; Sam H. Noh, Department of Computer Science, Virginia Tech, Arlington, United States; e-mail: samhnoh@vt.edu; Young-ri Choi (Corresponding author), Computer Science and Engineering, UNIST, Ulsan, Korea (the Republic of); e-mail: ychoi@unist.ac.kr.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1553-3077/2025/02-ART15
https://doi.org/10.1145/3707642

1 Introduction

For in-memory data management systems, such as in-memory databases and key-value stores [4, 14, 41, 42, 68], the efficiency of the index structure is critical, strongly affecting the final latency of the systems [24, 54, 70, 78]. Moreover, even for disk-based data management systems where, previously, storage I/O latency had been the dominant performance bottleneck [5], the software layer (i.e., the index structure) becomes important with the recent advent of fast storage systems such as the Samsung Z-SSD [66], KIOXIA XL-FLASH [19], and CXL SSDs [40]. Traditional indexes such as B+-trees and hash indexes are excellent structures that have been embraced by data management systems. They are an integral part of systems today supporting three key operations, specifically, search, insert, and scan. Their limitations, however, are that each excels for one operation but may not support another efficiently. For example, while hash indexes perform superbly for individual key search, they do not support range queries well, whereas the B+-tree is the other way around. Recently, learned index structures have been proposed that excel for all three operations [24, 30, 70]. However, there is a catch here as well. Learned indexes need to be trained, and training, in recent work, has been done in the form of bulk loading [24, 30, 70]. When the characteristics of a dataset are static and simple, they are excellent. However, once this assumption breaks, how much to train and when to train again become difficult hurdles to overcome, potentially resulting in degraded performance.

Consider that in a data management system, a *dataset* is defined by a key distribution (i.e., keys inserted to the system) and the insertion order of the keys. Many datasets in real life are actually complex and dynamic such that the key distribution is not uniform and also changes as keys are inserted. In this article, we present DyTIS (Dynamic dataset Targeted Index Structure), an index structure that competitively supports all search, insert, and scan operations, does not require a training phase (i.e., bulk loading), and is especially effective for dynamic datasets. DyTIS, although based on **Extendible Hashing (EH)** [29], leverages the **Cumulative Distribution Function (CDF)** of the key distribution of a dataset, and learns and adjusts its structure as the dataset grows. The key novelty behind DyTIS is to group keys by the natural key order and maintain keys in sorted order in each bucket to support scan operations within a hash index. A natural question, then, is how to manage the hash index when the dataset becomes non-uniform, that is, unbalanced or skewed, which is typical of datasets. To efficiently handle such key distributions, DyTIS employs *remapping functions* that redistribute the non-uniform keys into a uniform distribution while preserving the natural order of the keys.

Aside from learned indexes, to the best of our knowledge, DyTIS is the first index structure that is simultaneously efficient for all search, insert, and scan operations. Unlike learned indexes, DyTIS focuses on realistic real-world datasets with various dynamic characteristics that have *complex key insertion order* such that while keys are inserted over time, the distribution of inserted keys continuously changes.

The main contributions of this work are as follows. First, we give our definition of a dynamic dataset and propose a means to quantify its dynamic characteristics. Second, we design an efficient index, DyTIS, which adjusts the CDF, as keys are inserted, to reflect the real key distribution. Finally, we implement DyTIS[1] and evaluate its performance by comparing it with three different dynamic indexing techniques over five different real-world datasets. Our experimental results demonstrate that DyTIS provides higher performance than the state-of-the-art learned index for the dynamic datasets considered. We also analyze the effects of the dynamic characteristics of the datasets, that is, the effects of non-uniform key distributions and changes of key distributions over

[1] https://github.com/unist-ssl/DyTIS

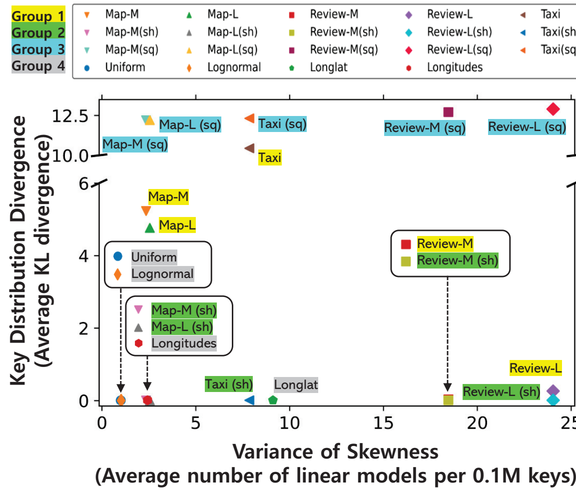


Fig. 1. Dynamic characteristics of various datasets.

time, on the performance of the indexes. These effects are further studied over sequential datasets, which have a very high degree of key distribution changes, and the performance of the indexes with multiple threads.

Through comprehensive analysis using the five real-world datasets, along with shuffled and sequential versions of the datasets, we show that the dynamic characteristics of datasets can significantly affect the insert, search, and scan performance as well as the performance with multiple threads, especially for indexes that leverage the key distribution of a dataset. For indexes based on the key distribution, the complex key insertion order makes learning the key distribution of a dataset harder. Moreover, for non-uniform datasets, it becomes difficult to approximate the key distribution accurately. Therefore, it is critical to consider the dynamic characteristics when designing such indexes for real-world datasets. We demonstrate that, for most cases, our proposed index, DyTIS, can provide robust insert, search, and scan performance over diverse dynamic datasets.

2 Motivation and Design Direction

2.1 Dynamic Datasets

Dynamic Dataset. We first define what we mean by a dynamic dataset and show how this is quantified. A dynamic dataset is defined based on two observations: one, that the whole key space of a dataset can consist of small key ranges with different key densities [13, 55, 82], and two, that the key densities of key ranges, and thus the key distribution of the dataset, change over time [50, 70]. The former is termed *skewness*, which refers to how dense keys are over a key range; if keys are dense (sparse), then skewness is high (low). The latter is termed *distribution divergence*, which captures the rate in which the key distribution of the dataset tends to change. We find that *variance of skewness* and *KDD* the key factors that strongly affect the performance of indexes that leverage the key distribution of a dataset. This is because the number of linear models used to approximate the CDF and the retraining/adjusting of the linear models have strong impact on the performance of these indexes, which we discuss in detail in the following. Figure 1 shows how various datasets fare in terms of these dynamic characteristics. (The datasets are discussed in detail in Section 5.2.) Note that the Uniform dataset shows no skewness and no divergence in key

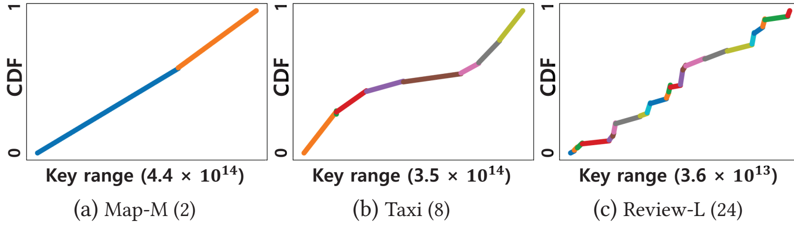


Fig. 2. Variance of skewness for three datasets (the number of models).

distribution throughout the whole key space. We now discuss how these values are quantified and how they are to be interpreted.

Variance of Skewness. Variance of skewness is a metric we use to capture how skewness changes throughout the dataset. This represents the complexity of approximating the CDF of the key distribution of the key space. Considering that piecewise linear regression is used to approximate the CDF [80], if the skewness of each small key range is highly varied, a large number of linear models are needed. Figure 2 shows how the number of linear models varies for the Map-M, Taxi, and Review-L datasets that show low, medium, and high variance of skewness, respectively, where lines with different colors represent different linear models. Hereafter, for simplicity, we use skewness to refer to variance of skewness.

To quantify skewness for a dataset, we take the average number of linear models used to approximate the CDF for a fixed number of keys per key range. Note that we use a fixed number of keys, then average them to normalize their values, as real-world datasets will all differ in size. The approximated CDF is measured by the maximum error-bounded PLR (Piecewise Linear Representation) technique [80].[2]

Key Distribution Divergence. For many real-world datasets, keys are inserted to a data management system in dynamic patterns. For example, in map datasets where keys are usually based on the longitudes and latitudes of locations [63], data with similar longitude and latitude values may be inserted consecutively as a bulk, whereas in sensing or weather-related datasets [11], data may be inserted with some periodic patterns such as diurnality and seasonality, and, as a final example, in datasets where a timestamp is used as a part of keys [32] such as time-series datasets, keys may be inserted sequentially. Therefore, the distribution of data (i.e., inserted keys) typically changes over time.

Key Distribution Divergence (KDD) captures how vastly the distributions change. It is measured by adopting the **Kullback–Leibler (KL)** divergence [49], which measures the difference between two probability distributions, where higher KL divergence value means that the two probability distributions are more different. To attain KDD, we divide the whole dataset into sub-datasets with a fixed number of keys and assume that each key in the sub-dataset is a random variable following the same discrete probability distribution. Then, the average of the KL divergence of every two consecutive sub-datasets is taken as the KDD for that dataset. To compute the KL divergence of two sub-datasets, we approximate the probability distribution of each sub-dataset using a histogram such that the key range of its histogram is determined by the minimum and maximum key values from the two sub-datasets. Note that for these metrics, the number of keys considered per key range and the size of a sub-dataset need to be chosen. While we choose 0.1 million for both, we also find the trend of the computed metrics are largely insensitive to this choice.

[2]https://github.com/RyanMarcus/plr. In this work, to construct the PLR of a dataset, the error bound is set such that the Uniform dataset only needs one linear model.

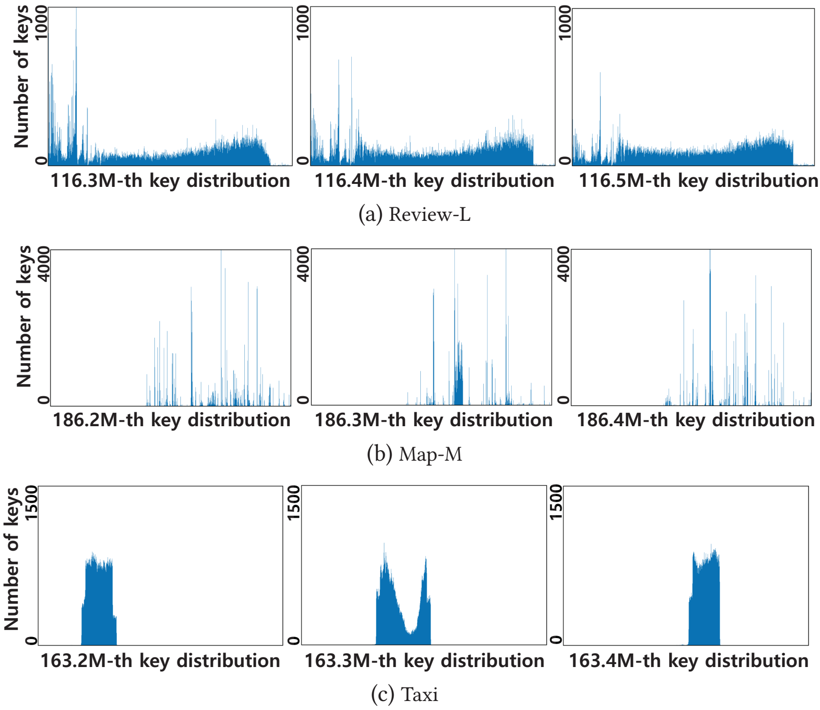


Fig. 3. KDD for three datasets.

Figure 3 shows how the key distributions diverge over three consecutive sub-datasets for Review-L, Map-M, and Taxi that represent datasets with low, middle, and high KDD values, respectively. We observe that the three key distributions of Review-L are virtually all the same, whereas for those of Taxi, the three show quite different distributions even to the naked eye.

Revisiting Figure 1. We now revisit Figure 1, where there are four groups of datasets. Group 1 contains the dynamic datasets that we use throughout our experiments, whereas those in Group 2, marked with (sh), are the shuffled version of those in Group 1, and Group 3, marked with (sq), are the sequential version of those in Group 1. The shuffled datasets contain keys that are randomly shuffled so that the keys in the key space are inserted uniformly over time, whereas the sequential datasets contain keys that are sorted in increasing order so that the keys are inserted sequentially over time.

We make the following observations of Groups 1, 2, and 3. First, real-world datasets (Group 1) show varying degrees of dynamism such as high skewness and low KDD (Review-M/L), to low skewness and moderate KDD (Map-M/L), to moderate skewness and high KDD (Taxi). Second, shuffled datasets (Group 2) have the effect of lowering the KDD value, that is, stabilizing the distribution of the keys. Third, sequential datasets (Group 3) have the effect of increasing the KDD value, that is, maximizing the key distribution change. In Figure 1, the shuffled datasets have similar KDD values close to 0, whereas the sequential datasets have similar KDD values of approximately 12.5.

