Adaptive Hybrid Indexes

Adaptive Hybrid Indexes

Christoph Anneser
Technical University of Munich
anneser@in.tum.de

Andreas Kipf
Massachusetts Institute of Technology
kipf@mit.edu

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

While index structures are crucial components in high-performance query processing systems, they occupy a large fraction of the available memory. Recently-proposed compact indexes reduce this space overhead and thus speed up queries by allowing the database to keep larger working sets in memory. These compact indexes, however, are slower than performance-optimized in-memory indexes because they adopt encodings that trade performance for memory efficiency. Applying different encodings within a single index might allow optimizing both dimensions at the same time – however, it is not clear which encodings should be applied to which index parts at *build-time*.

To take advantage of multiple encodings in *one* index structure, we present a new framework forming the basis of *workload-adaptive hybrid indexes* which moves encoding decisions to *run-time* instead. By sampling incoming queries adaptively, it tracks accesses to index parts and keeps fine-grained statistics which are used for space- and performance-optimized encoding migrations. We evaluated our framework using B+-trees and tries, and examine the adaptation process and space/performance trade-off for real-world and synthetic workloads. For skewed workloads, our framework can reduce the space by up to 82% while retaining more than 90% of the original performance.

CCS CONCEPTS

• **Information systems** → **Data access methods**; **Data layout**.

KEYWORDS

Space-efficient Index; Adaptive Index; Hybrid Index

ACM Reference Format:
Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526121

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526121

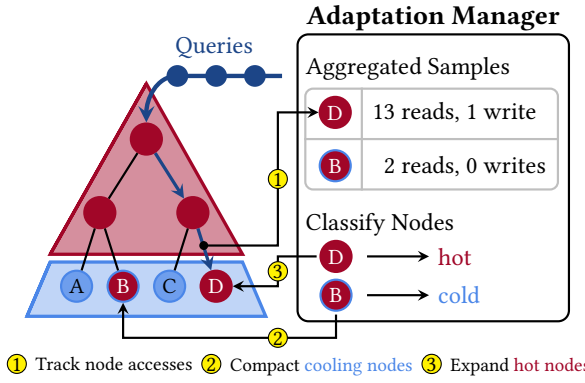

Figure 1: Our sampling-based workload adaptation supports hybrid index structures in choosing the most suitable encoding for each part based on fine-grained access statistics at run-time. It supports user-defined settings such as an upper memory budget and it keeps sampling-related overhead limited by following an adaptive cost-optimized approach.

1 INTRODUCTION

Back in 2006, Jim Gray stated that memory is the new disk and disk is the new tape [5]. This also applies to modern database systems that store the entire data in random access memory (RAM) to allow real-time analyses for trading companies and financial services, for example. They need to process large datasets efficiently to react to new developments and updates within a few milliseconds.

While the DRAM-prices have been stable during the last six to seven years, the data collected by sensors, smartphones, social media platforms, IoT-devices, and digital market-places *increases at a high rate* resulting in data overflows [54], and storing all data in memory becomes infeasible in many cases. However, as in-memory database systems become more and more popular for performance-critical businesses, AWS offers RAM instances that are optimized for in-memory database systems [1]. These instances are equipped with in-memory capacities of up to 24 TB, but the hourly cost of such an instance is more than $120.

To achieve high-performance query-processing for real-time analyses, index structures such as B-trees, tries, and hash tables are widely used by DBMSs. Because there might be multiple indexes per table, especially in OLTP DBMSs, the storage overhead for indexes can be significant. In many cases, more than half of the available memory of a DBMS can be attributed to index structures [54].

Over the last decades, multiple approaches have been developed to represent traditional data structures using more compact encodings [39]. For example, succinct representations avoid storing unnecessary pointers in a data structure by calculating the nodes' position offsets directly [55]. While succinct indexes require significantly less memory compared to performance-optimized state-of-the-art indexes, most of them are slower in point lookups and scans, and they do not support updates efficiently.

To overcome the disadvantages of memory-efficient but static indexes, Zhang et al. [53] proposed *hybrid index* which is a dual-stage architecture combining a regular dynamic index and the memory-efficient read-only one into a single logical index. The dynamic component in hybrid index absorbs all updates and periodically merges all the delta into the more compressed static component. This approach, however, imposes overhead in the expensive merge process because different node encodings are separated into two stand-alone data structures.

"The RUM conjecture" states that we cannot have all three of read, update, and space optimized for a data structure [11]. For example, succinct data structures achieve close to theoretically optimal space, but they sacrifice read performance and updatability. Most index structures used in today's DBMSs are designed for fast reads and updates, and therefore, often at the expense of the memory overhead.

However, we found that the RUM conjecture could have less effect when the workloads are skewed. Unlike standard benchmarks such as TPC-H where the data is uniformly distributed, we observe heavy skews in real-world workloads. The skew appears in multiple dimensions such as in query patterns, keys, and access-space [14].

We, therefore, propose to leverage the skewed workload patterns to determine node layouts at a fine-granularity based on their access frequencies sampled *adaptively at run-time* so that we can reduce the memory overhead of an index while sacrificing minimum performance (cf. Figure 1). More precisely, given an unbounded stream of index queries where the keys follow an unknown distribution, our approach adjusts the layout for each node adaptively so that "hot" nodes are encoded using performance-optimized formats while "cold" nodes are highly compressed.

The framework we proposed in this paper is divided into two phases: during the first phase, we sample and aggregate accesses to different parts in an index (1). In the second phase, we run a heuristic-based classification to identify hot and cold parts. Based on the access statistics and the most recent classifications, we compact cold parts (2) and expand hot parts (3) adaptively using different encodings to achieve a better performance-space trade-off. Furthermore, our framework separates all index-related code from the sampling and classification logic so that it can be easily integrated into existing indexes and systems.

The evaluation in Section 5 shows that our framework can successfully identify the hot and cold parts of an index at a fine granularity and then adjust their encodings adaptively. For skewed workloads, our workload-adaptive hybrid indexes reduce the memory overhead by up to 82% while retaining more than 90% of the performance compared to the original state-of-the-art indexes.

We make the following contributions:

1. A novel framework that helps indexes choose different encodings adaptively based on our lightweight workload sampling to make better performance-memory trade-offs.
2. An alternative offline training for hybrid indexes based on historic or predicted workloads.
3. Applied the framework to two widely-used index structures: B+- and prefix trees.
4. An in-depth evaluation using both real-world and synthetic workloads.

In the following, we first present an overview of sampling-based classification approaches in Section 2. These preliminary approaches are internally used by our approach, which is introduced in Section 3. We provide detailed insights into our approach at an algorithmic level and experimentally evaluate the used parameters. In Section 4, we integrate our approach into a B+-tree and a prefix tree. For both indexes, we present an in-depth evaluation in Section 5. We provide an overview of the related work in Section 6. Ultimately, we draw conclusions and outline possible future work in Section 7. Further experiments based on other datasets and workloads can be found in the online appendix, which is available at https://www.hybrid-index.online.

2 PRELIMINARIES

Many existing works leverage skew in the context of database systems (e.g., [8, 29, 33, 42, 45]). The main idea is to keep frequently accessed data in DRAM to improve overall system performance [29]. For example, Levandoski et al. proposed different *offline* algorithms to efficiently identify hot tuples in Microsoft's in-memory database Hekaton so that the cold ones can be swapped out to disk [33]. To speed up classification, they uniformly sample 10% of all record accesses and accept a *memory hit rate loss* of 2.5% compared to evaluating all record accesses. Depending on the context such as the available memory and the working set (data which is actively used), they *rephrase* the hot-cold-classification as a top-k frequent item detection problem: the k records with the highest estimated frequency are classified as *hot* and are thus kept in memory.

Existing optimizations to the top-k algorithms [15, 16, 37, 38] often assume predefined sizes for the data samples. Therefore, it requires us to determine an appropriate sample size first before we can identify frequently accessed items at *run-time*. While a smaller sample size can lead to higher classification errors, larger sample size brings extra overhead in collecting and analyzing the samples.

To keep classification errors limited, we make use of error-bounded top-k approximations which we formally define as follows. Let $\mathcal{I}$ be a set of items and let $\mathcal{D}$ be a multiset defined as a 2-tuple $\langle \mathcal{I}, m_{\mathcal{D}} \rangle$ with a function $m_{\mathcal{D}} : \mathcal{I} \rightarrow \mathbb{N}_0$ describing the multiplicity of each item in $\mathcal{D}$. Let $\mathcal{S} = \langle \mathcal{I}, m_{\mathcal{S}} \rangle$. We call $\mathcal{S}$ a *sample* of $\mathcal{D}$, iff $\forall x \in \mathcal{I} : m_{\mathcal{S}}(x) \leq m_{\mathcal{D}}(x)$. We define $f_{\mathcal{X}} : \mathcal{I} \rightarrow [0, 1]$ to be a function mapping items to their relative frequencies within an arbitrary multiset $\mathcal{X}$, with $f_{\mathcal{X}}(y) = m_{\mathcal{X}}(y) / \sum_{x \in \mathcal{I}} m_{\mathcal{X}}(x)$, and let $f_{\mathcal{X}}^k$ be the k$^{\text{th}}$ largest frequency within $\mathcal{X}$ for $1 \leq k \leq |\mathcal{I}|$.
According to [43], we define the set of top-k frequent items as follows:

$$TOPK(\mathcal{D}, \mathcal{I}, k) = \{(x, f_{\mathcal{D}}(x)) \mid x \in \mathcal{I} \wedge f_{\mathcal{D}}(x) \geq f_{\mathcal{D}}^k\}$$

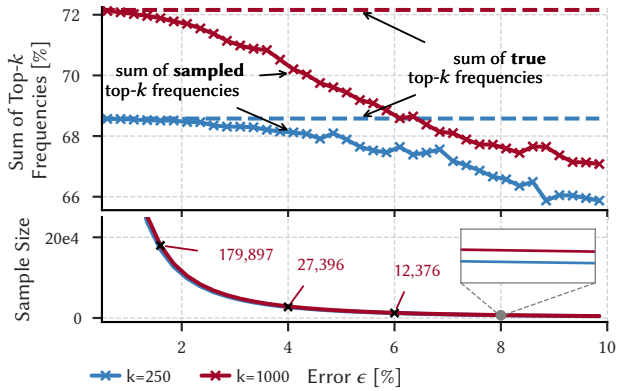

Figure 2: Sample sizes according to Equation (1) for error-bounded top-k analyzes for 1M items. Dashed lines denote the sum of the true top-k frequencies whereas solid lines show the sum of the sampled top-k item frequencies. The workload is generated using a Lognormal distribution. While $\epsilon < 5\%$ does not yield considerable precision gains, it yields larger sample sizes. Experiments using other distributions show similar results and can be found in the online appendix.

