Limousine: Blending Learned and Classical Indexes to Self-Design Larger-than-Memory Cloud Storage Engines

SUBARNA CHATTERJEE, MARK F. PEKALA, LEV KRUGLYAK, and STRATOS IDREOS,
Harvard University, USA

We present Limousine, a self-designing key-value storage engine, that can automatically morph to the near-optimal storage engine architecture shape given a workload, a cloud budget, and a target performance. At its core, Limousine identifies the fundamental design principles of storage engines as combinations of learned and classical data structures that collaborate through algorithms for data storage and access. By unifying these principles over diverse hardware and three major cloud providers (AWS, GCP, and Azure), Limousine creates a massive *design space* of quindecillion (10^{48}) storage engine designs the vast majority of which do not exist in literature or industry. Limousine contains a distribution-aware IO model to accurately evaluate any candidate design. Using these models, Limousine searches within the exhaustive design space to construct a navigable continuum of designs connected along a Pareto frontier of cloud cost and performance. If storage engines contain learned components, Limousine also introduces efficient lazy write algorithms to optimize the holistic read-write performance. Once the near-optimal design is decided for the given context, Limousine automatically materializes the corresponding design in Rust code. Using the YCSB benchmark, we demonstrate that storage engines automatically designed and generated by Limousine scale better by up to 3 orders of magnitude when compared with state-of-the-art industry-leading engines such as RocksDB, WiredTiger, FASTER, and Cosine, over diverse workloads, data sets, and cloud budgets.

CCS Concepts: • **Information systems → storage architectures**.

Additional Key Words and Phrases: self-designing systems, storage engines, key-value stores, big data systems, data layouts

ACM Reference Format:
Subarna Chatterjee, Mark F. Pekala, Lev Kruglyak, and Stratos Idreos. 2024. Limousine: Blending Learned and Classical Indexes to Self-Design Larger-than-Memory Cloud Storage Engines. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 47 (February 2024), 28 pages. https://doi.org/10.1145/3639302

1 REASONING ABOUT SCALABILITY

Continuous Data Growth and Application Diversity. We live in a world where data around us is growing at an exponential rate both in terms of volume and velocity [28, 67, 78, 79]. Alongside data growth, thousands of new data-driven applications are continuously created around diverse fields such as, banking, e-commerce [46], search engines [20, 47, 77], social media [17, 41], science [38, 81], and healthcare [80] that not only want to process and mine these data, but they also need to do so faster than ever.

Scalability: The Ultimate Expectation From a Data System. In the face of massive data volumes and growing application diversity, the scalability requirements from a data system are multifold

Authors' address: Subarna Chatterjee, subarna@seas.harvard.edu; Mark F. Pekala, mpekala@college.harvard.edu; Lev Kruglyak, levkruglyak@college.harvard.edu; Stratos Idreos, stratos@seas.harvard.edu , Harvard University, USA.

ACM 2836-6573/2024/2-ART47
https://doi.org/10.1145/3639302

and more critical than ever. Firstly, we need systems to *scale in terms of performance* i.e., regardless of data growth or application heterogeneity, we expect data systems to not only support faster ingestion of incoming data but also facilitate faster navigation to target data regions for efficient data extraction and query processing. Secondly, as more systems are increasingly deployed on cloud, we need these systems to *scale with respect to cloud cost* i.e., based on the different budget constraints of applications, the systems can still afford enough cloud resources to meet the target performance.

Key-Value Stores: Backbone of Big Data Systems. Key-value storage engines are widely used as the backbone of big data storage within a vast majority of systems [7–11, 21]. With the data growth and application diversity, these engines are deployed on the cloud to leverage auto-scaling of resources (bandwidth, computation, and storage) and handle workload fluctuations.