For datasets in Group 4, we observe little dynamic characteristics having both low skewness and low KDD. The Uniform dataset consists of keys generated by YCSB [18] with a uniform distribution. The Lognormal, Longlat, and Longitudes datasets are the same ones used in ALEX [24].[3]

[3]https://github.com/microsoft/ALEX

The Lognormal dataset is generated according to a lognormal distribution, whereas the Longlat and Longitudes datasets are generated from OpenStreetMap [63]. It is important to note that the keys in all the datasets used in ALEX [24] are randomly shuffled, resulting in uniform insertion over time and eliminating the dynamic key distribution shift (i.e., making the KDD close to zero). Therefore, the Longlat dataset, which combines longitudes and latitudes similarly to Map-M and Map-L datasets in Group 1, is categorized into Group 4. We emphasize here that the target dataset of our study, Group 1, is different from Group 4 that was used in previous studies [21, 24, 48].

2.2 DyTIS Design Focus and Philosophy

An index structure that will perform well for scan as well as insert and search operations has long been sought after. Recent developments seem to have finally found the solution in learned index structures [24, 30, 48, 70]. A learned index leverages the key distribution to build an index structure for keys stored in a sorted array. It uses machine learning algorithms such as neural networks and linear regression to approximate the CDF for the key distribution, and this CDF is used to predict the position of a given key in the sorted array. Of learned indexes, ALEX [24], a representative approach, maintains a data structure that looks like a B+-tree, where each internal node stores a linear model (i.e., an approximated CDF) and a pointer array to children nodes, and each leaf node (i.e., data node) stores a linear model and two arrays for keys and values. Unlike the original learned index [48] with the static **recursive-model index (RMI)**, ALEX is dynamically adapted such that depending on the dataset, each data node can have a different depth. The index maintaining operations such as split, which is conceptually similar to the B+-tree split, or expansion, where the data node size increases with a scaled or retrained linear model, are selected by a cost model that is learned with bulk loading and, thereafter, relearned during runtime. Through the adaptive RMI, ALEX can find a queried key by passing through multiple linear models in internal and data nodes.

Unfortunately, a couple of key constraints remain in learned indexes, especially for real-world datasets [13, 50, 55, 70, 82]. The first is that the model must first be pre-trained (i.e., bulk loaded). Currently, bulk loading is done manually in an ad-hoc manner, as it is difficult to know how many keys should be used for learning so that the model sufficiently reflects the real distribution. This limitation naturally requires retraining of the model if the key distribution changes, so as not to degrade performance. Second, existing learned indexes leverage a hierarchy of models for either the whole key space [48, 70] (i.e., RMI) or certain key ranges with high key density [24] (i.e., adaptive RMI). Therefore, model retraining in a node may result in a cascade of retraining of other models, such as the model of a parent or grandparent node, which will be an expensive operation. Moreover, the quality of the model in the upper hierarchy (e.g., the model of the root) will have stronger effect on the overall performance. Third, considering that a large number of linear models must be built to approximate the CDF for a dataset with high skewness, existing learned indexes that allow only one linear model for a node [24, 48] need to create a large number of nodes for such a dataset, resulting in a complex index structure.

DyTIS targets dynamic datasets with high skewness and/or high KDD. When designing DyTIS, we took into account the valuable lessons from learned indexes. However, the following unique considerations also needed to be distilled into the design to efficiently support the dynamic nature of datasets:

(1) *Free of bulk loading:* DyTIS learns and adjusts the CDF incrementally as keys are inserted without bulk loading overhead and also without performance degradation caused by inaccurate models built during bulk loading.

(2) *Local model retraining:* The cost of model retraining should be low to handle high KDD. DyTIS enables models for particular key ranges to be retrained locally and independently, making it more suitable for dynamic datasets.

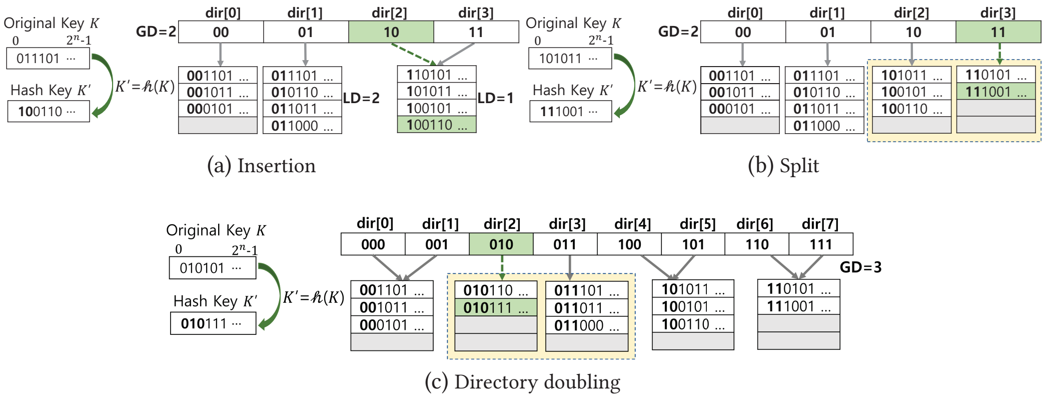


Fig. 4. Extendible hashing.

(3) *Multiple models per node:* To accommodate key ranges with high skewness more efficiently, DyTIS allows a node to have multiple models.

3 DyTIS: The Index Structure

3.1 Extendible Hashing

EH is a dynamic indexing structure that can *grow* and *shrink* as the workload dynamically changes [29]. It is unlike traditional hashing structures that require *rehashing* [23, 39, 59, 64, 65]. Figure 4 shows EH, in which n is the number of bits used for a key K, and a *pseudo-key* $K' = h(K)$, where h is the hash function, is used for indexing [29]. In EH, the hash table is organized with a *directory* and *buckets*. A directory (i.e., *dir*) is an array where prefixes (i.e., the **most significant bits (MSBs)**) or suffixes (i.e., the **least significant bits (LSBs)**) of the pseudo-key are used as an index. In Figure 4(a), two MSBs of a pseudo-key are used for the directory. The directory has a *global depth GD* that indicates how many MSBs are used as an index. Thus, *GD* determines the size of the directory.

Each entry in the directory points to a bucket that can store a fixed number of key-value pairs. Note that a value in each pair can be a pointer to an actual value associated with the key. When inserting a key-value pair, *GD* MSBs of its hash key are first used to find the directory entry, and then it is stored in the bucket pointed to by the directory entry (if there is free space). For example, Figure 4(a) shows the state after inserting the key "011101 ..." to the hashing index. Each bucket has a *local depth LD*, where $LD \leq GD$, indicating that this bucket contains all keys starting with *LD* MSBs of its associated directory index. Thus, two or more entries of the directory can point to the same bucket.

When a bucket finally overfills, the bucket must be split into two buckets. There are two situations when a bucket splits. The first is when $LD < GD$, which means that multiple directory entries were pointing to the bucket, as in the case of the rightmost bucket in Figure 4(a). For this case, *GD* simply remains the same and *LD* of both buckets is increased by 1, whereas the contents of the original bucket are adjusted according to the $LD + 1$ MSBs of the pseudo-keys. For example, the bucket pointed to by $dir[2]$ and $dir[3]$ in Figure 4(a) is split into two buckets in Figure 4(b) such that keys with prefixes "10" are stored in one bucket while keys with prefixes "11" are stored in the other bucket.

The second situation is when $GD = LD$. In this case, one bucket is pointed to by only one directory entry. Thus, the directory has to be expanded to accommodate the split, which is referred to as *directory doubling*. For this, the number of bits used as an index, that is, *GD*, must first be

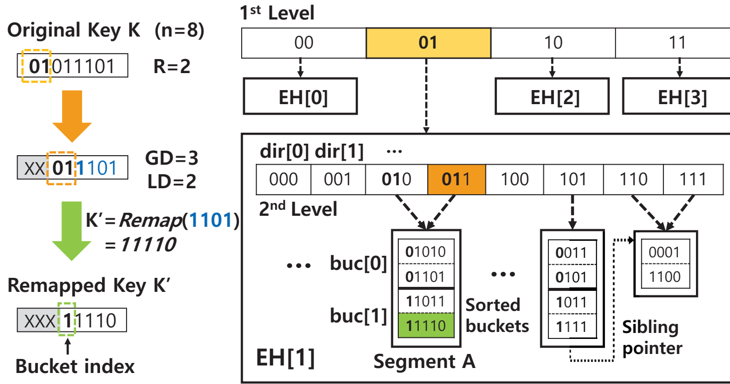


Fig. 5. DyTIS architecture.

increased by 1. Then, the dividing of the contents and the *LD* values are adjusted in the same manner as the first case. For example, as one more key is added to the bucket pointed to by *dir*[1] in Figure 4(b), the directory is doubled such that the values of *LD* for the buckets pointed to by *dir*[2] and *dir*[3] become 3, whereas those for other buckets remain unchanged, as shown in Figure 4(c).

CCEH [60] is a variant of EH that uses a three-level structure of a directory, intermediate segments composed of a number of buckets, and buckets. CCEH utilizes MSBs of a pseudo-key as a segment index, whereas it utilizes LSBs of the key as a bucket index in the segment. Having intermediate segments is useful in reducing the overhead of directory doubling. In DyTIS, we adopt this three-level structure of CCEH, but we do not use LSBs of the keys since keys are stored in sorted order to support scan operations.

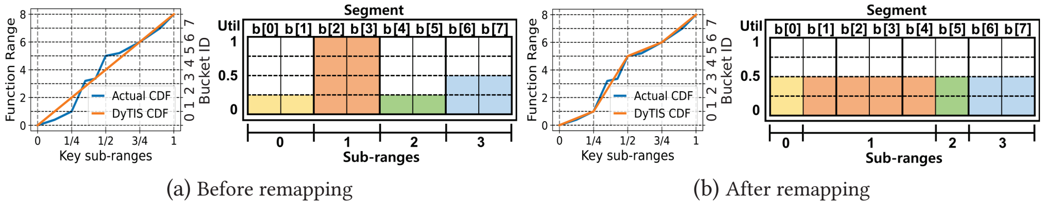
We design DyTIS based on EH to leverage its efficient insertion and search with constant lookup time $O(1)$, unlike tree-based structures. In addition, as it dynamically grows and shrinks as a workload changes without rehashing a hash table, it is a suitable base index structure to support dynamic datasets.

3.2 System Overview

DyTIS has an Extendible hash table structure, but uses *remapped* keys as pseudo-keys, rather than hash keys, to preserve the natural order of keys. It also leverages CDFs like learned indexes, but, as we will see, not in exactly the same manner.

Figure 5 presents the architecture of DyTIS as well as a walk-through example that we will use. DyTIS consists of multiple EH tables and has a key range of $[0,2^n)$, where n is the number of bits used for a key. It has a two-level architecture so that the first level has an array of EHs. The first level *statically* divides the entire key range into 2^R sub-ranges based on R MSBs of the key so that each EH handles, in the second level, keys only in a given sub-range. This two-level structure allows each EH to deal with a smaller number of keys within the sub-range individually. Therefore, the range of keys for each EH becomes $[0,2^{n-R})$ set by the $(n-R)$ LSBs from the keys. In Figure 5, $n = 8$ and $R = 2$, and thus each EH makes use of the six LSBs of the key.

An EH table in the second level has a three-level structure of a directory, segments, and buckets as in CCEH [60]. Each bucket is composed of an array of keys and an array of values such that a key and its value are stored in sorted order by the values of keys separately in the two different arrays [24]. In addition, segments within the same EH maintain a sibling pointer to the next segment to accelerate scan performance. Note that buckets have a fixed size. In this three-level structure, each segment contains all keys starting with the same *LD* MSBs, where *LD* is the local



(a) Before remapping

(b) After remapping

Fig. 6. Adjusting the remapping functions.

depth of the segment. In the example, $LD = 2$ and thus all keys with MSBs 01 are in Segment A. Consequently, the key range of a segment becomes $[0,2^{n-R-LD})$ set by the $(n - R - LD)$ LSBs of the keys. In Figure 5, the four LSBs (e.g., $1101_{(2)}$ for key $01011101_{(2)}$) are used, resulting in a range of $[0,2^4)$. Note that in Figure 5, instead of original keys, remapped keys, which are discussed next, are shown in Segment A.

We have, so far, only described how the bits (actually their positions) that comprise the key are used within DyTIS. Now we describe the key idea behind DyTIS, which is to use the raw keys themselves as pseudo-keys, rather than the hash keys, enabling efficient scan operations. However, the key predicament here is that non-uniform key distributions may result in a huge and imbalanced directory where a large number of keys exist in a certain range, whereas in the rest of the ranges, only a few keys exist so that many entries in the directory are pointing to the same bucket. Such situations will result in high directory doubling cost. To remedy this issue, DyTIS uses *remapping* functions that redistribute keys with any distribution uniformly. In other words, in each segment within the EH, if DyTIS deems the keys within the segment to be skewed, *remapping* spreads the keys evenly over the segment. This is depicted as K' = Remap(1101) in Figure 5.