An ϵ-approximation to $TOPK(\mathcal{D}, \mathcal{I}, k)$ is a set W of k pairs (x, f) such that $x \in \mathcal{I}, f \in [0, 1]$, and for which the following holds:

$$\forall (x, f) \in W : f_{\mathcal{D}}(x) \geq f^k_{\mathcal{D}}(x) - \epsilon$$
$$\forall (x, f) \notin W : f_{\mathcal{D}}(x) < f^k_{\mathcal{D}}(x) + \epsilon$$
$$\forall (x, f) \in W : |f - f^k_{\mathcal{D}}(x)| \leq \epsilon$$

We use the equation introduced in [43] to calculate the required sample size $|\mathcal{S}|$ for an ϵ-approximation at a probability of $1 - \delta$ with $\delta \in (0, 1)$.

$$|\mathcal{S}| = \frac{2}{\epsilon^2} \ln \frac{2n + k(n-k)}{\delta}, \text{ with } n = |\mathcal{I}| \quad (1)$$

In Figure 2, we visualize the classification precision and the required sample sizes for varying error rates ϵ. In the upper plot, we compare the sum of the top-k frequencies based on the entire dataset (dashed line) to the one based on the sample (solid line). The lower plot shows the required sample size based on Equation (1), where we observe fast-growing samples for decreasing ϵ. We conducted this experiment for other distributions as well (see online appendix) and found that $\epsilon = \delta = 0.05$ results in an overall frequency decrease of at most 2.5% for $k \leq 1000$, which provides a reasonable trade-off between sample size and accuracy for our application.

In the following section, we present our workload-adaptive approach which internally uses Equation (1) to calculate sufficient sample sizes for error-bounded top-k analyses. The result is used to adjust the node encodings adaptively in hybrid index structures.

3 ADAPTIVE HYBRID INDEXES

Many DBMS indexes are designed to equally support all possible access types such as lookups, updates, inserts, or deletes, by using *universal encodings*. An example of a universal encoding can be found in traditional B+-tree implementations (e.g. Postgres [48] and Umbra [40]): all leaf nodes use the same encoding where a fixed

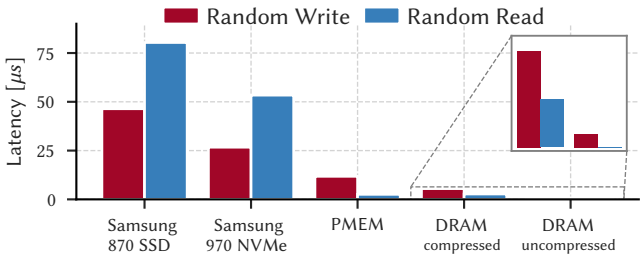

Figure 3: Read and write latencies to LZ4-compressed and uncompressed B+-tree leaf nodes having an average occupancy of 70% and being stored on different storage devices. The experiments were carried out on an Intel Xeon 6212U CPU (24 cores) equipped with 192GB DRAM and 768GB Intel Optane persistent memory. Before each leaf node access, we drop the caches to get IO-related latencies more accurately.

number of key and value slots are pre-allocated to allow efficient reads *and* writes. While such encodings simplify the implementation, on the one hand, they often result in space overhead. In some cases, this overhead might also result in indexes larger than main memory, leading to significant performance degradations due to paging. Therefore, it is attractive to minimize the space overhead of an index to allow pure memory residency.

If data does not fit in memory, buffer managers or more lightweight approaches such as LeanStore [30] will manage the page replacements. Despite recent advances in SSDs and NVMe-devices, I/O operations are still multiple orders of magnitude slower than memory accesses [52]. In Figure 3, we experimentally compare lookup and insert operations on uncompressed and compressed B+-tree leaf nodes. With the leaf nodes having an average occupancy of 70%, LZ4-based compression allowed to reduce the storage overhead by up to 47%. On-the-fly (de-)compression of in-memory nodes is faster by multiple orders of magnitude compared to (de-)compression of disk-resident nodes but much slower compared to uncompressed in-memory nodes. Therefore, applying more compact or even compressed encodings to *rarely accessed parts* might improve the overall latency of indexes by reducing storage overhead and preventing expensive I/O operations.

However, the main problem of index structures with different encoding schemes is that the actual workload is *not available beforehand* – instead, indexes get optimized for all possible access types *at development time* by applying universal encodings. Using different encodings, therefore, requires us to get more fine-grained information at *run-time*.

There are two different approaches to collect the required information. In a *decentralized* scheme, we would store tracking information in the index structure itself. For example, we could add an information unit (IU) that contains the last access time, the number of reads, writes, etc. for each index part. Such intrusive changes, however, add space overhead to *all* parts of the index – even to the never-accessed ones. Instead, we propose a new *centralized* approach, which stores IUs for accessed parts only. We combine it with lightweight sampling to reduce tracking-related overhead.

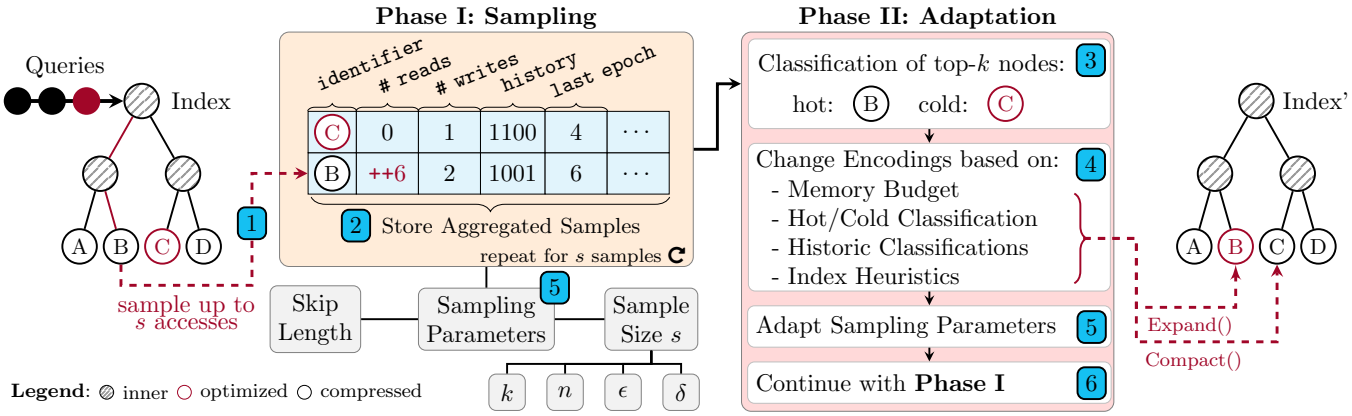

Figure 4: Basic overview of our workload-sampling approach applied to the example of a tree-like index structure.

In Figure 4, we show a conceptual overview of our approach applied to a tree having both a performance-optimized and a space-optimized encoding for the leaf nodes which is similar to our proposed Hybrid B+-tree in Section 4.1 (for implementation details, please refer to Section 3.1). To identify an appropriate encoding for each leaf node based on its access frequency, our approach has a sampling and an adaptation phase.

Phase I – Sampling: As the index processes incoming queries, it traverses the internal structure from top to bottom. Based on a predefined, adaptive skip length *sk*, every sk^{th} leaf node access gets sampled 1. For each sample, the leaf node and the access type (cf. Figure 4 *read* access to node Ⓑ) get passed to the *adaptation manager* and stored in aggregated form 2. To consider the most recent accesses of the *current* sampling phase only, we enumerate the sampling phases by increasing epoch numbers and let aggregates store the epoch in which they were accessed last. In Section 3.1.3, we discuss the usage of epochs in more detail. When the required sample size *s* has been reached, the approach continues to the *adaptation phase*.

Phase II – Adaptation: Based on the aggregations, all samples get classified as either *hot* or *cold* 3. As an abbreviation, let $\mathcal{N}_e$ denote the set of nodes accessed during epoch *e*. Therefore, we run a top-*k* classification on all nodes in $\mathcal{N}_c$ with *c* being the current epoch, while nodes that were not sampled during *c* are considered to be cold. In this context, we set *k* to the number of *theoretically expandable nodes* based on the index size and the memory budget. Affording *n* additional bytes, we can keep up to $8n$ of the last classification results as historic information to support future encoding decisions.