Problem 1: Fixed Designs of Existing Key-Value Stores Hurt Performance Scalability. While existing storage engines work well for some applications, performance-wise they do not always scale. This is because, state-of-the-art key-value stores rely on fixed and static designs for data storage and navigation. Fundamentally, this is sub-optimal as different applications inherently create drastically different contexts with variability of workloads, data properties, and performance requirements. For instance, Memcached and RocksDB are widely used storage engines with Log Structured Merge (LSM) Tree as the core data structure. While both engines are tailored for write-heavy workloads, they do not guarantee optimality for other workloads [53, 62, 63]. Fig. 1(A) highlights this problem with a mixed workload (R:W 50:50) executed on three widely used industrial (RocksDB [9], WiredTiger [11], FASTER [21]) and one academic (Cosine [22]) key-value stores. For this experiment, we set a fixed hardware with Core i5 processor and 8GB DDR4 RAM and collected the performance numbers for 10M operations. We observe that as data grows with fixed hardware, the throughput of these storage engines can decrease by more than 25× over time. While some reduction is expected as hardware is fixed, the ideal scenario is one where this decline occurs with a smoother gradient, akin to the behavior represented by the blue line. Throughout the paper, we will show how our solution, Limousine achieves this.

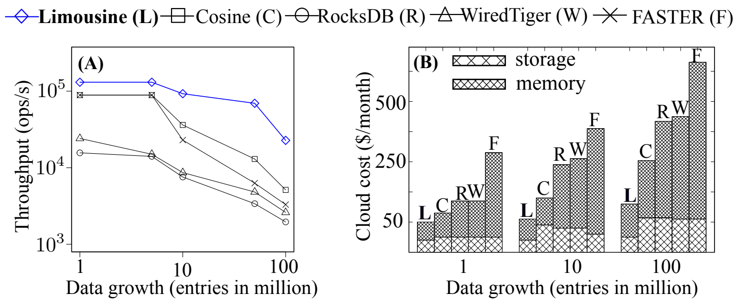
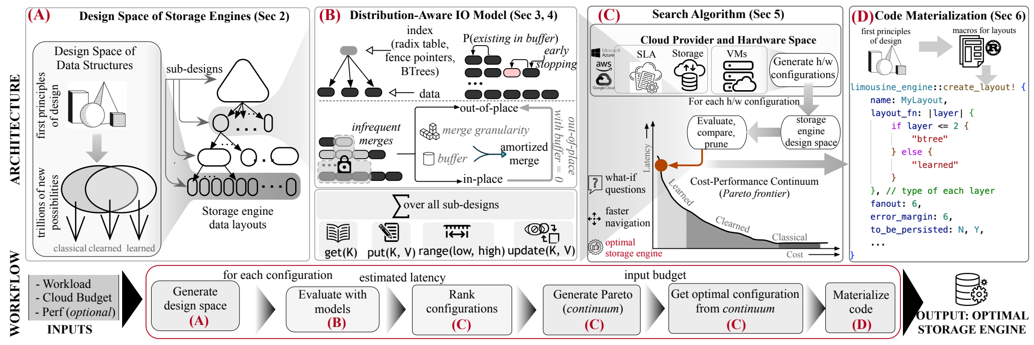


Fig. 1. (A) Existing storage engines fail to scale with data growth. (B) With more data, cloud-cost grows significantly, more than 80% emanating from purchasing high memory.

Problem 2: Linear Growth of Memory Hurts Cost Scalability. The cloud-cost of storage engines grows linearly with data as in-memory navigational structures (such as, filters and indexes) are essentially classical data structures agnostic to underlying data patterns. These structures consume fixed unit of memory for indexing per unit of data and blow up cloud costs for excessive memory usage. For instance, Google reports that in practice, Bigtable (the key-value storage engine used at Google) instances can grow so big that indexes do not fit in-memory and repeated fetching of large index blocks into cache lead to frequent cache misses [12]. Fig. 1(B) highlights this by providing as much hardware necessary to the scenario of 1(A) to prevent the performance drop. We observe this blows up the cloud-cost up to 5× with about 80% of the cost emanating from buying