DyTIS leverages the CDF for a key distribution as the remapping function of each segment. We exploit the fact that since the CDF is a monotonic increasing function, each key K has a unique K' value, and thus all the keys with any distribution mapped using its CDF are mapped to a uniform distribution. In DyTIS, the CDF for the remapping function is approximated as a set of linear functions (similarly to other learned index techniques [24, 32, 46, 70]). In other words, given the key range of a segment, the key range is *statically* divided into 2^P sub-ranges based on P MSBs of the given keys, and each sub-range is associated with a linear function.

Initially, we assume that keys are uniformly distributed within each sub-range, initializing the linear function as $K' = K$, as shown in the left sub-figure of Figure 6(a) where there are four sub-ranges for a segment. As keys are inserted into EH, we incrementally and dynamically adjust the linear function (i.e., slope and intercept) for each sub-range to reflect the real distribution as in the left sub-figure of Figure 6(b). For a sub-range where more keys are inserted than predicted (e.g., [1/4, 1/2)), we increase the slope of its linear function, whereas for a sub-range where less keys are inserted than predicted (e.g., [0, 1/4)), we decrease the slope of the function. Moreover, intercept values of linear functions need to be modified such that the functions are connected to handle the entire range of the remapped keys. (Note that the right sub-figures of Figure 6(a) and (b) will be discussed in Section 3.3.)

Along with remapping functions, to handle possibly highly skewed key distributions without directory doubling or split, DyTIS also exploits segments with varying sizes such that DyTIS can dynamically adjust the number of buckets that a segment can hold. We now discuss how a key is rescaled, that is, how the bucket index of a key is computed, within a segment with multiple buckets. Consider a segment where $K' = F(K)$, with K and K' being the raw key and the remapped key, respectively, and F is the scaled approximate CDF for the distribution to the function range

(i.e., the range of y-axis of F). The function domain (i.e., the range of x-axis of F) is the same as the key range of the corresponding segment, which is $[0,2^{n-R-LD})$, whereas, initially, when the segment has only one bucket, the function range is $[0,2^{n-R-LD})$. As the number of buckets in the segment, B, increases, this range will proportionally extend to $[0, B \times 2^{n-R-LD})$. Accordingly, function range $[i \times 2^{n-R-LD}, (i+1) \times 2^{n-R-LD})$ corresponds to a bucket index i, where $0 \leq i < B$. Therefore, with a given remapped key K', the bucket index, where K' is stored, can be computed by dividing K' with 2^{n-R-LD}, the size of the function domain. Note that in general, the number of buckets in a segment need not be a multiple of 2 as described in Section 3.3, and thus only the quotient is used since the bucket index is an integer.

In addition, note that in Figure 6(a), the domain of the function are both given as [0,1). A similar simplification is made for Figure 7 with the function domain and range. While this is to simplify the explanations, the actual function domain and range are as discussed previously.

3.3 Individual Operations

Search. We explain how the search operation works using the walk-through example in Figure 5. In this figure, we have an 8 bit key (i.e., $n = 8$) and use 2 bits in the first level (i.e., $R = 2$). Assume we want to find key $K = 01011101_{(2)}$. We find EH[1] using two MSBs ($01_{(2)}$) of K. Within EH[1], the $n-R$ LSBs ($011101_{(2)}$) of K are used to compute the index in the directory, whose size is always a multiple of 2, using GD MSBs. As GD is 3, we use three MSBs, $011_{(2)}$, to find the directory index, which points to segment A. Since LD of segment A is 2, its key range is $[0,2^4)$ and the range of the remapping function is $[0,2^5)$ as it has two buckets, where $b[0]$ and $b[1]$ correspond to $[0, 10000_{(2)})$ and $[10000_{(2)}, 100000_{(2)})$, respectively. Then, we remap the four LSBs of K ($1101_{(2)}$) to a new key $K' = 11110_{(2)}$ and find the bucket index to be 1 by dividing K' by $2^4(10000_{(2)})$.

Once the bucket is located, an exponential search [24] is performed on the key array. If K is found, DyTIS reads the value of K stored at the same location in the value array and returns it. Otherwise, it returns "not exist."

Scan. For a scan operation, a starting key K and a scan key range c are given. DyTIS first finds the index of the directory entry d that could contain K. It finds the starting position by searching for K or the smallest key larger than K in the segment of $dir[d]$. If not found in that segment, the starting position is the first key in the first bucket of the segment of $dir[d+1]$. It then linearly reads c keys, possibly continuing on to subsequent buckets, segments, or EHs, or until it reaches the end of the index. For scanning multiple segments within an EH, sibling pointers of segments are used.

Insertion. DyTIS inherits the basic schemes of split and directory doubling of EH. DyTIS, however, employs additional schemes, remapping, and expansion, to handle non-uniform key distributions. DyTIS is required to use the MSBs of remapped keys so that keys can be clustered by the natural key order in this way. Note that a remapped key is used to find the bucket index, but the raw key is stored in the bucket. For the operations that alter the indexing structure, we utilize a threshold U_t that determines whether the utilization of a segment is high or low.

Algorithm 1 shows the insertion algorithm of DyTIS. To insert a key K, DyTIS first finds bucket b of segment s where the key will be stored, as in the search operation. DyTIS checks if key K already exists. If so, it performs an in-place update for K such that the updated value will be stored at the same location in the value array of b. Otherwise, if b is not full, DyTIS inserts K and the value of K to b (Line 1), which could shift keys larger than K and their values to maintain the sorted order.

If b is full, then there is no space to insert K. Depending on the local depth of segment s, and the current utilization of s, DyTIS performs one of the following operations: split, remapping,

ALGORITHM 1: Insert(*key K*, *segment s*, *bucket b*)

```
1:  If b in s is not full, insert K to b and return;
2:  LD = a local depth of s;
3:  U_s = utilization of s;
4:  if LD < GD then
5:      if U_s > U_t then
6:          Do Split;
7:      else
8:          Do Remapping;
9:          If it fails, do split;
10:     end if
11: else if LD == GD then
12:     if U_s > U_t then
13:         Do Expansion;
14:     else
15:         Do Remapping;
16:     end if
17:     if the above fails then
18:         Do Directory Doubling;
19:     end if
20: end if
```

expansion, and doubling. Note that at the beginning, DyTIS only performs basic schemes of EH until EH reaches a certain local depth, having a number of inserted keys, before leveraging the key distribution to build an index structure.

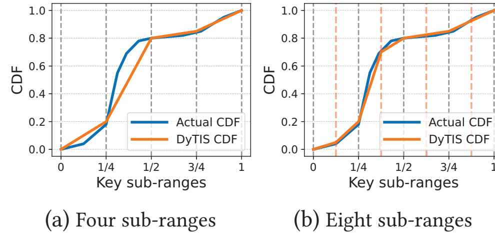
Low utilization of segment s indicates that the key distribution of s is currently non-uniform (because the bucket of interest is full, yet the overall utilization of the segment is low). Therefore, in such cases, regardless of $LD < GD$ or $LD == GD$, DyTIS attempts to adjust the remapping functions of s to alleviate the skewed distribution of keys in s (Lines 8 and 15). While adjusting the remapping functions, it might increase the size of segment s.

However, high utilization of s means that the current remapping functions for s are distributing keys uniformly to a good degree, but s lacks space to store more keys. Therefore, when $LD < GD$, it performs a split (Line 6). Yet, when $LD == GD$, DyTIS performs an expansion, which doubles the size of s while scaling the remapping functions (Line 13). This has the same effect of directory doubling for keys in segment s. As there is a cap on the segment size that is dependent on its local depth, which is described in the last part of this section, the expansion and remapping of a segment can fail if DyTIS cannot increase the size of s due to this limit. To handle the failure, when $LD == GD$, it performs directory doubling (Line 18). When $LD < GD$, if remapping fails, it performs a split (Line 9). Note that remapping and expansion do not increase data movement (i.e., memory copy), as they are performed to avoid segment splits that incur data movement (which is discussed in Section 5).

Next, we discuss each of the operations in Algorithm 1.

Directory Doubling (Line 18). DyTIS uses the same doubling mechanism as EH described previously.

Split (Lines 6 and 9). DyTIS uses a similar split mechanism as EH, but with the following difference. When the original segment is split into two new segments, it is possible that one of the



(a) Four sub-ranges (b) Eight sub-ranges

Fig. 7. Dividing a sub-range into smaller ones.

new segments may contain most of the keys, if the keys are skewed. Therefore, for each of the new segments, DyTIS first computes the segment size that will accommodate the keys of a sub-range from the original segment and then doubles its size, while keeping the slope(s) of the remapping function(s) of the sub-range for the new segment. For example, assume that the original segment has four buckets, where one bucket is used to store keys from the left half of its key range and the other three buckets are used to store keys from the right half of its key range. Then the segments are doubled such that one segment will have two buckets, whereas the other will have six buckets.

Expansion (Line 13). DyTIS simply doubles the size while scaling the remapping functions (i.e., doubling the slope). Therefore, a certain key range that originally uses one bucket will use two buckets after the expansion.

Remapping (Lines 8 and 15). Recall that for a segment s, we divide its key range into multiple sub-ranges. Initially, a segment has one sub-range but may be divided into multiple sub-ranges as in the following discussion. Consider that we are trying to insert key K into bucket b in segment s but have found that b is full and the utilization of s is low. It is at this condition that we are performing remapping. Thus, DyTIS first checks if the sub-ranges of s are fine-grained enough such that each linear function for each sub-range, denoted r, truly reflects the real CDF of r. For example, Figure 7(a) illustrates a case where the DyTIS CDF in [1/4, 1/2) does not approximate the real CDF accurately. In this sub-range, many keys exist in the left half of the sub-range while only a few keys exist in the right half. In such cases, the DyTIS CDF is too coarse and cannot represent the skewness of the sub-range. This results in the utilization of r being low but the bucket in which K is to be inserted being full. To remedy this, remapping is employed. In remapping, we partition the key range of s into smaller sub-ranges until the target sub-range to which K will belong has utilization larger than U_t. Figure 7(b) shows the case where the four sub-ranges in Figure 7(a) are divided into eight sub-ranges. We observe that DyTIS CDF for [1/4, 3/8) and [3/8, 1/2) now becomes closer to the real CDF. For this, the remapping functions also need to be adjusted.

Adjusting the remapping functions, generally speaking, is done in the following manner. If there exists a sub-range whose utilization is low, then DyTIS simply steals some number of buckets from that sub-range and gives those buckets to the target sub-range t with high utilization.

Let us illustrate how remapping functions are adjusted using the example in Figure 6. In this example, $U_t = 0.5$. In the figure, a segment with eight buckets (b[0] to b[7]) is divided into four sub-ranges, having each sub-range correspond to a slope of 8 as in Figure 6(a). To be remapped as Figure 6(b), DyTIS steals two buckets, one each from sub-ranges 0 and 2 (i.e., [0,1/4) and [1/2, 3/4)) whose utilization is 0.25, and gives them to sub-range 2 (i.e., [1/4, 1/2)), altering the remapping functions as follows. The slopes of sub-ranges 0, 1, 2, and 3 are first computed as 4, 16, 4, and 8, and the intercepts of the sub-ranges are computed such that the remapping functions are started at (0,0), (1/4,1), (1/2, 5), and (3/4, 6), respectively. After remapping, sub-range 1 can use four buckets, lowering the utilization of the sub-range to 0.5 and making it the same as the other sub-ranges.

Specifically, in DyTIS, we attempt to double the number of buckets for the target sub-range t. To do so, we compute how many buckets we can steal from other sub-ranges in segment s based on the utilization of a sub-range. For each sub-range (that is not t) whose utilization is less than U_t, we compute the minimum number of needed buckets. Based on this, we compute the number of buckets that this sub-range can give to t. If we can steal all the needed buckets from other sub-ranges, we adjust the remapping functions accordingly.

The preceding process can fail if the utilization of other sub-ranges is also high, and thus DyTIS is prevented from stealing the needed buckets from other sub-ranges. In this case, DyTIS increases the size of segment s such that the number of buckets for sub-range t is doubled. Therefore, the slope of the remapping function for sub-range t is doubled, and thus the overall function range for segment s is increased proportionally to the number of buckets in s.

Once the new remapping functions are computed along with the size of the segment, DyTIS creates a new segment and copies each key from the old segment to the new segment using the new remapping functions. Once this is done, the segment location that points to the old segment, which is stored in the directory entry, is made to point to the new segment. Then, the old segment is deleted.

Deletion. To delete a key K, DyTIS searches for the bucket where K is stored and deletes it from the bucket. This will shift keys that are larger than K and also their values, if any. Similar to ALEX [24], if deletion causes high under-utilization of a bucket, the bucket can be merged with others, reducing the size of the segment.[4] This process is similar to remapping but in the opposite direction.