Next, the adaptation manager determines *promising* encoding changes based on the available memory, the current and the historic classifications, and heuristics 4. These heuristics are index-dependent as they take encoding migration costs and performance gains into account (cf. Section 3.1.4).

For example, as in Figure 4, node Ⓑ, which has been classified as *hot*, gets expanded from the compressed to the performance-optimized encoding, whereas node Ⓒ, which is no longer hot, gets compacted the other way around.

Sample-based Classification: As tracking all of the internal node accesses will cause severe performance overhead (cf. Section 3.1.3), the adaptation manager considers a sampled subset of queries only. In this context, we introduce two relevant sampling parameters: the *skip length* and the *sample size*. The skip length corresponds to the number of skipped queries between two samples, while the sample size defines the number of sampled accesses before starting Phase II. Based on the skip length *sk*, the costs for sampling one access get amortized over *sk* queries: larger skip lengths will reduce the costs per query and vice versa. However, larger skips will also increase the time until optimizing frequently accessed nodes. The adaptation manager sets the skip length adaptively at run-time: frequent encoding migrations will lead to *smaller* skip lengths so that the hybrid index can quickly adapt to workload changes. For more details, please refer to Section 3.1.3.

To bound sampling errors, we introduced Equation (1) in Section 2 to get the sample sizes for error-bounded top-*k* approximations, with *n* being the number of leaf nodes, ϵ denoting the classification error, and δ representing its reliability. Based on the results in Figure 2, we set $\epsilon = \delta = 5\%$ as the default values because they provide a reasonable trade-off between sample size and precision. To set *k*, we approximate the number of nodes that could be expanded without exceeding the memory budget. Assuming a tree has n_c compressed and n_u uncompressed nodes, where compressed/uncompressed nodes use m_c/m_u bytes each on average. For a memory budget *mb*, we can approximate $k = (mb-(n_c \cdot m_c+n_u \cdot m_u))/(m_u-m_c)$.

At the end of each adaptation phase, the adaptation manager changes both parameters skip length and sample size adaptively 5.

We next present the architecture, the unified interface, and the different strategies for sampling-based classifications in detail. We then discuss supported user-defined parameters and show how to use heuristics to determine nodes for potential encoding migrations.

3.1 Architecture and Interface

While Figure 4 shows the concepts of adaptive hybrid indexes, this section describes the concrete steps to implement them. We present the framework's architecture and its unified interface in more detail. In the rest of this paper, we refer to the controlling instance of the workload adaptation framework as *adaptation manager*.

3.1.1 Tracking Granularity and Encodings. Before making a hybrid index adaptive, we must determine the *tracking granularity*: the basic unit (e.g., key, node, bucket) where we collect statistics and apply encoding migrations. We then design different encodings for this basic unit. Each encoding comes with different trade-offs in read/write performance and space efficiency. Based on the different encoding characteristics, we implement heuristics that map a basic unit to an encoding by taking its sampled access information and the available resources into account. We describe the usage of such heuristics in more detail in Section 3.1.4.

We must also provide unique *identifiers* for the basic units. In most cases (e.g., B-tree nodes) we simply use pointers. For others such as succinct index representations, we use position offsets.

In our approach, each hybrid index keeps its adaptation manager as a member variable (cf. Listing 1, line 50). This instance is then used for all workload adaptation-related tasks such as monitoring the space consumption and storing the fine-grained access information, which separates index- from sampling-related code.

3.1.2 Interface. As a next step, we identify those functions which access or modify the index at the predefined granularity, such as lookups, inserts, or iterator increments and dereferencing operators. Each of the relevant functions first checks whether the current access is considered to be a sample (Listing 1, lines 38 and 42). Depending on the return value we conditionally pass the accessed part (here `cur_node`) alongside the access type to the adaptation manager using the `Track`-function (lines 39 and 43).

To enable the adaptation manager to change the encoding of a tracked part, we implement a callback function that handles the migration logic between different encodings (line 49). As an example, for a B+-tree with compressed and uncompressed leaf nodes, we provide a callback function implementing the compression and decompression of leaf nodes.

3.1.3 Phase I: Sampling Phase. During the sampling phase, the hybrid index invokes the adaptation manager to track a sampled subset of basic units and corresponding access types. For each unit, the adaptation manager maintains individual access statistics (cf. Listing 1, line 6), where accesses are grouped by access type (read, insert, update, and delete).

To consider only the sampled accesses of the *current* phase, the adaptation manager maintains a global epoch counter (cf. line 28), and each access statistic stores the epoch of last access (cf. line 6). Before updating access statistics, we first check if its epoch matches the global epoch. In the case of different epochs, we first reset the aggregate counters and set the local to the global epoch before registering new accesses. Storing the epoch of the last access further adds new relevant information when deciding which node encodings should be changed.

To efficiently map identifiers to their access statistics (cf. line 25), we use a high-performance hop-scotch hash map for single-threaded execution [6], and a concurrent cuckoo-based hash map for parallel workloads (cf. Listing 1, line 25) [34]. It allows concurrent readers and writers while it retains high throughput under contention.

As an optimization, we install a *bloom filter* in front of the hash table to prevent cold nodes from being tracked accidentally (cf. line 26). Before an identifier gets tracked in the hash map, it must be added to the filter first. Only in the case the identifier is already

```
1  // AdaptationManager.hpp
2  template <class Index, typename Identifier, typename Context>
3  class AdaptationManager {
4   public:
5    enum AccessType { READ, WRITE, UPDATE, DELETE };
6    enum AccessStats { size_t reads, size_t writes, BitSet
     ↪  last_classifications, Epoch last_epoch, ... };
7    explicit AdaptationManager(Index *index);
8    bool IsSample() {
9      if (--skip_length == 0) {
10         skip_length = global_skip_length_.load(); // synchronized
11         return true;
12     }
13     return false;
14   };
15   template <typename... Args>
16   void Track(Identifier&, AccessType&, Args&& ...);
17   void UpdateContext(Identifier&, Context&);
18  private:
19   void Classify(); // Classify nodes as hot and cold
20   void Adapt(); // Start encoding migrations
21   atomic<size_t> global_skip_length_; // Adaptive parameter
22   static thread_local size_t skip_length;
23   atomic<size_t> global_sample_size_; // Adaptive parameter
24   static thread_local size_t sample_size;
25   HashMap<Identifier, pair<AccessStats, Context>> samples_;
26   BloomFilter<Identifier> filter_;
27   Index *index_;
28   Epoch current_epoch_;
29 };
30 // HybridIndex.hpp
31 #include "AdaptationManager.hpp"
32 template <typename K, typename V>
33 class HybridIndex {
34   struct Node {...}
35   friend class AdaptationManager;
36  public:
37   V Lookup(const K& k) { // leave out lookup logic
38     if (adapt_manager_.IsSample())
39         adapt_manager_.Track(node, READ)
40   }
41   bool Insert(K& k, V& v) { // leave out insert logic
42     if (adapt_manager_.IsSample())
43         adapt_manager_.Track(node, INSERT)
44   }
45  private:
46   // Callback functions invoked by adapt_manager_
47   size_t GetUsedMemory();
48   Encoding EvaluateHeuristic(const AccessStats&);
49   void Encode(Node*, EncodingSchema& /*target*/, Node*
     ↪  /*parent*/);
50   AdaptationManager<HybridIndex<K,V>, Node*, /*Parent-*/ Node*>
     ↪  adapt_manager_;
51 };
```

Listing 1: Simplified draft of the workload sampling interface. We left out constructors, index-dependent lookup- and insert-function logic, and thread-local sampling maps.

contained in the filter, it gets added to the hash map. We reset the filter after each sampling phase. The bloom filter is configured to use 10 bits per item and its capacity is set to half of the sample size.

To reduce the tracking-related overhead, (e.g. hashing the identifiers, accessing and modifying the aggregated sampling statistics) we do not consider *all* node accesses, but a sampled subset only. As our approach aims to classify index parts into categories such as hot and cold, the chance to miss frequently accessed parts decreases inversely proportional with increasing access frequencies.

While sampling reduces the *tracking-related* overhead, it introduces *new* overhead to decide which accesses get sampled. Instead

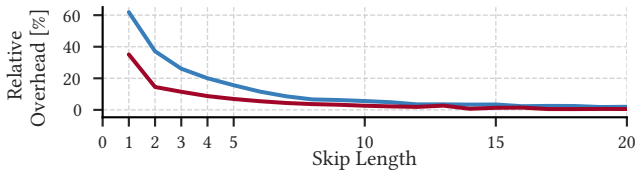

Figure 5: Relative sampling overhead for different skip lengths using the Hybrid B+-tree (cf. Section 4.1). The well-known STX-B+-tree represents the baseline. The blue line additionally samples leaf nodes accesses and collects individual tracking information. It shows the relative overhead of the workload sampling. The red line shows the performance overhead for the filter-based optimization. Further experiments can be found in the online appendix.