Selecting a Segment Size. DyTIS basically imposes a limit on the segment size that is dependent on the local depth of the segment. This limit doubles when the local depth increases by 1 so that with a larger local depth, a larger number of keys can be stored in a segment. The effect of the segment size on performance will differ depending on the type of key distribution. With highly skewed keys, DyTIS will invoke remapping frequently, and thus a large segment size will increase remapping overhead. With uniform keys, then DyTIS leverages expansion, and thus a large segment size is beneficial, as more keys can be handled without increasing the local or global depth. DyTIS optimizes performance considering both cases as follows. Assume DyTIS starts to perform remapping and expansion at local depth L. Until DyTIS reaches a certain local depth L' where $L' > L$, it keeps track of the occurrences of the expansion operations. At local depth L', if the portion of segments where expansion is performed is large, this means that the keys are uniformly distributed. Thus, DyTIS increases the limit on the segment size at L'. Note that the value of L' needs to be selected so that DyTIS can collect sufficient information about the occurrences of split, remapping, and expansion operations, but, at the same time, early enough such that not too many doubling operations occur. For this, we empirically decide that $L' = L + 2$ and use this in our implementation.

3.4 Concurrency

With the advance of multi-core technologies, a server node commonly used for data management systems can leverage a large number of cores, which requires efficient concurrent accesses to the index structure [16, 22, 36, 37, 54, 70, 71]. Providing efficient concurrency support is critical in sustaining performance scalability as the number of cores continues to increase [22, 28, 54]. Concurrency is supported in DyTIS so that it can be used for a multi-threaded system such as

[4] As in ALEX [24], the merge operation is not implemented in DyTIS for simplicity. For XIndex, the merge operation is limited to integrating the delta index with the data array within an individual group node. Therefore, the merge operation is not performed for deletion similarly to ALEX and DyTIS.

Memcached [1, 31]. Note that storage systems, developed for distributed clusters and/or multi-core servers, may leverage multiple single-threaded engines for data access as in H-Store [6, 42] and Redis Cluster [3, 14]. Such systems may also use the single-threaded version of DyTIS that does not use locks instead of the multi-threaded one discussed in this section. Recall that we adopt the structure of CCEH [60], which supports concurrent accesses based on the two-level locking scheme adapted from Ellis [26]. Inherently, DyTIS adopts two levels of locking for each EH. At the high level, the directory array *dir* needs to be synchronized, that is, at the EH level, whereas at the lower level, we synchronize at the segment level. For segments, synchronization is done through a metadata container that we call a *segment object*, which contains information such as remapping functions of the segment and a pointer to a data array where key-value pairs are actually stored.

Using reader/writer locks, synchronization is administered only at the segment level when performing insert operations such as the normal insert (which inserts a key to a bucket without altering the indexing structure), remapping, and expansion, as these operations only change values of the information inside a segment object. Similarly, for search and scan operations, segment-level locking is used. For scan, multiple segments, within or across EHs, are locked one by one over the scan range, and once done, all the locks are released. After accessing the segment containing the starting key, the next segment is accessed through the sibling pointer, ensuring data consistency without using the EH-level lock even in the presence of concurrent insertions that modify the directory.

For insert operations that change the structure such as the split, which creates a new segment object and then updates the segment object pointer to the new one, both segment and EH-level locks are used to prevent other threads from performing directory doubling. Similarly, directory doubling or updating a sibling pointer between two segments requires segment and EH-level lock synchronization. Note that when both EH and segment locks are needed in the preceding cases, the segment lock is always taken first, followed by the EH lock. This order of locking ensures that deadlocks do not occur.

Note that CCEH leverages concurrency at finer grains of buckets within segments. We also explored this but found that performance of DyTIS generally degrades. Our analysis shows that this is due to the overhead of additional memory for the fine-grained locks and the handling of segments with variable sizes, which is unlike CCEH that uses a fixed segment size.

4 Implementation

Sequential Dataset Handling. DyTIS is implemented to detect and efficiently handle sequential datasets such as time-series datasets [9, 27, 35]. A sequential dataset can be identified as all keys are guaranteed to be inserted in increasing order. Initially, DyTIS assumes that a dataset is a sequential one. In the single-threaded version of DyTIS, to determine if a dataset is sequential, DyTIS basically compares the newly inserted key with the previously inserted key. If the new key is smaller than the previously inserted key, DyTIS can conclude that the dataset is not sequential. Otherwise, the dataset remains to be considered as a sequential dataset.

For sequential datasets, when inserting a key K into a specific bucket with index i within a segment, no further keys will be inserted into buckets with indexes smaller than i since those buckets are already filled in sequential order. Additionally, buckets with indexes larger than i are currently empty and do not contain any keys. As a result, when computing the utilization of a segment U_s in a sequential dataset, DyTIS considers only the buckets with indexes less than or equal to i. This selective consideration optimizes DyTIS's operations, eliminating the need to analyze and adjust the utilization of empty or higher-indexed buckets. This results in improved efficiency and allows DyTIS to effectively manage and index sequential datasets with enhanced performance and scalability.

Identifying sequential datasets can be more challenging in a multi-threaded version where keys are inserted by more than one thread. In DyTIS, each thread independently detects a sequential dataset as follows. Each thread, similar to the single-threaded version, locally compares the key previously inserted by itself with a new key that it is about to insert to determine if it is a part of a sequential dataset or not. If any thread discovers that the newly inserted key is smaller than the previously inserted key, it notifies all the other threads of the fact that the current dataset is not sequential through a shared flag. Note that a situation where each thread continuously processes sequential keys even though, in reality, the overall dataset is not sequential may be possible, but such a situation will be exceedingly rare in practice. This situation may lead to inaccurate computation of a segment utilization, but it will not affect the overall correctness of the index.

Reducing Memory Access. To reduce memory accesses during various operations in DyTIS, we leverage the unused 16 MSBs in pointers similarly to previous studies [16, 62, 73]. In other words, we piggyback the information about segment s, including the local depth of s, on those bits. This results in a reduction in metadata access overhead [16]. (The experimental results regarding its effect will be discussed in Section 5.3.)

Adjusting Key Range. To optimize the index performance of DyTIS, the range of valid key values of a dataset needs to be considered. When this range is too small compared to the total key space used, DyTIS can adjust keys as follows. Consider that n is the number of bits used for a key, that is, n bits are used for the total key space. However, if only n' LSBs, where $n' \leq n$, are used for a dataset, $n - n'$ MSBs are all zero. Therefore, in this case, DyTIS adjusts keys by left-shifting $n - n'$ bits such that n' MSBs of the adjusted keys contain valid key information. Note that during index operations, an adjusted key is used but the raw key will be stored in the bucket. This adjustment eliminates unnecessary computations and storage overhead, resulting in more efficient and faster operations. In the current implementation of DyTIS, n' can be provided by the user. By incorporating this optimization into the DyTIS index structure, its performance is enhanced, especially in scenarios where the valid key value range is much smaller than the total key space.

5 Experimental Results

5.1 Methodology

For our experimental study, we use a machine with two Intel Core i9-9900K (8-core, 3.6GHz) with 16MB L3 cache and 64GB DDR DRAM, where Ubuntu 18.04 LTS with Linux kernel version 5.4 was installed on the machine, except for Section 5.7, which we elaborate on later. To evaluate DyTIS, we compare its performance with the STX B+-tree[5] [2, 12] (hereafter, simply referred to as B+-tree), ALEX [7, 24], and XIndex [8, 70], which uses a two-level architecture for learned models, where the first level uses a learned RMI while the second level uses linear models and supports concurrent operations. For B+-tree, the fanout is set to 128, which shows the best performance in our setup. ALEX and XIndex require bulk loading of datasets. Thus, 70% of each dataset is bulk loaded for XIndex, whereas 10% or 70% is used for ALEX. Note that in the original ALEX paper [24], the authors use roughly 10% to 50% of the datasets to train the index. A discussion of the effect of bulk loading is given in Section 5.3. In addition, unlike DyTIS and XIndex, the original ALEX and B+-tree do not support in-place updates. For our experiments, we modified ALEX and B+-tree to support in-place updates as suggested within the ALEX code.[6] For DyTIS, in the default setting, the bucket size B_{size} is 2KB and the local depth L_{start} where DyTIS starts remapping and expansion is set to 6. The array size in the first level is set to 2^9 (i.e., $R = 9$), and U_t is set to 0.6. For datasets with

[5] We choose this open-sourced B+-tree implementation, as it was the choice of multiple earlier studies on efficient index structures [24, 55, 70].

[6] https://github.com/microsoft/ALEX/blob/master/src/core/alex.h, lines 1123 and 1124.

Table 1. Datasets Used in Our Experiments

Name	Description	Number of keys	Key range size ($\times 10^{18}$)	Dataset size	Skewness, KDD
MM	South America	356M	5.92	7.0GB	L,M
ML	Africa	903M	15.4	18GB	L,M
RM	Deduplicated data	82M	1.31	1.6GB	H,L
RL	Ratings only	228M	5.18	4.5GB	H,L
TX	Taxi trip in New York	325M	1.08	6.7GB	M,H

large expansion operations (which is determined by DyTIS dynamically), at local depth 8, a limit on the segment size $Limit_{seg}$ is increased to 128 times (from 2 times by default). We will discuss the effect of these parameter settings in Section 5.3. We run the experiments with a single thread, except for those in Sections 5.7 through 5.9, and report the average of three runs.

5.2 Datasets

Table 1 shows the five real-world datasets with different key distribution characteristics used in our experiments where MM, ML, RM, RL, and TX denote the Map-M, Map-L, Review-M, Review-L, and Taxi datasets, respectively, and L, M, and H denote low, medium, and high skewness or KDD, respectively. For each record in the datasets, the size of the keys is configured to 8 bytes as done in prior works [24, 30, 48, 70] and also the size of the values is configured in the same way. In the table, for each dataset, we provide the number of keys (in millions: M), the key range size (i.e., the difference between the minimum and maximum key values), and the dataset size.

For the datasets, we adopt real-world datasets that have been widely used in existing indexing studies [24, 25, 32, 70, 78]. To generate unique integer keys, transformations are applied with selected fields in the real-world datasets. MM and ML are generated by combining the longitudes and latitudes of two different continents, respectively, from OpenStreetMap [34, 63] (similarly to ALEX). RM and RL are generated from Amazon review data [57]. From the original data, we select three fields, item ID, user ID, and review time, and generate unique keys by concatenating them. TX consists of pickup time and drop-off time fields of yellow taxi trips in New York City from 2017 to 2020 in the TLC Trip Record data [17]. Note that unlike ALEX [24], where the keys are randomly shuffled, our datasets are generated from the original datasets without shuffling, and thus the key distribution can dynamically shift over time, as discussed in Section 2.1.

5.3 Results with Real-World Workloads

We evaluate the performance of the four different index structures, DyTIS, ALEX, XIndex, and B+-tree, using seven workloads that roughly correspond to workloads Load, A, B, C, D, E, and F of YCSB [18]. For ALEX, two versions with 10% and 70% bulk loading, ALEX-10 and ALEX-70, are used. Using such workloads for real-world datasets is similar to that done for the ALEX study [24], where for each workload except Load, a batch of the workload is repeated for at least 60 seconds. (In detail, we implement a batch-based approach similar to the method used in ALEX [7]. We set the batch size, which refers to the number of operations per batch, to 10% of a dataset, and for each workload, we run five or more batches for at least 60 seconds.) The Load workload is composed of 100% inserts, the A workload, 50% reads and 50% updates, the B workload, 95% reads and 5% updates, the C workload, 100% reads, and the E workload, 95% scans where the range value is set to 100, and 5% inserts. Note that unlike in the ALEX study [24], we also include workloads F, which

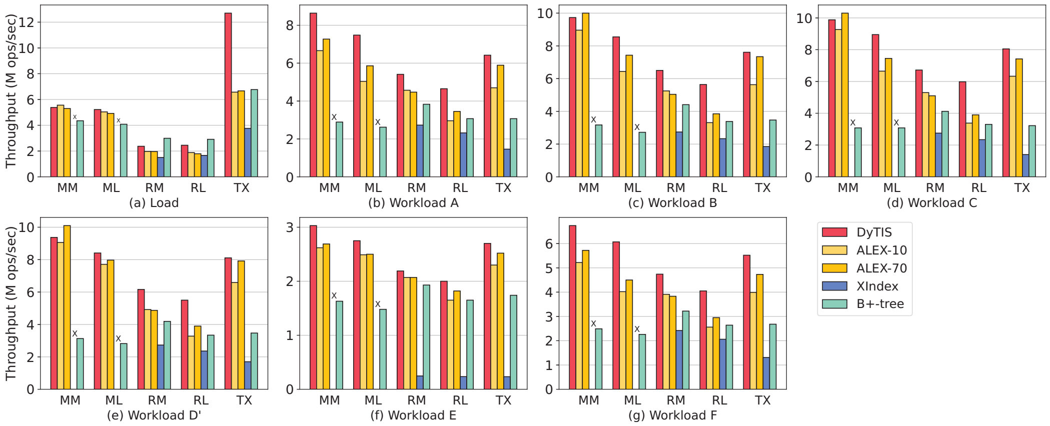


Fig. 8. Throughput of seven real-world workloads over various datasets.

consists of 50% reads and 50% read-modify-write, and D'. D' is the same as the original workload D in that it includes 5% inserts, but it is different in that the 95% reads are selected from the existing keys, and not from the latest keys.[7] For all workloads except Load, the keys are selected from the datasets using Zipfian distribution (with the default Zipfian constant in YCSB, 0.99). Note that we also ran all the experiments with uniform distribution as well, finding the results to be similar with Zipfian distribution.

For workloads A, B, C, and F, the Load workload is first executed to insert all the records in the dataset to the index. In this Load phase, for ALEX, 10%/70% are bulk loaded and 90%/30% are inserted, whereas for XIndex, 70% are bulk loaded and 30% are inserted. Then, we run each workload such that the number of operations performed is more than 50% of the dataset size for at least 60 seconds. For workloads D' and E that include inserting new keys, 80% of the keys are loaded. Then, we start measuring the throughput until all the keys in the dataset are inserted.

Figure 8 shows the throughput of seven workloads of YCSB (i.e., Load, A, B, C, D', E, and F), respectively, for DyTIS, ALEX-10, ALEX-70, XIndex, and B+-tree. From the results of Load, a purely insert workload, shown in Figure 8(a), we observe the following. First, DyTIS tends to provide better throughput than the other indexes for datasets with high KDD such as TX and for datasets with medium KDD and low skewness such as ML, as the local adjustment of its model is effective when the key distribution changes. Second, for datasets with high skewness such as RM and RL, DyTIS performance is lower (24.86% lower on average) than that of B+-tree because of remapping overhead. However, it is still higher than those of the learned indexes, ALEX-10, ALEX-70, and XIndex. Third, DyTIS shows better insertion performance than ALEX for more dynamic datasets of RM, RL, and TX. Note that for ALEX and XIndex, the results do not include bulk-loaded keys. Now, from the results of a purely search workload C shown in Figure 8(d), we observe that the search throughput of DyTIS is the highest, with DyTIS being higher than the other indexes except for MM, where ALEX-70 is 6.8% better than DyTIS.

The results of the preceding insert and search operations exemplify the key differences between DyTIS and ALEX and their performance implications. Recall that DyTIS uses a hash-based structure with segments, each with multiple models, whereas ALEX uses an adaptive RMI based structure with data nodes, each with a single model. Due to these structural differences, for insert operations, with DyTIS, the overhead for maintaining the index structure changes due to splits,

[7] Measuring the performance of repeated batches makes the exact modeling of the D workload complex.

expansions, and remappings being low when KDD is relatively high, such as for ML and TX, as the local model is effectively adjusted. This overhead, however, increases with higher skewness, such as for RM and RL. With ALEX-10, we find that similar structure maintenance overhead is, compared to DyTIS, 50% higher on average, as it needs to perform many more expensive operations to retrain or create models for datasets with high skewness or high KDD. For MM, ML, RM, RL, and TX with ALEX-10, the percentage of those expensive operations over the total maintaining operations is 21%, 30%, 41%, 47%, and 24%, respectively. For TX, even though the size of TX is around 3 times smaller than ML, the absolute number of the maintaining operations and, consequently, that of the expensive operations are around 10 and 8 times higher than those of ML. For search operations, we first note that to query a key, DyTIS always uses a linear model once, but ALEX uses at least two linear models, one at the root node and the other at a data node, with, possibly, more models in internal nodes of the tree. For the datasets, the average number of models used in ALEX-10 is up to 3.33 times (for RL) that in DyTIS. Due to this, we find that the traversal overhead to traverse to a target bucket or data node for a given key with ALEX-10 is 22% higher on average than that with DyTIS. Finally, we find that the performance difference between DyTIS and ALEX for searching or inserting a key within a bucket or data node is not a dominant factor.

In Figure 8, we also observe that for MM, compared to DyTIS, the performance of ALEX-10 is higher for Load, whereas that of ALEX-70 is higher for workloads B, C, and D'. MM has low skewness and is smaller in size relative to ML. In addition, its KDD is medium. Therefore, the built structure with ALEX-10 has minimum depth similar to the Uniform dataset, showing higher insert performance. Note that even though the index structure with ALEX-10 has minimum depth, its search performance is lower than DyTIS as the root node has a large number of children nodes (i.e., data nodes), which increases the traversal overhead. With ALEX-70, the effect of KDD for MM becomes much lower as it bulk loads a larger amount of the dataset compared to ALEX-10. Therefore, ALEX-70 ends up having the smallest number of nodes for the index among all the datasets (which is 7% smaller than that with ALEX-10). Consequently, its traversal overhead becomes small, resulting in good performance as exemplified for workloads B, C, and D'.

From Figure 8, we also observe that aside from Load and MM with ALEX-70 as mentioned earlier, DyTIS performance is always the highest among all the indexes for all datasets, as DyTIS provides good search and update performance. For workload E, we find that performance of B+-tree is lower than those of ALEX and DyTIS mainly due to its small data node size. The average data node size is 4.0 to 295.1 times and 4.7 to 319.2 times smaller than those of ALEX(-10 and -70) and DyTIS, respectively, forcing accesses to a much larger number of data nodes.

Overall, XIndex performs considerably worse than DyTIS or ALEX because it employs a delta index and a temporary delta index to handle key insertion along with a background compaction thread. Such additional structures and background thread overhead for merging data in the index and the delta index take their toll compared to DyTIS and ALEX, which are free of such structures and overhead. In the experiments, XIndex cannot load MM and ML (with 70% bulk loading) due to out-of-memory faults, which occur when available system memory is insufficient, and thus subsequent workloads cannot run.

Next, we discuss the performance of DyTIS when using the datasets in Groups 2 and 4 of Figure 1, which are relatively simple ones. For the datasets in Group 2, which are the shuffled versions of those in Group 1, DyTIS shows the highest throughput among all the indexing techniques for all the YCSB workloads, except for Load with RM and RL, similarly to the original datasets, and for MM. The datasets in Group 4, which are used in the ALEX study [24], are much less dynamic than those in Group 1 such that their key distributions are easy to predict, which is ideal for traditional learned indexes. For the Uniform dataset of 1 billion keys, ALEX-10 shows 18.6% better throughput than DyTIS on average for the YCSB workloads. DyTIS, however, still provides up to 3.5 times (for

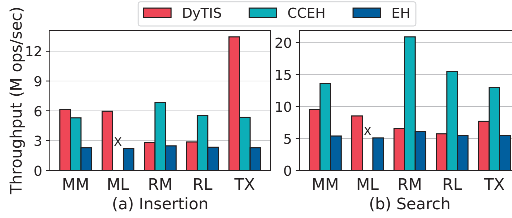


Fig. 9. Comparison of DyTIS with CCEH and EH.

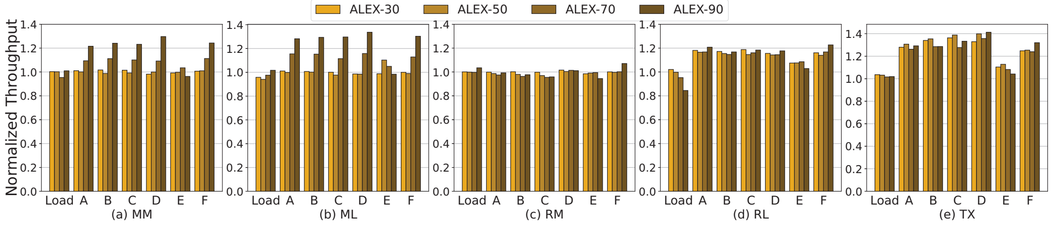


Fig. 10. Throughput of ALEX over various bulk loading percentages normalized to ALEX-10.

workload C) higher throughput compared to B+-tree for all the workloads. For Longlat, which has the highest skewness in Group 4, DyTIS performs better (7.3% to 16.6%) than ALEX-10 for workloads A, E, and F, while performing lower (2.5% to 6.4%) for workloads Load, B, C, and D'.

Figure 9 shows the performance of DyTIS, CCEH, and EH described in Section 3.1. Figure 9 shows that DyTIS provides better insert and search performance than EH for all the datasets. It also shows that for insertion, CCEH and DyTIS give and take depending on the dataset, whereas for search performance, DyTIS is lower than CCEH by an average of 50%. Note that CCEH fails for ML due to out-of-memory, which occurs when available system memory is insufficient for the indexing process. Generally speaking, DyTIS supports scans by replacing a hash function with a remapping function resulting in deteriorated search performance compared to CCEH, but which is still higher than B+-tree, ALEX-10, ALEX-70 (except for MM), and XIndex.

Effect of Bulk Loading on Learned Indexes. We discuss how bulk loading affects the performance of ALEX and XIndex, which influences our first design consideration. As shown in Figure 8, ALEX-70 does not always provide better performance than ALEX-10. In particular, for RM, the performance of ALEX-10 is better than or similar to that of ALEX-70 for the YCSB workloads. In addition, as shown in Figure 8(a), the load performance of ALEX-70 tends to be lower than that of ALEX-10. This is because as nodes are updated with splits and expansions due to insertion, the structure becomes more complex and more keys need to be moved. In the end, we find that the node size and depth of ALEX-70 are 337% larger and 26% deeper, on average, than ALEX-10 after bulk loading. To observe the influence of bulk loading further, we also evaluate ALEX with 30%, 50%, and 90% bulk loading for each dataset. Figure 10 shows the throughput of each workload normalized to that with ALEX-10. Note that for workloads D' and E, 80% of the keys are loaded as in the earlier experiments except for ALEX-90 (which bulk-loads 90% of the dataset), and the number of operations for each of these workloads remains the same over various bulk loading percentages except for Load, and workloads D' and E with ALEX-90. The key finding from the experiments is that no regularity can be found between load size and performance. For example, in the case of RM, as we increase bulk loading from 10% to 70%, performance generally decreases or remains similar, whereas for TX, ALEX-50 performs better than or similarly to ALEX-90, in most cases. In addition, for MM and ML, ALEX-70/90 tend to provide better performance than ALEX with lower bulk loading (i.e.,

Table 2. Average, 99th, and 99.99th Percentile Tail Latencies

Average/99th/99.99th percentile latency (in nanoseconds)										
	Load					YCSB A				
	DyTIS	ALEX-10	ALEX-70	XIndex	B+-tree	DyTIS	ALEX-10	ALEX-70	XIndex	B+-tree
MM	223/624/43054	***201***/582/122640	210/***573***/131120	na/na/na	254/740/***12050***	***185***/***541***/***797***	207/627/954	197/610/920	na/na/na	361/800/2063
ML	233/643/51346	***221***/617/131380	224/***583***/119780	na/na/na	275/802/***2253***	***211***/***586***/***828***	257/712/1093	236/685/1044	na/na/na	401/842/2190
RM	477/1066/46086	542/1240/90494	538/1204/61704	668/1288/18195	***370***/***810***/***2793***	***269***/719/1008	274/743/1097	271/748/1434	376/892/2535	277/***679***/***970***
RL	460/1052/37576	560/1235/86570	578/1178/91276	618/1225/16448	***374***/***852***/***2456***	***313***/***772***/***1473***	396/971/2209	348/912/2178	445/1047/5806	339/773/1994
TX	***97***/***140***/26250	169/338/177400	165/324/172400	281/536/9360	184/307/***1698***	236/***653***/***898***	273/743/1287	***230***/674/950	697/1557/15996	340/770/2000

na, Not available (best values are boldfaced).

ALEX-10/30/50), whereas ALEX-10/30/50 show similar performance with each other. Overall, for our datasets, the performance difference over the different bulk loading percentages is as high as 21%, 31%, 36%, 40%, 42%, 14%, and 32% for the Load, A, B, C, D', E, and F workloads, respectively. Our deeper analysis reveals that bulk loading is critical with ALEX, as once the structure is built with a particular depth during bulk loading, ALEX vigorously deters increasing this depth.

Similar experiments were conducted for XIndex for TX, showing similar impact of bulk-loading percentages on its performance to ALEX. We also find that for RM and RL, insertion failed with less than 70% bulk loading, conjecturing that memory access issues during insertion result in the failures. Thus, 70% bulk loading was chosen for the earlier experiments.