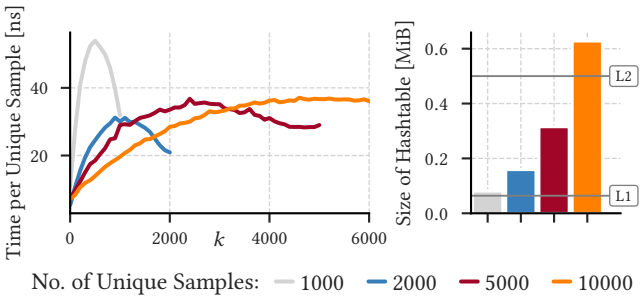

Figure 6: The left plot shows the classification overhead per sample for different sample sizes and different values for k. In the right subplot, we show the size of the hash map that stores the individual samples and the access statistics.

of probabilistically deciding for each access whether it gets sampled or not, Vitter et al. suggest minimizing sampling-related overhead by defining so-called *skip lengths*: a skip length defines how many accesses are skipped between two samples [49]. We experimentally evaluated the sampling overhead for different skip lengths using the OSM dataset and a log-normal distributed workload in Figure 5. The STX-B+-tree is the baseline in this experiment, and the tree represented by the blue line additionally collects access statistics. For a skip length of 0, which means that all accesses get sampled, we observe significant overhead of up to 61.9% compared to the baseline. However, the sampling overhead quickly decrease for larger skip lengths, e.g. 1.6% for a skip length of 20. The red line shows the overhead after adding the additional bloom filter. The filter prevents cold nodes from being tracked and reduces sampling-related overhead significantly. While this experiment shows results for the log-normal workload, other workloads show similar overhead.

We can further reduce contention by defining *one skip per thread* (line 24): decrementing the thread-local skip (line 9) does not require synchronization. Only in case the skip becomes zero, we reset the skip to the global skip using an atomic load instruction (line 10).

In some cases we need a way to store additional *context information* alongside the identifier to allow for efficient encoding migrations. For example, we could integrate the adaptation manager to identify hot and cold leaf nodes in a B+-tree. Whenever we expand or compact a node, its parent must efficiently be made available to change the corresponding child-node pointer. Based on *variadic template arguments* and *perfect forwarding*, hybrid indexes can efficiently pass arbitrary context information to the adaptation manager without requiring changes of our framework.

As context information might change over time (e.g. parent changes because of node splits), our framework allows to propagate context changes to the adaptation manager (cf. line 17).

3.1.4 Phase II: Adaptation Phase. When the sample reaches the predefined size, the adaptation manager terminates the sampling and passes over to the adaptation phase, which consists of three steps (cf. 3 - 5 in Figure 4). First, the top-k frequent nodes of the last sampling phase get labeled as hot, the rest is considered to be cold. Second, based on the classification and the index heuristic function, the adaptation manager determines the most suitable encoding for each tracked node and applies the appropriate encoding migrations. Last, the adaptation manager adapts the parameters skip length and sample size before the next sampling phase starts.

Classification: We implement the top-k analysis using a priority queue based on a binary heap having a capacity of k. In our experiment, we use the sum of the read and write access counters as default priority. However, we could also assign custom weights to the different access counters. Then, we traverse the hash map and insert those aggregated samples whose epoch matches the global epoch, and label them as hot. Nodes, which were not accessed during the last sampling phase will not be inserted into the priority queue at all – instead, we can directly classify them as cold. When nodes are displaced from the priority queue, they are marked cold again. Therefore, we find the top-k frequent items in a single pass and classify all nodes accordingly. This algorithm runs in $O(u(1 + log(k)))$ with u being the number of unique samples, and the space to store the priority queue is $O(k)$.

We experimentally evaluated the classification performance for different numbers of unique samples and different values for k in Figure 6. For $k \approx s/2$, the sum of heap inserts and removals reaches its maximum, while it decreases for smaller and larger k, explaining the different latencies. Assuming a classification latency of 60ns per sample and a skip length of $sk = 20$, the classification overhead per query can be amortized to 60ns/20 = 3ns.

Identifying hot nodes based on sampling will also introduce inaccuracies, therefore, we further back up future encoding decisions by keeping the most recent n classifications. In our example implementation, we use one additional byte to keep the last eight classifications. This information can be used in the heuristics to further improve encoding decisions.

Heuristics: After the classification, the adaptation manager might optionally change the encoding for tracked parts. Making *optimal* encoding-decisions is not possible, as there will be sampling inaccuracies and future queries and accesses are not known beforehand. However, we can *react* to current workload developments based on the sampled access statistics. Therefore, *context-sensitive heuristic functions* (CSHF) support the workload manager to decide which encoding migrations *might* improve the performance. As shown in Figure 7, a CSHF is similar to a decision tree that takes sampled access statistics and other context information into account and returns an encoding. Branches represent *decisions*, while

leaf nodes contain the *suggested encoding*. Furthermore, the CSHF can decide to stop tracking of specific nodes, e.g. if they are cold or were not sampled for a longer time. Each hybrid index will implement its own, tailored CSHF as it must take the different encodings and space-performance implications into account (cf. line 48).

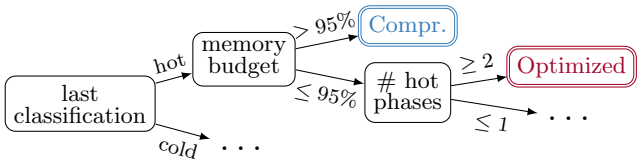

Figure 7: Heuristic functions propose target encodings depending on a variety of factors such as the current and the last classifications, system resources, or the last access.

Next, the adaptation manager evaluates the CSHF for all tracked items and if required, migrates the basic units to their suggested encoding. Therefore, the hybrid index implements a callback function (cf. line 49) to handle the migration between different encodings.

Sampling Parameters: Providing an equation or a metric that adequately considers all parameters is challenging. Instead, we identified three important parameters *sample size*, *skip length*, and *encoding migration costs* and provided experimental validation for each. Based on these experiments, we let the adaptation manager set the parameters adaptively at runtime. E.g., the adaptation manager will calculate the new *sample size* based on Equation (1) as well as a new *skip length*. As we discussed earlier, a smaller skip length allows hybrid indexes to adapt more quickly to workload changes, but it imposes more sampling overhead. And we use the number of node encoding changes in the current adaptation phase to approximate workload stability. For example, if the migrated nodes make up less than 10% of the sampled accesses, the skip length will increase to reduce the sampling overhead. Contrary, if the share exceeds 30%, we decrease the skip length and therefore increase the sampling frequency. In our example implementation, the adaptation manager will adaptively set the skip length within the range [50, 500]. Additionally, the adaptation manager could randomize *sk* in a limited range to cope with query patterns.

3.1.5 Concurrency. While hybrid indexes work best under skewed workloads, concurrency requires contention and synchronization to be kept at a minimum. We compare and evaluate two approaches: (1) **GS**: All worker threads (WT) track samples in a global cuckoo hash map which is optimized for concurrent readers and writers [34]. During adaptation, the map gets locked globally to process each sample. (2) **TLS**: All WTs track the samples in thread-local maps, which get merged once the target sample size is reached. In both approaches, one WT runs the adaptation, while the remaining WTs continue with the sampling phase. While GS optimizes space efficiency, TLS allocates more memory for thread-local sampling.

3.1.6 Optional Memory Budget. Our framework allows to set either an *absolute* or a *relative* memory budget. While absolute budgets are suited for read-only workloads, relative budgets allow us to define an average ratio of bits per item and therefore provide more flexibility with inserts and deletes. In other words, a memory budget

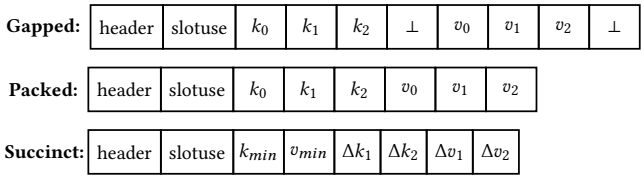

Figure 8: Three different leaf node encodings are used in our Hybrid B+-tree implementation. The Gapped encoding stores a fixed number of slots with possible empty slots (⊥) at the end. The Packed encoding stores keys and values densely packed, and the Succinct layout further employs frame-of-reference encoding.

lets us define a compression ratio in which the index structure can be adaptively optimized. During execution, the adaptation manager optimizes the index while keeping its size below the upper bound.

3.2 Trained Hybrid Indexes

In some contexts, dataset and workload remain stable for a longer time, or a workload prediction might be available beforehand. E.g., self-driving database systems as proposed by Pavlo et al. in 2017 [41] build indexes based on predicted workload patterns. Therefore, more space-efficient hybrid indexes could make use of such fine-grained predictions for training. Furthermore, *online* workload adaptation imposes additional overhead to collect, aggregate, and classify samples, so it might be desirable to train hybrid indexes based on previous workloads in these cases beforehand.

Therefore, our framework also implements an offline solution for hybrid indexes: given a predicted or a historic workload, the adaptation manager analyzes the access patterns and ranks the nodes according to their access frequencies. Starting with the most promising nodes, the adaptation manager optimizes the nodes until all nodes are optimized or the memory budget is reached.

4 EXAMPLE IMPLEMENTATIONS

In this section, we apply our approach to two state-of-the-art index structures: B+trees and radix trees.

4.1 Hybrid B+-Tree

Although invented for disk-based database systems, B+-trees are still the most widely-used indexes, even for in-memory DBMSs [53]. Most implementations make use of two encoding schemes: one for inner and another one for leaf nodes. While both node types have a fixed number of slots, long-running systems with millions of insert and delete queries lead to unused slots. While these empty slots do allow for efficient inserts and deletes, they also often result in space utilization below 70% [8].

4.1.1 Tracking Granularity and Encodings. We first determine a suitable tracking granularity. For the B+-tree, *leaf nodes* make a good candidate as they make up the largest part of the overall data structure and maintain all keys and values. Figure 8 introduces three leaf node encodings that trade-off space and performance differently.