Insertion Breakdown and Tail Latency Analysis. We analyze the insertion execution time breakdown for DyTIS, that is, how much time it spends for each operation. We observe that for RM and RL with high skewness, remapping is leveraged the most. For TX with high KDD, it spends a relatively large portion of time for both remapping and expansion. We also analyze that the overhead of remapping is composed of memory copy overhead (58% on average over the five datasets) and remapping function adjustments (42%), and thus it is proportional to the size of the segment. It is worth noting that considering a segment with size x, the overhead of remapping, which ends up having a size slightly larger than x, is similar to or only a bit larger than that of split, which ends up having a size of $2x$, for the segment. Furthermore, the expansion of x, which results in a segment of $2x$, has the same memory copy overhead as split, having similar overhead to remapping.

Table 2 presents the average, the 99th, and the 99.99th percentile tail latency numbers of workloads Load and A, where best values are boldfaced, for DyTIS, ALEX-10, ALEX-70, XIndex, and B+-tree. In the case of the Load workload, DyTIS is better than ALEX(-10 and -70) for dynamic datasets, RM, RL, and TX. However, B+-tree performs the best for most cases. For the 99.99th percentile tail latency, DyTIS performs worse than B+-tree due to the overhead for remapping large segments. However, we also see that the 99.99th tail latency of ALEX(-10 and -70) whose (re)training operations can be more expensive is 3.3 and 3.1 times larger on average, respectively, than that of DyTIS. For workload A, DyTIS does better than ALEX-10 and ALEX-70 except for the average latency for TX with ALEX-70. Note that although not shown, for the other workloads, DyTIS provides better or similar performance compared to ALEX in most cases.

Memory Usage Analysis. We measure the maximum memory usage of each index structure using "dstat" where the amount of memory used for the benchmark execution and for bulk loading in the case of ALEX and XIndex is included, but it is important to note that the memory needed for bulk loading is released once the bulk-loading process is done. We find that on average, the maximum memory usage of ALEX-10, ALEX-30, ALEX-50, ALEX-70, ALEX-90, and B+-tree is 25.9%, 26.3%, 27.0%, 22.6%, 5.3%, and 27.4% less than that of DyTIS for the YCSB workloads. DyTIS uses more memory because a segment consists of multiple buckets as in CCEH [60], and thus each key must be stored in a particular bucket unlike ALEX and B+-tree. Interestingly, the memory usage of ALEX-90 increases (using only 5.3% less memory than DyTIS), as the sum of the memory needed for bulk loading and for the index structure when the bulk loading is done, which is the point

of the maximum memory usage, becomes much larger. For XIndex, we observe that its memory usage is much higher than the others (e.g., 4.2 times higher than DyTIS).

Note that to reduce the memory overhead in DyTIS, several strategies can be implemented. Allowing linear probing to adjacent buckets, as employed in CCEH [60], can help distribute the memory load more evenly, potentially reducing the overall memory footprint. Additionally, increasing the utilization threshold U_t of the remapping function can enhance memory usage efficiency. However, these strategies involve tradeoffs, as linear probing increases search latency, and a higher utilization threshold causes slowdown of insertions.

Effect of Reducing Memory Access. Recall that in DyTIS, the unused 16 MSBs in pointers are used to piggyback the information about a segment, reducing memory accesses. For our setup, when microbenchmarks of insertion, search, and scan are used for the five datasets, the impact of this optimization varies across datasets and operations. The performance of DyTIS increases by 3.9% on average and up to 10.4% with this feature.

5.4 Parameter Effect

In this section, we investigate the various parameters that affect DyTIS performance, namely, the bucket size B_{size}, the local depth L_{start} at which local remapping and expansion starts, the value of the first-level bit R, the utilization threshold U_t, and the limit on the segment size $Limit_{seg}$. We use throughput as our performance metric, averaged over all the datasets for the YCSB workloads, normalized to that of the default parameter value.

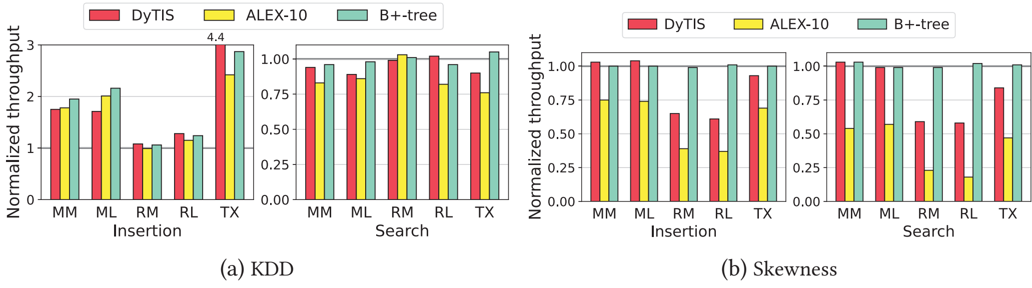
Bucket Size B_{size}. With smaller B_{size}, insertion and search become cheaper as the overhead to shift keys for insert and to find a key within a bucket is reduced. However, more buckets need to be accessed for scan, degrading its performance. Over different B_{size} values, 1KB, 2KB (default), and 4KB, insertion, search, and scan throughput changes, relative to the default, are −16.3% to 0.2%, −10.1% to 12.8%, and −13.2 to 2.8%, respectively. In general, we see a trend that with smaller bucket size, insert and search throughput improves, whereas with larger buckets, scan throughput improves.

Local Depth L_{start}. Larger L_{start} reduces remapping overhead, improving insertion performance, but generates more segments, degrading search and scan performance. However, smaller L_{start} degrades insertion performance due to structure changes based on a small number of inserted keys. Over different L_{start} values, 4, 6 (default), 8, and 10, insertion, search, and scan performance is affected, relative to the default, by −11.0% to 7.3%, −2.8% to 0.3%, and −5.5% to 2.8%, respectively.

First-Level Bit R. The value of the first-level bit R determines the number of EHs in DyTIS. Larger first-level bit R increases the number of EHs having the effect of spreading keys and reducing rebalancing overhead, resulting in improved insertion performance. The performance change for insertion, relative to the default, over different R values, 7, 9 (default), 11, and 13, is −6.7% to 6.2%.

Utilization Threshold U_t. Lower utilization threshold U_t reduces remappings, improving insertion performance. However, this also decreases memory utilization. Over different U_t values, 0.5, 0.55, 0.6 (default), 0.65, and 0.7, insertion performance is affected, relative to the default, by −12.6% to 6.8%.

Limit on the Segment Size $Limit_{seg}$. Recall that DyTIS decides $Limit_{seg}$ dynamically depending on the dataset. Thus, for our analysis, we use different default values, 2 and 128, for datasets with high and low skewness, respectively. In case of RM, RL, and TX with high skewness, DyTIS performs remapping frequently. Thus, using a small limit value is beneficial for insertion performance. Over different $Limit_{seg}$ values 2 (default), 8, 32, and 128, insertion performance for those datasets decreases as the $Limit_{seg}$ increases, with changes, relative to the default, ranging from −69.0% to −14.0%, whereas search and scan performance, relative to the default, changes 7% on average. However, for MM and ML with low skewness, it has only a small effect on insertion performance while search and scan performance increases, such that over different $Limit_{seg}$ values, 2, 8, 32, and 128



(a) KDD (b) Skewness

Fig. 11. Influence of dynamic characteristics.

(default), the throughput changes, relative to the default, for insertion operations range within 2%, whereas those are −12.5% to −3.7% and −8.3% to −3.0% for search and scan operations, respectively.

5.5 Effects of Dynamic Characteristics

In this section, we discuss how our second and third design considerations in Section 2.2 are manifested in supporting dynamic datasets by analyzing the effects of KDD and skewness of datasets on the performance of the indexes. We consider insertion and search, and use the results of Load and workload C of the previous section. To separately analyze the effect of KDD, we compare the performance of the original datasets where the key distribution changes over time with that of the shuffled versions that remove such changes. Figure 11(a) shows the insert and search performance of DyTIS, ALEX-10, and B+-tree for the original datasets, normalized to that for the shuffled versions. From these results, we observe the following. First, in the case of insert, higher KDD has a more positive effect on performance for all the index structures. For RM and RL whose KDD is very small, the performance difference is relatively small compared to other datasets. For TX with high KDD, the performance improvement of DyTIS is as high as 339.76%. Similarly, MM and ML, with relatively high KDD, also show substantial performance differences. This is due to the spatial locality that exists in the original dataset, and the indexes exploit this when inserting. Second, let us now observe search, which is an operation that we call upon after the structure is constructed. Initially, we see that B+-tree is insensitive to KDD, which is logical as a balanced structure is constructed in the end, regardless of KDD. In contrast, we observe that DyTIS and ALEX-10 are affected somewhat, with ALEX-10 being more strongly so. In addition, the performance degradation generally tends to be proportional to the KDD values of the datasets. The reason for this is that the structures were constructed in a more imbalanced manner due to the spatial locality (i.e., KDD) during insertion. However, ALEX-10 is more strongly affected as its structure becomes more complex with a larger number of linear models when the models in the upper hierarchy are inaccurate for a dataset with high KDD such as TX, unlike DyTIS which is based on remapping. One peculiar point is that of RL, which has relatively low KDD and yet strongly influences ALEX-10. We find that although KDD of RL is not high, as RL also has very high skewness, this results in divergence affecting the built structure more strongly, increasing the number of linear models by 83,209 (20% increase).

We now analyze the effect of skewness, and to do so, we compare the performance for the shuffled versions of the original datasets with that for the Uniform datasets of the same size, which have no skewness. Figure 11(b) shows the performance for shuffled datasets normalized to that for the corresponding Uniform datasets. We observe the following. First, for B+-tree, there is no performance effect caused by skewness of a dataset as the normalized performance for all the datasets is 1. Second, DyTIS is robust to low skewness, as we can see from the performance results for MM and ML. However, for datasets with moderate and high skewness, the performance of

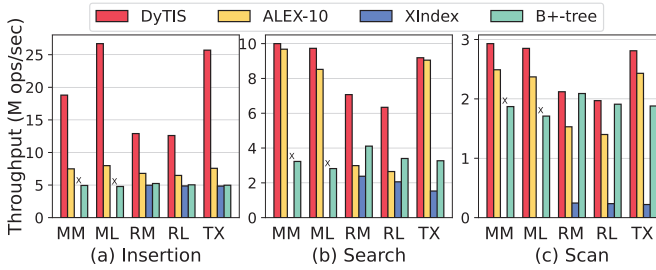


Fig. 12. Throughput over various sequential datasets.

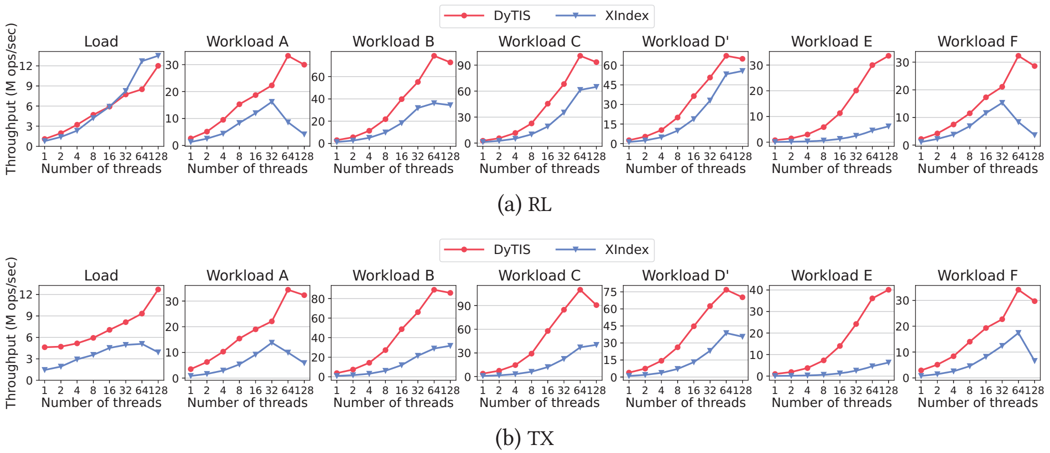
DyTIS degrades as DyTIS needs more models to accommodate such skewness. Third, we find that ALEX-10 is sensitive to any degree of skewness. This is because, unlike DyTIS, ALEX-10 creates a node for every model resulting in a large number of nodes compared to the Uniform dataset. For RM and RL with high skewness, its performance degradation becomes severe as the number of nodes increases by 1,341 times on average, whereas for DyTIS, there is only a 17 times increase.

5.6 Results with Sequential Datasets

As earlier studies have considered sequential datasets where keys are inserted in increasing order [32, 45], we also consider them here. DyTIS detects sequential datasets and handles them in an optimized way as discussed in Section 4. Similarly, ALEX has optimized logic to handle datasets with an extreme distribution shift pattern such as sequential datasets. Figure 12 shows the performance of DyTIS, ALEX-10, XIndex, and B+-tree over sequential versions of the original datasets (where all the datasets are inserted in sorted order). The results clearly show that DyTIS consistently outperforms the other indexing methods for all operations. We have extended our evaluation by conducting YCSB workloads on these datasets and observed that DyTIS shows better or comparable performance compared to the other indexes for all the workloads, similar to Figure 12.