Gapped is the traditional, universal encoding for B+-tree leaf nodes. Those nodes support all access types efficiently by accepting

free slots (⊥), but they require a fixed amount of memory. The more space-efficient *Packed* layout allocates memory for the *used* slots only. This *packed* representation supports efficient read, update, and delete operations (using tombstones), but does not support efficient inserts. It stores the number of elements and two arrays for keys and values. As an alternative, the Succinct encoding trades performance for space efficiency more aggressively. For a leaf node, it combines frame-of-reference (FOR) encoding with bit packing to store the keys and values in a compressed fashion. The first (and smallest) key/value is stored separately. While this node layout still allows for random access, it requires additional instructions and bitwise operations to access keys and values. This results in higher access latencies compared to the gapped and packed encoding schemes for smaller indexes, however, for larger indexes exceeding the caches, the succinct layout will cause fewer cache misses which *might* outweigh the additional CPU costs. In Table 1, we provide a performance analysis for the different layouts. Gapped and packed nodes achieve significantly higher throughput, whereas succinct nodes require 73% less space on average compared to gapped nodes. Furthermore, special allocators and memory pools can optimize the allocation of differently sized nodes to prevent heap fragmentation.

4.1.2 Encoding Migrations. In Figure 9, we experimentally evaluate the migration costs between the different node encodings for two index sizes. The left index consumes around 10MB and entirely fits into the L3 cache, while the right index needs around 1GB. For both indexes, we observe significant overhead for switching between the *succinct* and one of the other encodings: in these cases, the migration will modify the physical key and value representation and comes at the cost of additional instructions. In contrast, migrating between packed and gapped node layouts is cheaper as these migrations use a system call to copy keys and values.

4.1.3 Tracking Leaf Nodes. Tracking leaf nodes during reads and inserts is straightforward as the leaf and its context is directly available. Scans, however, require minor, structural changes: each inner node has a link to its right sibling, and iterators keep a pointer to the current parent. In the case of a sampled leaf node access through an iterator, the parent can be efficiently retrieved.

4.1.4 Handling Updates. Inserts and deletes will cause split and merged nodes. In case a leaf node gets a new parent, this information must be propagated to the tracking framework: we pass the leaf node and its new parent to the adaptation manager, that will update the context information of the *actually tracked* leaf nodes only.

Table 1: Different leaf node encodings storing 64-bit key-value pairs of the OSM dataset (cf. Section 5.1) and their performance implications on uniform lookups for a node occupancy of 70%.

Leaf Node Encoding	Average Size	Lookup Latency	Instruc.	LLC Misses	Branch Misses
Gapped	4096B	56ns	85	2.1	4.44
Packed	2872B	57ns	84	1.4	4.46
Succinct	1076B	125ns	341	1.1	6.69

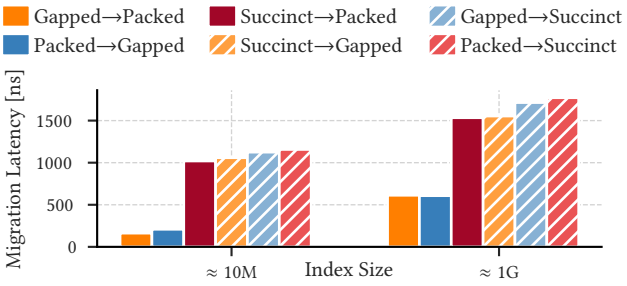

Figure 9: Evaluation of the migration costs between different leaf node encodings for two selected index sizes. The used CPU is equipped with an L3 cache size of 64MB.

4.1.5 Concurrency Control. We synchronize Hybrid B+-tree using Optimistic Lock Coupling (OLC) as described in [32]. Each node stores a lock and an atomic version counter. Compared to lock coupling, OLC scales significantly better on multi-core systems, because it minimizes the number of acquired locks.

4.2 Hybrid Trie: ART and FST

Tries are pointer-based index structures and are mainly used to index variable-length keys. Especially on modern hardware, research has shown that tries achieve high performance [10, 13, 31, 35, 50, 55]. Compared to B-trees, tries do not store entire keys on each level, but they index key suffixes, also referred to as *labels*, instead, where each level stores the next $k > 0$ bits of the key.

The Adaptive Radix Tree (ART) was introduced in 2013 and represents the default index structure in HyPer [31]. It allows to dynamically choose between four differently sized node types based on the number of labels stored within a node. While each node type has different implications on lookup performance [19], the node type is chosen based on the indexed keys only and does not depend on the actual workload. ART requires k to be 8 and therefore limits the maximum fanout to $2^8 = 256$.

Besides ART, there is another state-of-the-art trie called Fast Succinct Trie (FST) which has been introduced by Zhang et al. in 2018 [55]. Compared to ART, FST does not store child pointers to traverse the index structure, but instead, it *computes* the position of the next node based on two bitmaps, one storing the existing labels and another one maintaining the information whether a path terminates. While FST does not impose any restrictions on k, we assume $k = 8$ in the following for the sake of simplicity. Furthermore, FST uses two different encoding schemes for upper and lower levels: The upper, more frequently accessed parts are encoded using a more performance-optimized and space-demanding encoding, referred to as *FST-dense*, where each node stores the key-labels implicitly by using 2^k bits per node, which allows for fast random access within each node. The lower levels use the *FST-sparse* encoding which stores existing *labels* explicitly. This *might* reduce the space usage[1] but also requires an explicit search within the nodes and therefore more computations for sparse-encoded nodes.

[1]The sparse encoding requires less space compared to the dense encoding when the average number of stored labels l within the nodes is smaller than $256/8 = 32$.

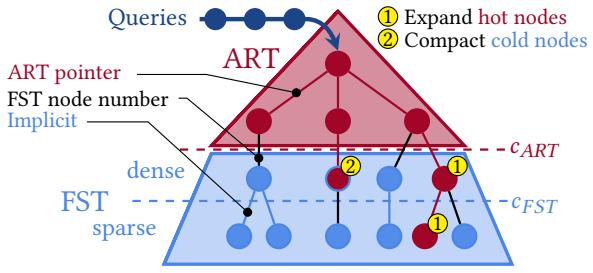

Figure 10: Our workload-adaptive Hybrid Trie. It combines the Adaptive Radix Tree and the Fast Succinct Trie level-wise at build-time. Combined with our sampling framework, it supports branch-wise refinements at run-time.

In Table 2, we compare the sizes of ART and the two FST encodings for the prefix-random dataset (cf. Section 5.1). While ART allows for faster lookups, it requires significantly more memory. FST requires additional instructions to compute the position of the next node and value resulting in decreased performance. For FST-sparse, we measured more cache misses than for ART, which can be explained with the efficient path compression in ART: while FST-sparse requires less memory, its lookups will traverse more nodes on average and therefore cause more cache misses.

Based on the work of Anneser et al. in 2020 [9], we introduce Hybrid Trie: an adaptive, level-wise combination of ART and FST.

4.2.1 Tracking Granularity and Encodings. Hybrid Trie combines ART and FST level-wise: levels 0 to c_{ART} (incl.) are represented by ART, levels between c_{ART} and c_{FST} (incl.) are represented by FST-dense, and the remaining levels are encoded using FST-sparse, as illustrated in Figure 10. We chose the level-wise combination with ART being at the top based on the fact that all queries start at the root node and ART achieves significantly higher throughput.

To allow for more fine-grained control and branch-wise expansions *beyond the cutoff level* c_{ART}, we extend Hybrid Trie with vertical refinements. While ART uses pointer tagging to differentiate *pointers* and inlined *TIDs*, we use an extra bit to further differentiate the case of inlined FST node numbers. The tagged pointers can then be used as unique identifiers by the adaptation manager.

4.2.2 Interface and Callbacks. Since FST is a static index supporting only lookups and range scans, we do *not* handle inserts as they would require a complete *rebuild* of FST each time. To support efficient inserts, we experimented with storing multiple FSTs (one per "cold" subtree) instead of a single, global one. However, as each FST adds some storage overhead (for header information and auxiliary data structures), this approach did not pay off. We hence leave inserts for future work.

Next, we identify the functions accessing nodes below c_{ART}. In Listing 2, we show how the simplified lookup code integrates with the sampling framework. To enable fast node migrations, we provide additional context for each tracked identifier: we store its parent, the key label within the parent, and the FST node number.

The callback function `Encode(...)` implements the migration logic between ART and FST nodes. Compacting ART nodes to the FST representations (cf. ② in Figure 10) requires deleting the expanded node and replacing the tagged identifier within the parent

```
1  V Lookup(const K& key) {
2    const bool isSample = adapt_manager_->IsSample();
3    Node* node = root_;
4    while (node != nullptr && isARTPointer(node)) {
5      node = findChild(node, key[level++]);
6      if (isSample && level > c_art)
7        adapt_manager_->Track(node, READ, ...);
8    }
9    if (isFSTNode(node))
10     return fst_->Lookup(getFSTNode(node), key, level);
11   return getValue(node);
12 }
```

Listing 2: Simplified lookup code for Hybrid Trie. The highlighted lines handle the required calls to the workload sampling framework. In line 7, we dropped additional arguments such as the parent identifier and the key-part at the current level. This function is intentionally not declared `const` as it may modify the internal structure.

Table 2: Space and performance metrics for different trie indexes measured for the prefix-random dataset and workload (cf. Section 5.1).

Index	Size	Per Lookup-Query			
		Latency	Instruc.	LLC misses	Branch misses
ART	274MB	81ns	177	8.49	0.03
FST-dense	116MB	206ns	675	6.33	1.82
FST-sparse	104MB	576ns	4337	9.2	9.64

node with the FST node number. Expanding FST nodes to ART nodes (cf. ①) requires us to determine the appropriate ART node type based on the number of labels within the node. In both cases, we retain the historic access statistics in the workload tracking.

We experimentally evaluated the latencies for migrating nodes between ART and FST. Migrations from FST to ART cause overhead of up to ≈5000ns on average (assuming a node occupancy of 50%): labels stored within the FST node must first be collected and then inserted into the new ART node. Migrating the other way around takes up to ≈100ns only, as it does not involve the construction of a new node, but the deletion of the existing ART node and its replacement in the parent node with the FST node number.

5 EVALUATION

We conduct all experiments on a 16-core AMD Ryzen 9 3950X CPU @ 3.5GHz equipped with 64GB DDR4-2667 RAM and compile the `C++` code with GCC 9.3.0, using the flags `O3` and `march=native`. Please note that the CPU overhead for sampling, compacting, and expanding nodes are already included in the shown performance.

5.1 Datasets and Workloads

The *OSM*-dataset [26, 36] comprises 400M uniformly sampled Open Street Map locations represented as 64-bit S2-cell-identifiers [4]. We further use the RocksDB tool dbbench to generate 64-bit user-ids

Table 3: Operation counts and distribution types for each workload. The synthetic workloads W1.1-W2 were generated by us, W3 is a realistic workload generated using RocksDB's dbbench [14], and W4 has been generated using YCSB [17]. The scan length is uniformly distributed within [10, 50], and for W4 [100, 250].

	Reads	Scans	Inserts
W1.1	49% Zipfian	49% Zipfian	2% Zipfian
W1.2	49% Normal	49% Normal	2% Zipfian
W1.3	49% Lognormal	49% Lognormal	2% Lognormal
W2	56% Lognormal 24% Uniform	20% Lognormal	
W3 [14]	100% prefix-rand.		
W4 YCSB	75% Zipfian	25% Zipfian	
W5.1		20% Zipfian	80% Zipfian
W5.2	20% Zipfian	80% Zipfian	
W6.1	100% Zipfian		
W6.2		100% Zipfian	

and anonymized workloads that contain common patterns seen at Facebook [3, 14]. Besides fixed-size keys, we use a dataset of 33M unique email addresses (host-reversed, e.g. foo.com@) drawn from a real-world dataset (average length = 22 bytes, max length = 49 bytes). Further optimizations such as key compression are orthogonal to our approach: Adaptive indexes choose the most promising internal node encodings adaptively at run-time, while key compression, e.g. [56], is performed at key granularity.

Based on the datasets, we generate different workloads. Figure 11 visualizes the cumulative distribution functions (CDF) of the workloads W1.1 - W1.3 applied to the OSM dataset.

In Table 3, we show the number of operations and the used distribution for each workload and query type. We use the following *relative* distributions to decide on *record selections* and *scan lengths*: Zipf with $\alpha \in [1, 1.5]$ and N being the number of keys, Normal with μ=0.5 and σ=0.03, Lognormal with μ=0 and σ=0.1, and Uniform.

Additionally, we used RocksDB's dbbench to generate a more realistic workload based on the *prefix-random* configuration described by Cao et al. in 2020 [14]. They analyzed RocksDB workloads at Facebook and extracted common characteristics, which are re-generated by dbbench. Furthermore, they found a correlation between key prefixes and lookup frequencies: while most keys are not accessed at all, there are some *hot* key prefix ranges which are accessed frequently. We evaluate this workload using ART and FST. We use a custom read-only YCSB configuration with a hot set size of 1% of the dataset (cf. W4).

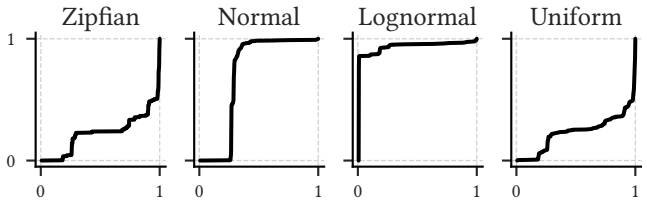

Figure 11: Cumulative distribution functions (CDFs) of the workloads (from left to right) W1.1, W1.2, W1.3, and a uniform distribution on the OSM dataset.

We further use the Yahoo! Cloud Service Benchmark (YCSB) [17] to generate a dataset of 200M key-value pairs (16 bytes each) and workload W4 with 200M Zipfian-distributed queries. Workloads W5.1 and W5.2 let us investigate the performance of adaptive indexes for write-dominated workloads, while workload W6 evaluates point lookups and scans on the mail dataset using Hybrid Trie.

5.2 Hybrid B+-tree

For the following evaluation of the Hybrid B+-tree, we assume an average leaf node occupancy of 70%. We refer to the adaptive Hybrid B+-tree as AHI-BTree.

In Figure 12, we use the OSM dataset and the workloads W1.1 - W1.3 to show the performance *developing over time* and the average space consumption for the adaptive and pre-trained Hybrid B+-tree and compare them to the *succinct, packed,* and *gapped* tree variants which do apply a single encoding to all of their leaf nodes. For each workload phase, we observe a short period of time in which the latencies of the adaptive index decrease, and then stabilize at a lower level. During this time, the workload adaptation detects frequently accessed nodes and migrates them to performance-optimized encodings. In contrast to the first and last phases, the second phase W1.2 is less skewed, which is the reason for the increased latencies. The adaptive tree achieves 85%, 99%, and 84% of the throughput of the performance-optimized Gapped tree on average per workload phase. At the same time, the adaptive tree reduces the memory footprint (2.36GB) by up to 72% compared to the Gapped tree (8.66GB).

To better understand the space-performance trade-off, we use the cost function $C = P^r S$ defined by Zhang et al. in 2018 [55], with P representing the performance (latency) and S representing the index size. The exponent r defines the relative importance between P and S: $0 \leq r < 1$ considers space to be more important, while $r > 1$ trades performance for space.

Figure 13 visualizes C for $r = 1$ (space and performance are equally important) by using blue curves and shows the *average* performance and *last measured* index size. Indexes on the same curve are considered to be "indifferent" in the space-performance trade-off. According to C, the succinct, adaptive, and pre-trained variants provide a better space-performance trade-off than the gapped and packed variants. For the highly skewed Lognormal workload W1.3, the adaptive tree achieves the *best* trade-off.

In Figure 14, we investigate to what extent the adaptive tree can leverage differently skewed workloads. We generate the workloads based on W1.1 for parameter $\alpha \in (0, 1.6]$. For $\alpha = 1$, our adaptive tree reduces the index size by 71%/59% while it increases query latency by 17%/7% wrt. the Gapped/Packed trees. With decreasing α, AHI-BTree cannot retain the high performance improvements: the access frequency of the 10K most frequent nodes (out of 2M nodes) decreases from 67% for $\alpha = 1$ to 45%/28%/11% for $\alpha = 0.9/0.8/0.7$. For this experiment, the break-even point is at $\alpha \approx 0.6$: for $\alpha < 0.6$, sampling overhead outweighs performance improvements due to node expansions, and for $\alpha > 0.6$, the contrary is the case.
Despite the sampling overhead, we observe no considerable performance decreases for AHI-BTree wrt. the Succinct tree (3% higher

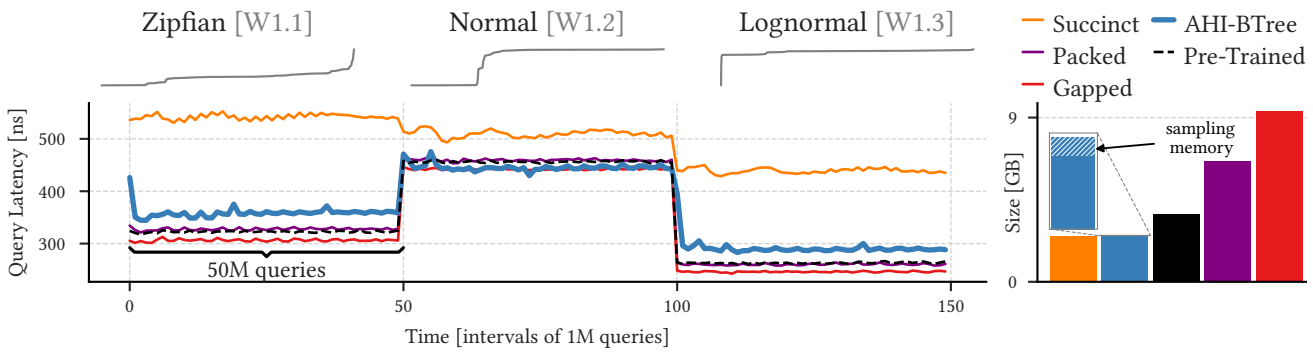

Figure 12: Query latency evolving over time for the OSM-dataset and three selected workloads using the Hybrid B+-tree and its baselines. Each workload phase comprises 50M queries and at the top, we show the corresponding CDF. For all phases, we observe a performance increase over time for the adaptive tree (AHI-BTree). The bar chart to the right shows the overall index structure sizes. The sampling framework takes up to 2.53MB, which is 0.1% of the adaptive index size of 2.36GB.

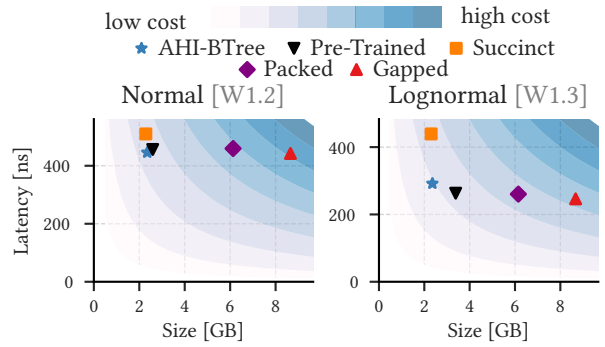

Figure 13: Average latencies and index sizes for B+-trees having different leaf encodings for the OSM dataset and workloads W1.2 and W1.3. The blue curves show a cost function that considers performance and space as equally important. Points on the same curve are considered to be *indifferent*.