Next, we further analyze the effect of the *very* high KDD values of the sequential datasets on the index performance by comparing their performance with that of the original datasets. Recall that thanks to the spatial locality that datasets with high KDD have, high KDD has a more positive effect on insert performance as discussed in the previous section, and the KDD values of all the sequential datasets are as high as 12.5 as shown in Figure 1. Therefore, the insert performance of the sequential datasets is higher than that of the corresponding original datasets, except TX with B+-tree, such that DyTIS, ALEX-10, and XIndex provide 329%, 120%, and 153% higher throughput on average, respectively, for the sequential datasets. DyTIS shows the highest improvement over the original datasets as DyTIS has no need to shift keys for insertion within a bucket for sequential datasets, and it provides insert performance higher than B+-tree even for RM and RL. For TX, the original dataset has high KDD (i.e., higher than 10), and thus the performance gain with the sequential dataset is lower than that with other datasets. In the case of B+-tree, its insert performance is 13.2% lower than that of the original dataset, because the overhead of node splits becomes higher as keys with larger values are continuously inserted to the B+-tree. Note that the performance gain of B+-tree with the sequential datasets is lower than the other indexes for all the datasets.

For search and scan, DyTIS shows better or similar performance, and B+-tree shows almost the same performance, compared to the original datasets. However, ALEX provides up to 62.6% and 18.2% higher search and scan performance, respectively, for MM, ML, and TX whose skewness is not high, because the number of models decreases up to 84.9% when the index structure is built with the optimization for the sequential datasets. Yet, the search and scan performance of ALEX is degraded by up to 41.6% and 22.5%, respectively, for RM and RL whose skewness is high, because



(a) RL

(b) TX

Fig. 13. Throughput with different numbers of threads.

the depth of the structure increases, despite a decrease in the number of models. In the case of XIndex, it provides 12.1% lower search performance for RM and RL, while demonstrating similar scan performance for RM and RL, and similar search and scan performance for TX, compared to the original datasets.

5.7 Concurrency

All results presented thus far were with single-thread executions, as ALEX and B+-tree are single-thread solutions. In this section, we evaluate the performance of DyTIS with XIndex over various numbers of threads. In this section, different from all the other experiments, we use a machine with two Intel Xeon Gold 6338 (32-core, 2.00GHz) with 48MB L3 cache and 128GB DDR DRAM where Ubuntu 20.04.4 LTS with Linux kernel version 5.4 was installed to evaluate the performance with multiple threads using a larger number of threads.[8] During our experiments, hyperthreading is turned on, providing 128 logical cores in total, and the Interleave NUMA memory allocation policy is used while the automatic NUMA balancing feature is disabled. Figure 13 shows the throughput of the seven workloads of YCSB for DyTIS and XIndex on the RL and TX datasets, comparing the performance of DyTIS with that of XIndex over different thread counts up to 128 threads. For the Load workload, we assign requests for operations to threads in a round-robin fashion to analyze the effect of key insertion order.

Across all workloads, DyTIS consistently outperforms XIndex up to 128 threads, except for Load with 32, 64, and 128 threads in the RL dataset. We analyze the impact of multi-threading on performance using the Intel Vtune Profiler [38], particularly focusing on the variations across different thread counts. Generally, while increasing the number of threads can lead to performance improvements, these gains often decrease or may even reverse with 128 threads due to increased resource contention and cache pollution.

For workloads Load and E, DyTIS shows better scalability and improved performance with 128 threads, primarily due to more efficient utilization of memory bandwidth and CPU resources compared to other workloads. Specifically, for Workload E, DyTIS benefits from the lowest memory bandwidth usage and highest CPU resource utilization among all workloads, which minimizes

[8]For experiments with single thread executions, we use a machine with specifications similar to those used in the ALEX study [24] to evaluate the performance with a single thread on a similar machine configuration for the indexes.

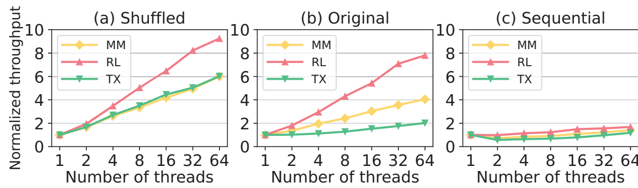


Fig. 14. Insertion throughput of DyTIS over shuffled, original, and sequential datasets normalized to a single thread for the corresponding dataset.

memory contention and consequently shows better performance at higher thread counts. Similarly, for Load, DyTIS achieves the lowest LLC cache miss with 128 threads, indicating effective cache performance, which contrasts with the higher cache and memory access issues observed in other workloads. Moreover, for TX with high KDD, DyTIS benefits from high spatial locality, which reduces memory bandwidth usage.

For Load with RL, XIndex outperforms DyTIS at higher thread counts due to its efficient support for concurrent insertions through features like the delta buffer and Two-Phase Compaction. However, in the TX dataset, DyTIS consistently outperforms XIndex. This is because the high spatial locality in TX causes key clustering within specific group nodes in XIndex, reducing the effectiveness for concurrent insertion, which diminishes its performance advantage.

In the case of RM whose results are not shown, its performance trend is similar to RL such that for all the workloads except Load, DyTIS performs better than XIndex in all cases with different numbers of threads. In summary, the performance trends show that DyTIS outperforms XIndex across all datasets and thread counts for real-world workloads, exhibiting 3.3 times higher throughput, on average, when using 32, 64, and 128 threads, over all the workloads except Load.

Next, we analyze the effect of KDD of datasets on the performance of DyTIS with multiple threads. Figure 14 shows the insert throughput of DyTIS with different versions for MM, RL, and TX datasets over various numbers of threads, normalized to that using a single thread for their corresponding datasets. The results show that for each dataset, as the KDD values increase in the order of shuffled, original, and sequential versions, the scalability tends to decrease. Regardless of the datasets, the shuffled versions have very low KDD, showing high insertion scalability. Especially with RL, the average segment size is eight times smaller than those with the other datasets, showing the highest scalability. However, in the case of the sequential versions with very high KDD, the benefit of concurrency is almost none as a range of keys accessed around the same time by multiple threads becomes very small, causing lock contentions. For the original datasets, KDD increases in the order of RL, MM, and TX, whereas the insertion scalability decreases in the order of RL, MM, and TX. In the case of RL, its KDD is similar to that of the shuffled version, showing high concurrency, whereas TX has high KDD similar to its sequential version, showing concurrency a bit higher than that of the sequential version. Note that although not shown, XIndex shows a similar effect of KDD on performance with multiple threads. Note that DyTIS currently does not implement NUMA-aware techniques but can be optimized for NUMA architectures. For example, a directory can be replicated or partitioned across NUMA nodes to minimize remote memory access. In addition, by organizing segments and directories within an EH to align with a NUMA node, DyTIS can further ensure that operations remain localized to the NUMA node where the data resides, potentially improving performance and scalability in multiple NUMA node environments.

5.8 Performance with Deletion

To analyze the performance of DyTIS for deletion operations, we set up workloads with different read-write ratios, where read operations are all search operations, whereas write operations are

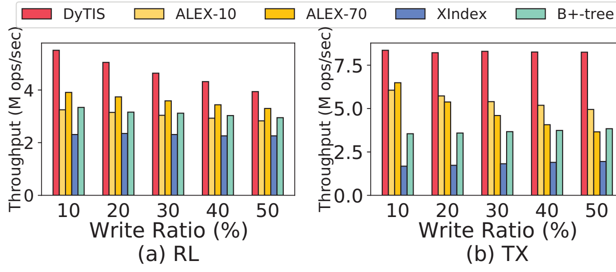
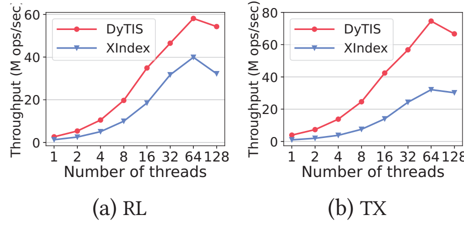


Fig. 15. Throughput with varying write ratios, including deletions.



(a) RL (b) TX

Fig. 16. Throughput with different numbers of threads with 10% writes, including deletions.

composed of insertions, updates, and deletions with a fixed ratio of 1:1:2 to maintain a stable database size. These experiments are similar to those in the XIndex study [70]. For these experiments with real-world datasets, we follow a similar approach to workloads D' and E in Section 5.3, which include inserting new keys. Initially, 80% of the keys are loaded, and then we measure the throughput until all the keys in the dataset are fully inserted. For deletion operations, keys that will be deleted are selected in the same order as inserted keys during Load.

Varying Write Ratios. Figure 15 demonstrates throughput of the four index structures on the RL and TX datasets with a single thread over various write ratios from 10% to 50%. DyTIS consistently outperforms the other indexing methods across all write ratios in both datasets. For RL, as the write ratio increases, throughput tends to decrease across all the methods as higher write ratios typically introduce more overhead than search operations, thereby reducing overall throughput. However, for TX, DyTIS maintains almost consistent performance regardless of the write ratio. This is because the high KDD of TX allows DyTIS to exploit spatial locality during writes, achieving performance comparable to search operations.

Scalability. Figure 16 shows the throughput with different numbers of threads and 10% writes for the RL and TX datasets. Overall, DyTIS demonstrates better scalability and higher throughput than XIndex across varying thread counts in both datasets.

5.9 Comparison with Recent Developments

In this section, we discuss the performance of recently proposed learned indexes, LIPP [76] and FINEdex [51]. LIPP attempts to reduce the exponential search cost in the leaf node as well as to eliminate unbounded last-mile searches in ALEX. However, in our setup, LIPP could not build an index for four of the five datasets due to out-of-memory issues, where memory demand exceeds system capacity, or type conversion errors. While we were able to insert keys for the RM dataset, we observed a significant number of key losses upon search.

FINEdex presents a learned index structure designed to improve concurrency and scalability. FINEdex is optimized to provide fine-grained scalable indexing for high concurrency workloads

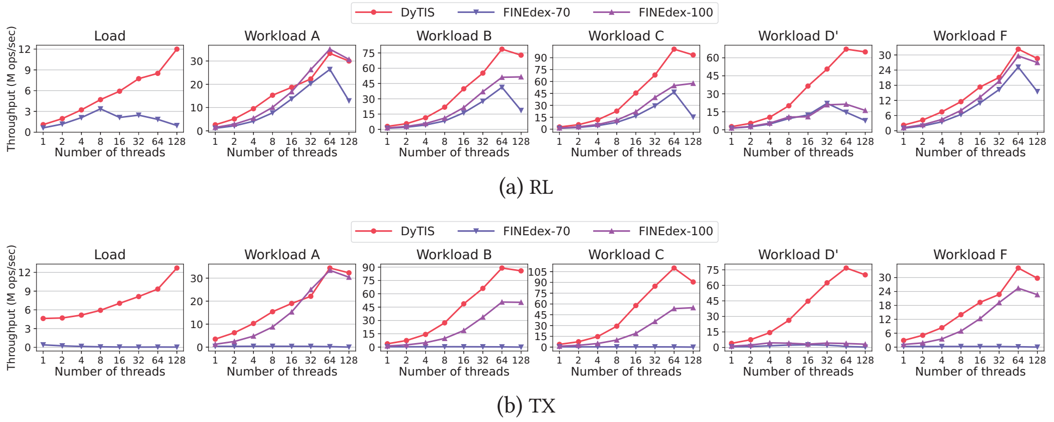


Fig. 17. Comparison of DyTIS with FINEdex over different numbers of threads.

by constructing independent models with a flattened data structure. For FINEdex, we experimentally found the maximum threshold of the model error parameter, 2^{13}, that provides the best performance for all datasets. We ran a microbenchmark composed of load and search for the datasets by doubling the value from 2^5 (which is a default value used in the work of Li et al. [51]) to 2^{17}. We use the found parameter for the experiments and report the results with it in the following. As FINEdex is designed to achieve high scalability in multi-threaded environments, we center our comparison on multi-thread performance to highlight the scalability and concurrency capabilities of DyTIS against FINEdex.

Figure 17 shows the throughput comparison between DyTIS and FINEdex under six workloads of YCSB (i.e., Load, A, B, C, D', and F) with various numbers of threads. Although FINEdex supports range queries, we excluded Workload E, which includes scans, from the results because we observed that the current scan results of FINEdex in our setup contain duplicate keys. For FINEdex, we use two versions with 70% and 100% bulk loading, FINEdex-70 and FINEdex-100.[9]

In Figure 17, DyTIS outperforms both FINEdex-70 and FINEdex-100 in most workloads and thread counts for both RL and TX datasets. For TX, which has a high KDD, FINEdex-100 shows a notable performance improvement for workloads A, B, C, D', and F, compared to FINEdex-70, benefiting from complete knowledge about a dataset during index creation. In contrast, FINEdex-70 provides low performance as the data distribution of 30% of a dataset, inserted during Load, differs from 70% of the dataset, initially bulk loaded for TX, and FINEdex has difficulty handling the dynamic key distribution shift.

6 Related Work

This work extends our previous study on DyTIS [81] by considering a wider range of dynamic datasets in real-world scenarios such that sequential datasets with sorted keys are included, providing a detailed study of the performance with multiple threads, and presenting comprehensive analyses of the effects of the dynamic characteristics of datasets on the performance of indexes.

There are efforts to make an efficient updatable learned index [24, 30, 32, 70, 77]. The PGM index has the structure of a linear piecewise regression model to predict the CDF of each segment where the data is partitioned [30]. The FITing-Tree also leverages piecewise linear functions [32]. NFL

[9]For workload D', which includes insertions during execution, with FINEdex-100, 80% of the keys are bulk loaded initially, and the remaining 20% of the keys are inserted dynamically during the workload execution.

employs a two-stage normalizing flow based approach, first transforming key distributions to a near-uniform state by normalizing flow and then building the after-flow learned index based on the transformed keys [77]. NFL requires previous knowledge of the key distribution for a dataset before building an index. APEX proposes a learned index that is optimized for persistent memory [52]. While APEX targets the DRAM-NVM hybrid architecture, PLIN [83] addresses the challenges of integrating learned indexes into NVM-only systems, including the tradeoff between the benefits of having small nodes for crash consistency and having larger nodes for learned indexes.

The RS index [46] that leverages a linear spine has been proposed as another way of approximating the CDF of the dataset. Flood [61] and Tsunami [25] were proposed as multi-dimensional learned indexes to address the limitation of existing learned indexes that consider one-dimensional data. However, Flood and Tsunami aim to optimize multi-dimensional read-only workloads and do not support insertion operations and concurrency. While most learned index designs consider only integer keys, there have been efforts to support variable-length string keys [67, 74]. Kipf et al. [45] used real-world datasets for benchmarks for learned indexes but only focused on search performance over sorted datasets. Simultaneously to our work, skewness of a dataset is similarly analyzed for learned indexes, where skewness is called *hardness* [75].

SOSD [56] first introduces a benchmarking framework with immutable learned indexes over sorted real-world datasets, demonstrating that learned models often outperform traditional implementations. Andersen and Tözün [10] provide a micro-architectural comparison of ALEX [24] to traditional tree-based structures like ART and B+Tree. Sun et al. [69] provide a detailed review of existing learned indexes and offer insights for selecting suitable learned indexes in practical scenarios. Ge et al. [33] provide a comprehensive evaluation of learned indexes and traditional indexes in an NVM-oriented key-value store and show that learned indexes excel in performance and space efficiency for scan operations, with some tradeoffs in recovery times and sensitivity to data distribution. Bourbon [21] and SageDB [47] leverage learning of data patterns for database system components. Bourbon [21] replaces block indexes in LSM (Log-structured Merge Tree) with learned indexes, whereas SageDB [47] employs index structures and sorting algorithms based on learned models.

Hybrid indexing structures that use both the hash and B+-tree indexes have been investigated for large-scale memory-based data storage systems [43] and for DRAM-NVM memory systems [79]. HiKV proposes to use a hash index persisted to persistent memory, for single-key operations such as insert and search, and it also uses a B+-tree index, which resides in DRAM, to support scan operations [79]. Wormhole is an in-memory indexing scheme that makes use of the hash, B+-tree, and trie, focusing on string key optimizations [78].[10]

FILM [53] introduces a fully learned index for larger-than-memory databases, where data is distributed across diverse storage devices. FILM employs simple approximation models within a learned tree structure to efficiently index data distributed across diverse storage devices and utilizes an adaptive LRU (Least Recently Used) scheme to disk, typically using the LRU strategy for identifying and managing cold data.

For concurrency, Masstree leverages fine-grained locking for updates and optimistic concurrency control [15, 58] for lookups [54]. Clevel hashing [16] supports high scalability by eliminating expensive lock-based concurrency control.

Several works address the challenges of remote memory accesses and scalability in multiple NUMA node environments [20, 44, 72]. DiffLex is a NUMA-aware learned index designed for space-performance tradeoffs and scalability in multiple NUMA node machines [20]. It manages

[10]As it targets string keys, its performance is not high for our datasets that are composed of integer keys, as similarly discussed by Marcus et al. [55].

keys based on access frequency, storing newly added keys in sparse deltas and frequently accessed ones in a sparse hot cache. By partitioning sparse deltas and replicating the hot cache across NUMA nodes, DiffLex mitigates remote memory access overhead. ListDB incorporates a Braided SkipList, a NUMA-aware data structure to reduce the NUMA effects of non-volatile main memory, thus minimizing performance penalties associated with remote memory accesses [44]. Nap converts existing persistent memory indexes into NUMA-aware counterparts by introducing a NUMA-aware layer that maintains per-node partial views in persistent memory for writes and a global view in DRAM for reads, effectively eliminating remote persistent memory accesses to hot items and significantly reducing latency and overhead [72].

7 Conclusion

In this article, we presented an index structure targeting dynamic datasets called *DyTIS*, which is simultaneously efficient for all search, insert, and scan operations. We also defined what dynamic datasets are and presented a means to quantify their characteristics. Our experimental results with real-world datasets demonstrated that DyTIS provides performance that is superior to ALEX, the state-of-the-art learned index. We also demonstrated that DyTIS can provide robust insert, search, and scan performance over diverse dynamic datasets for most cases.

Acknowledgments

We would like to thank the anonymous reviewers for their invaluable comments.

References

[1] Memcached. 2004. Memcached Home Page. Retrieved December 20, 2024 from http://memcached.org/
[2] GitHub. 2007. STX B+-tree. Retrieved December 20, 2024 from https://github.com/bingmann/stx-btree
[3] Redis. 2009. Redis Home Page. Retrieved December 20, 2024 from https://redis.io/docs/management/scaling/
[4] 2011. Kyoto Cabinet: A Straightforward Implementation of DBM. Retrieved December 20, 2024 from https://dbmx.net/kyotocabinet/
[5] Fallabs Tech. 2014. A Fast and Lightweight Key-Value Store Library by Google. Retrieved December 20, 2024 from http://code.google.com/p/leveldb
[6] H-Store. 2016. H-Store Home Page. Retrieved December 20, 2024 from https://hstore.cs.brown.edu/
[7] GitHub. 2020. ALEX. Retrieved December 20, 2024 from https://github.com/microsoft/ALEX/tree/57efb5
[8] XIndex. 2020. XIndex: Project Information. Retrieved December 20, 2024 from https://ipads.se.sjtu.edu.cn:1312/opensource/xindex
[9] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. 2015. Time-series clustering—A decade review. *Information Systems* 53 (2015), 16–38.
[10] Mikkel Møller Andersen and Pinar Tözün. 2022. Micro-architectural analysis of a learned index. In *Proceedings of the 5th International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '22)*. Article 5, 12 pages.
[11] D. F. Andrews, A. M. Herzberg, D. F. Andrews, and A. M. Herzberg. 1985. Monthly mean sunspot numbers. In *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. Springer Series in Statistics. Springer, 67–74.
[12] Timo Bingmann. 2007. STX B+ Tree C++ Template Classes. Retrieved December 20, 2024 from https://panthema.net/2007/stx-btree/
[13] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST '20)*. 209–223.
[14] Josiah Carlson. 2013. *Redis in Action*. Simon & Schuster.
[15] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. 181–190.
[16] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. 799–812.

[17] NYC Taxi & Limousine Commission. 2020. TLC Trip Record Data. Retrieved December 20, 2024 from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.
[19] Toshiba Memory Corporation. 2019. Toshiba Memory Corporation Introduces XL-FLASH™ Storage Class Memory Solution. Retrieved December 20, 2024 from https://www.kioxia.com/en-jp/about/news/2019/20190806-1.html
[20] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. 2023. DiffLex: A high-performance, memory-efficient and NUMA-aware learned index using differentiated management. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23)*. 62–71.
[21] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 155–171.
[22] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.
[23] Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science* 380 (June 2007), 47–68.
[24] 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 (SIGMOD '20)*. 969–984.
[25] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86.
[26] Carla Schlatter Ellis. 1983. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '83)*. 106–116.
[27] Philippe Esling and Carlos Agon. 2012. Time-series data mining. *ACM Computing Surveys* 45, 1 (2012), Article 12, 34 pages.
[28] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main memory database systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130.
[29] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems* 4, 3 (1979), 315–344.
[30] 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.
[31] Brad Fitzpatrick. 2004. Distributed caching with Memcached. *Linux Journal* 2004 (2004), 5.
[32] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A data-aware index structure. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*. 1189–1206.
[33] Jiake Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting learned index into pieces: An in-depth inquiry into updatable learned indexes. In *Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE '23)*. 315–327.
[34] Geofabrik. 2019. OpenStreetMap Data Extracts. Retrieved December 20, 2024 from https://download.geofabrik.de/
[35] Jiawei Han, Jian Pei, and Hanghang Tong. 2022. *Data Mining: Concepts and Techniques* (4th ed.). The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann.
[36] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*. 522–529.
[37] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*. 187–200.
[38] Intel Corporation. 2024. *Intel VTune Profiler*. Retrieved December 20, 2024 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[39] L. R. Johnson. 1961. An indirect chaining method for addressing on secondary keys. *Communications of the ACM* 4, 5 (1961), 218–222.
[40] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22)*. 45–51.
[41] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*. 191–205.
[42] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-Store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[43] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. 2016. SLIK: Scalable low-latency indexes for a key-value store. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*. 57–70.

[44] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of write-ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *Proceedings of 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. 161–177.

[45] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. In *Proceedings of the NeurIPS Workshop on Machine Learning for Systems*.

[46] 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 3rd International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '20)*. 1–5.

[47] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A learned database system. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*.

[48] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. 489–504.

[49] Solomon Kullback. 1997. *Information Theory and Statistics*. Courier Corporation.

[50] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE '13)*. 26–37.

[51] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[52] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A high-performance learned index on persistent memory. *Proceedings of the VLDB Endowment* 15, 3 (2021), 597–610.

[53] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: A fully learned index for larger-than-memory databases. *Proceedings of the VLDB Endowment* 16, 3 (2022), 561–573.

[54] 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 (EuroSys '12)*. 183–196.

[55] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14, 1 (2020), 1–13.

[56] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14, 1 (2020), 1–13.

[57] Julian McAuley and Alex Yang. 2016. Addressing complex and subjective product-related queries with customer reviews. In *Proceedings of the 25th International Conference on World Wide Web*. 625–635.

[58] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-copy update. In *AUUG Conference Proceedings*. AUUG, 175.

[59] Robert Morris. 1968. Scatter storage techniques. *Communications of the ACM* 11, 1 (1968), 38–44.

[60] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*. 31–44.

[61] 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 (SIGMOD '20)*. 985–1000.

[62] N. Nguyen and P. Tsigas. 2014. Lock-free cuckoo hashing. In *Proceedings of the 2014 IEEE 34th the International Conference on Distributed Computing Systems (ICDCS '14)*. 627–636.

[63] OpenStreetMap on AWS. 2021. Amazon AWS. Retrieved December 20, 2024 from https://registry.opendata.aws/osm/

[64] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[65] W. W. Peterson. 1957. Addressing for random-access storage. *IBM Journal of Research and Development* 1, 2 (1957), 130–146.

[66] Samsung. 2017. Ultra-Low Latency with Samsung Z-NAND SSD. Retrieved December 20, 2024 from https://download.semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf

[67] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the last mile: Efficient learned string indexing. In *Proceedings of 3rd International Workshop on Applied AI for Database Systems and Applications (AIDB '21)*. 1–5.

[68] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB main memory DBMS. *IEEE Database Engineering Bulletin* 2013 (2013), 21–27.

[69] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.

[70] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. 308–320.

[71] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC '11)*. 11.

[72] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A black-box approach to NUMA-Aware persistent memory indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. 93–111.

[73] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE '18)*. 461–472.

[74] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: A scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*. 17–24.

[75] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017.

[76] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.

[77] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust learned index via distribution transformation. *arXiv preprint arXiv:2205.11807* (2022).

[78] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys '19)*. 1–16.

[79] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 349–362.

[80] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *VLDB Journal* 23 (2014), 915–937.

[81] Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. 2023. DyTIS: A dynamic dataset targeted index structure simultaneously efficient for search, insert, and scan. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys '23)*. 800–816.

[82] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting memory-level parallelism for efficient DRAM indexing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 147–162.

[83] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. 2022. PLIN: A persistent learned index for non-volatile memory with high performance and instant recovery. *Proceedings of the VLDB Endowment* 16, 2 (2022), 243–255.

Received 25 April 2024; revised 4 September 2024; accepted 22 November 2024