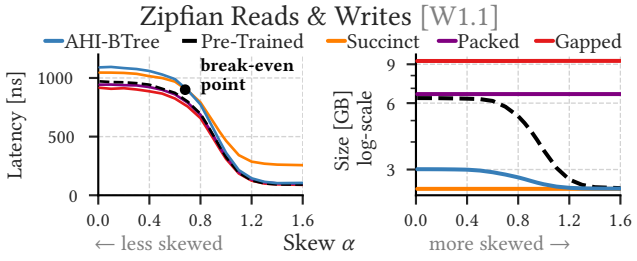

Figure 14: Average latencies and sizes of the Hybrid B+-tree indexing the OSM dataset for workload W1.1 for varying α.

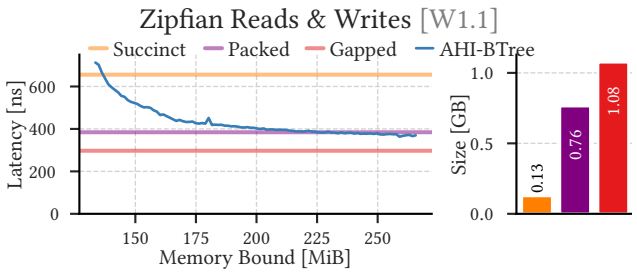

Figure 15: Latency and size of Hybrid B+-tree indexing 50M consecutive 64-bit keys for different memory budgets.

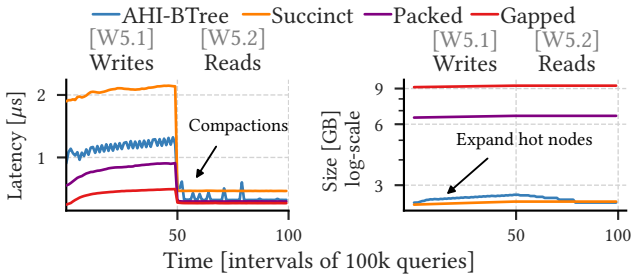

Figure 16: Latencies and index sizes for the Hybrid B+-tree running workload W5 on the OSM dataset. W5.1 is write-dominated and W5.2 is scan-dominated. We run both workloads consecutively.

latency) under less skewed workloads ($\alpha = 0.01$).
AHI-BTree eagerly migrates Succinct nodes to the Gapped encoding on inserts and defers their compaction until they are cold again. For $\alpha = 0.01$, inserts affect 26% of all nodes and the adaptive tree allocates 46% more memory compared to the Succinct tree.

Figure 15 shows the impact of the memory budget on the performance of AHI-BTree. With increasing budgets, AHI-BTree can expand more nodes to the performance-optimized encodings. As the most frequently accessed nodes get optimized first, the performance improvements per additional MB are larger for smaller memory budgets under skewed workloads.

Figure 16 shows the performance for AHI-BTree running workload W5. With Succinct nodes being optimized for read accesses only, insert operations during the write-intensive workload W5.1 require expensive changes to the node structure. While AHI-BTree uses the Succinct encoding as default for cold nodes, it eagerly migrates nodes to the Gapped encoding on inserts. At the beginning

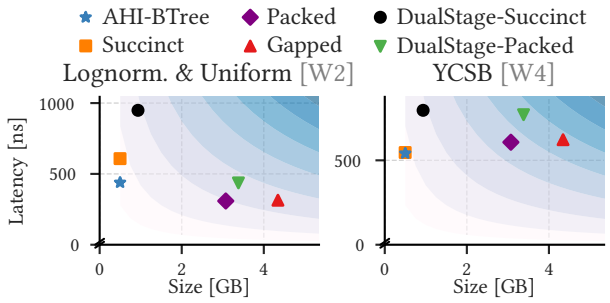

Figure 17: We compare the space and performance of our Hybrid B+-tree to the Dual-Stage Hybrid B+-Tree described in [53]. During the benchmark, the dynamic stage contains the latest inserted keys (5% of all data). The dataset contains 200M consecutive 64-bit keys and 64-bit TIDs.

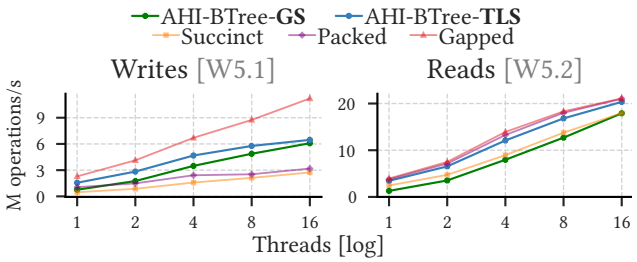

Figure 18: Average throughput for the two concurrent workload adaptations based on Global Sampling (GS) and Thread-Local Sampling (TLS) applied to the Hybrid B+-tree. We run both workloads W5.1 and W5.2 *separately* using different numbers of worker threads.

of W5.2, previously expanded nodes, which are rarely accessed in W5.2, get compacted again to reduce the memory footprint.

In Figure 17, we compare our approach to the *Dual-Stage (DS) framework* proposed by Zhang et al. in 2016 [53]. DS consists of a dynamic stage for recently modified data and a static stage for the remaining data. Inserts, deletes, and updates modify the dynamic stage, whereas reads first check the dynamic stage, and if the key was not found, they continue the lookup in the static stage. We can see that our approach outperforms DS in both dimensions, space and performance. Based on the access statistics, it allows for more fine-grained encoding decisions and can therefore leverage skew to a higher extent. Contrary, DS keeps only recently inserted or modified items in performance-optimized structures *independent* of the workload skew. As described in [53], we add the LevelDB [2] bloom filter to DS to further speed up lookups as it allows to *skip* the dynamic stage in most cases when the key does not exist there.

In Figure 18, we compare the performance of the two concurrent adaptation approaches (cf. Section 3.1.5) and apply them to the Hybrid B+-tree to run workloads W5.1 and W5.2 on the OSM dataset. We pin each thread to one logical core. For both workloads, thread-local sampling (TLS) achieves higher throughputs compared to global sampling (GS). GS locks the entire map during the adaptation phases and table resizing operations, which leads to high contention that severely degrades performance while executing the read-only

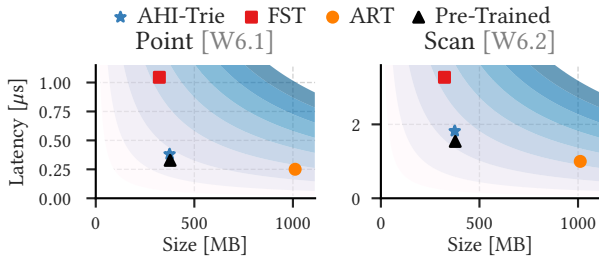

Figure 19: Space and performance for point lookups (W6.1) and scans (W6.2) on 33M unique email addresses.

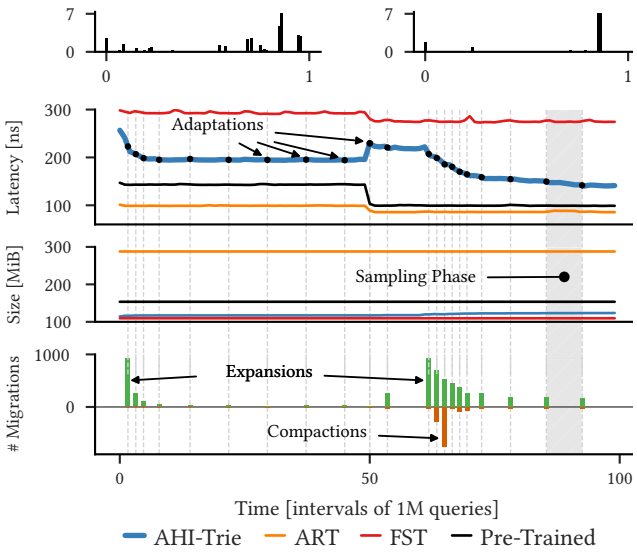

Figure 20: Latencies, index sizes, and encoding migrations shown over time for the prefix-random workload W3 and a dataset of 172M user ids. We split the workload into two phases, each containing different hot prefix ranges visualized by the two histograms at the top. The vertical dashed lines show the time of the adaptations. Sampling phases take place between two subsequent adaptations.

workload W5.2. The skewed inserts in W5.1, however, already incur high contention and the performance gains due to node expansions outweigh the sampling overhead of both approaches. Compared to single-threaded workload adaptation, the shared and the thread-local maps allocate up to 10x more memory (up to 1.5%/2.4% of the index) to reduce sampling-related contention.

5.3 Hybrid Trie

In Figure 19, we consider index size and performance of FST, ART, as well as the trained and the adaptive Hybrid Tries (AHI-Trie) for point lookups and scans on 33M email addresses. While FST stores one character per level, ART nodes inline up to eight *common prefix* characters. This reduces the tree height from 49 to 32 levels and improves performance. For Hybrid Trie, ART stores the upper 9 levels which contain 5.23% of all nodes (≈ 4.28% of the size).

Table 4: Overview of the Lines of Code (LOC) per lookup and insert functions of our two workload adaptive hybrid indexes compared to their non-adaptive counterparts.

Index	Lookup		Insert	
	Logic	Tracking	Logic	Tracking
B+-tree	13	-	100	-
AHI-BTree	15	+1	119	+5
ART/FST	18/46	-	-	-
AHI-Trie (ART/FST)	25/47	+3/+0	-	-

We compare the performance and size of the adaptive and pre-trained Hybrid Trie to ART and FST in Figure 20 for the prefix-random workload W3 and a dataset of 172M 64-bit user ids. We split the workload into two phases and assigned the different prefix ranges (defined by the 44 most significant key bits) randomly to one of the phases. The sampling and adaptation phases for the adaptive trie are highlighted: black dots indicate adaptation phases, while sampling phases take place between two adaptations. The duration of a sampling phase is the product of skip length and sample size. As the sample size does not significantly change in this example (not visualized), the sampling phase duration is mainly determined by the changing skip lengths.

At the beginning of each phase, we observe an increased number of encoding migrations. For phase 1, there are expansions only, as all nodes below c_{ART} are stored in FST. However, during phase 2, nodes frequently accessed in phase 1, are considered to be cold now, and, after a short delay, get compacted again.

After the workload manager identified and expanded/compacted the hot/cold nodes, it increases the skip length to lower sampling-related overhead (cf. Section 3.1.3). This increase results in a larger distance between two consecutive adaptation phases. In contrast, when the workload manager detects an increased number of migrations, it decreases the skip length to allow faster adaptations.

5.4 Code Complexity

To give a rough overview of the required changes and the additional code complexity, we use the metric Lines of Code (LoC) – without considering comments, locks, and empty lines. In Table 4, we denote the lookup and insert functions of the original indexes and compare them to our workload-adaptive variants. We differentiate LoC into the actual logic (e.g., traversing the B+-tree) and the workload-tracking-related code (e.g., adding a sample to the adaptation manager). It can be seen that the tracking-related overhead is limited to at most 3/5 additional lines for lookups/inserts while coping with different encodings adds also extra complexity. An additional function handles the encoding migrations (140 lines for the Hybrid B+-tree, 51/70 lines for expansions/compactions in Hybrid Trie). Subclasses further encapsulate the communication between index and adaptation manager. These consist of 107/88 LoC for Hybrid B+-tree/Hybrid Trie.

6 RELATED WORK

Previous research proposed different strategies to reduce storage overhead and to leverage skew in DBMSs. Back in 2012, Funke et al. introduced an *online* compaction of hybrid in-memory OLTP/OLAP DBMSs based on a hot/cold clustering [21]. In this approach, the access frequencies get tracked at a VM page level. In contrast to this, Levandoski et al. monitor sampled accesses at a record level and write them to a log-file which is evaluated *offline* at a later point in time [33]. Both approaches primarily aim to move cold data to secondary storage devices to free memory capacities.

In 2016, Zhang et al. propose several *compaction rules* to reduce the memory footprints of in-memory DBMSs by reducing the space overhead of index structures such as B+-trees, radix trees, and skip lists [53]. In contrast to previous work, these techniques aim for full in-memory indexing as opposed to migrating cold data to disk: frequently accessed parts get stored using performance-optimized structures, whereas cold data gets compacted, but *remains* in memory. Our experiment in Figure 3 confirms that despite the most recent advances of SSD and NVMe disks, random I/O is still multiple orders of magnitude slower than on-the-fly in-memory decompression. While the introduced compaction rules might create *immutable indexes*, this problem is mitigated by their proposed dual-stage architecture: the dynamic stage contains the deltas created by inserts which are periodically merged into the compacted index.

Contrary, our approach applies different encodings within one single-stage index based on fine-grained access statistics. It does not require index developers to define complicated or expensive merge routines. Nevertheless, implementing different encodings and migration functions might also increase the code complexity. Besides the complexity, our adaptation framework has shown that it can leverage skewed workloads to a higher extent.

Succinct data structures such as FST [55] (which we use in Hybrid Trie) or its alternatives [12, 23, 44] also trade performance for memory efficiency. Succinct [7] and BlowFish [25] are two examples of data systems that use succinct data structures (in this case compressed suffix arrays [24]) for reduced space utilization and improved query performance through fitting more data in memory.

Learned indexes [18, 20, 22, 27, 28, 46, 47] also aim to reduce index size while retraining or even increasing lookup performance over traditional structures. Yet, we argue that the idea of learned indexes is orthogonal to our approach. For example, the Learned Index with Precise Precisions (LIPP) provides tight precision guarantees for all key ranges [51]. Combined with our approach, we could detect hot/cold ranges and increase/lower their precision bounds to reduce LIPP's size without affecting query performance.

7 CONCLUSIONS

We have presented an adaptive workload sampling approach that allows for switching between different node encodings at run-time and applied it to B+-trees and tries. We have shown that it provides significant space benefits without severely impacting performance under skewed workloads while causing negligible overhead under uniform workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was funded by the German Research Foundation (DFG) within the SPP2037 under grant no. Ke 401/22-2.

REFERENCES

[1] AWS EC2 Instances. https://aws.amazon.com/en/ec2/instance-types/high-memory [accessed 2021-03-01].
[2] LevelDB. https://github.com/google/leveldb [accessed 2021-03-01].
[3] RocksDB. https://rocksdb.org/ [accessed 2021-03-01].
[4] S2 Geometry Library. https://s2geometry.io/ [accessed 2021-03-01].
[5] Tape is dead, Disk is tape, Flash is disk, RAM locality is king. http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt [accessed 2021-03-01].
[6] C++ HopscotchMap. https://github.com/Tessil/hopscotch-map [accessed 2021-03-01].
[7] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling Queries on Compressed Data. In *NSDI*. USENIX Association, 337–350.
[8] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*. 26–37.
[9] Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. 2020. The Case for Hybrid Succinct Data Structures. In *EDBT*. 391–394.
[10] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *ACSC*. 97–105.
[11] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *EDBT*. 461–466.
[12] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing Trees of Higher Degree. *Algorithmica* 43, 4 (Nov. 2005), 275–292.
[13] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*. 227–246.
[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX*. 209–223.
[15] Kun-Ta Chuang, Jiun-Long Huang, and Ming-Syan Chen. 2008. Mining top-k frequent patterns in the presence of the memory constraint. *The VLDB Journal* 17, 5 (Aug. 2008), 1321–1344. https://doi.org/10.1007/s00778-007-0078-6
[16] Edith Cohen, Nadav Grossaug, and Haim Kaplan. 2006. Processing Top k Queries from Samples. In *CoNEXT*.
[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154. https://doi.org/10.1145/1807128.1807152
[18] 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 *SIGMOD*. 969–984. https://doi.org/10.1145/3318464.3389711
[19] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. 2020. START—Self-Tuning Adaptive Radix Tree. In *ICDEW*. IEEE, 147–153. https://doi.org/10.1109/ICDEW49219.2020.00015
[20] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135
[21] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *VLDB* 5, 11 (2012). https://doi.org/10.14778/2350229.2350258
[22] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. ACM, 1189–1206. https://doi.org/10.1145/3299869.3319860
[23] Roberto Grossi and Giuseppe Ottaviano. 2014. Fast Compressed Tries through Path Decompositions. *ACM J. Exp. Algorithmics* 19, 1 (2014). https://doi.org/10.1145/2656332
[24] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* 35, 2 (2005), 378–407. https://doi.org/10.1137/S0097539702402354
[25] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2016. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *NSDI*. USENIX Association, 485–500.
[26] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (Dec. 2019). http://arxiv.org/abs/1911.13014
[27] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *aiDM*. 1–5.
[28] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. ACM, 489–504. https://doi.org/10.1145/3183713.3196909
[29] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. ACM, 311–326.
[30] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE, 185–196. https://doi.org/10.1109/ICDE.2018.00026
[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, Vol. 13. 38–49.
[32] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *DaMoN*. 1–8.
[33] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying Hot and Cold Data in Main-Memory Databases. In *ICDE*. IEEE, 26–37.
[34] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *EuroSys*. 1–14.
[35] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*. 183–196.
[36] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13. https://doi.org/10.14778/3421424.3421425
[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*. Springer, 398–412.
[38] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. 2006. Continuous Monitoring of Top-k Queries over Sliding Windows. In *SIGMOD*. 635–646.
[39] Gonzalo Navarro. 2016. *Compact Data Structures - A Practical Approach*. Cambridge University Press.
[40] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
[41] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*.
[42] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*. 61–72.
[43] Andrea Pietracaprina, Matteo Riondato, Eli Upfal, and Fabio Vandin. 2010. Mining top-K frequent itemsets through progressive sampling. *Data Mining and Knowledge Discovery* 21, 2 (2010), 310–326.
[44] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Trans. Algorithms* 3, 4 (2007), 43.
[45] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-Speed Networks. In *ICDE*. IEEE, 1194–1205.
[46] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. *3rd International Workshop on Applied AI for Database Systems and Applications* (2021).
[47] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *3rd International Workshop on Applied AI for Database Systems and Applications* (2021).
[48] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. 1990. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 125–142.
[49] Jeffrey S Vitter. 1985. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.
[50] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.
[51] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *VLDB* 14, 8 (2021), 1276–1288.
[52] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and their Implication on Real World Databases. In *SYSTOR*. 1–11.
[53] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. ACM, 1567–1581.
[54] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *TKDE* 27, 7 (2015), 1920–1948.
[55] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*. 323–336.
[56] Huanchen Zhang, Xiaoxuan Liu, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *SIGMOD*. 1601–1615.