ARCHITECTURE
(A) Design Space of Storage Engines (Sec 2)
Design Space of Data Structures
first principles of design
trillions of new possibilities
classical cleamed learned
sub-designs
Storage engine data layouts
(B) Distribution-Aware IO Model (Sec 3, 4)
index (radix table, fence pointers, BTrees)
data
P(existing in buffer)
early stopping
infrequent merges
out-of-place
merge granularity
buffer
amortized merge
in-place
out-of-place with buffer = 0
over all sub-designs
get(K) put(K, V) range(low, high) update(K, V)
(C) Search Algorithm (Sec 5)
Cloud Provider and Hardware Space
SLA Storage VMs
Generate h/w configurations
For each h/w configuration
Evaluate, compare, prune
storage engine design space
what-if questions
faster navigation
optimal storage engine
Latency
Learned
Cleamed
Classical
Cost-Performance Continuum (Pareto frontier)
Cost
(D) Code Materialization (Sec 6)
first principles of design
macros for layouts
WORKFLOW
- Workload
- Cloud Budget
- Perf (optional)
INPUTS
for each configuration
estimated latency
input budget
Generate design space (A)
Evaluate with models (B)
Rank configurations (C)
Generate Pareto (continuum) (C)
Get optimal configuration from continuum (C)
Materialize code (D)
OUTPUT: OPTIMAL STORAGE ENGINE

```
limousine_engine::create_layout! {
    name: MyLayout,
    layout_fn: |layer| {
        if layer <= 2 {
            "btree"
        } else {
            "learned"
        }
    }, // type of each layer
    fanout: 6,
    error_margin: 6,
    to_be_persisted: N, Y,
    ...
}
```

Fig. 2. The architecture and workflow of Limousine to design and generate new storage engines.

compute resources or VMs with larger memory. From §2 and onwards, we show how our solution addresses these challenges, ultimately reducing memory expenses to a more economical level.

Key Intuition 1. The design of a storage engine is characterized by combinations of multiple data structures such as filters, buffers, cache, and indexes, that play crucial role in realizing the core key-value operations. Our first key intuition is that, as the space of data structures is ever-growing [48, 52], it naturally creates a massive space of design possibilities for storage engines. Existing systems only support a single engine design each that consist of a single set of specific data structures and algorithms. Naturally, this bounds the performance properties each system can offer and it also bounds the kinds of performance that can be achieved since naturally it is not possible to create literally millions of individual systems manually. Out first intuition here is "what if we can create systems that automatically self-design to any possible system design to achieve the best possible performance?". In other words: "can we know all systems that are possible to design?". Toward this goal, we introduced Cosine [22], the first self-designing key-value storage engine which showed that indeed it is possible to self-design the core of a system. However, Cosine does not scale as shown in Fig. (1) i.e., it works only for scenarios involving small data sets or high cloud budgets. Thus, the overarching question about how can we automatically design and build systems that scale with growing dataset sizes and constrained budgets still remains.

Key Intuition 2. As the footprint of classical data structures grows linearly with data, can we possibly explore other data structures that are more memory-optimized to create better storage engines? Toward this, since 2018 we have witnessed learned data structures [35, 40, 42, 55, 56] the central idea of which is to learn the underlying data patterns and properties to create much smaller, tailored data structures. However, like any other data structure design, learned structures are not perfect: they trade memory footprint and read performance for inferior writes. Consequently, today we still do not have holistic learned key-value storage engines.

The Research Challenge. The research challenge is to (i) appropriately combine the best of learned and classical data structure design principles as core storage engine design decisions to synthesize the optimal storage engine design for any application scenario and (ii) automatically transform any design specification to ready-to-use implementation of the resulting storage engine.

We outline the technical research questions as follows:

(1) Can we know all storage engines that are possible to design?
(2) How can we combine multiple learned and classical structures so that they collaborate to create the perfect storage engine for any application scenario?
(3) How can we deal with the core problem that learned components, by design, are inefficient for writes?

(4) With so many design possibilities, how big is the overall decision space and how can we efficiently search within it?
(5) If we know the perfect storage engine design, can we automatically materialize a ready-to-use implementation?

We address the above technical challenges and take a step toward building self-designing and self-materializing storage engines that scale for diverse data, applications, and cloud budgets.

Our Solution: Limousine. We propose **Limousine**, a self-designing key-value storage engine that automatically instantiates the storage engine architecture to optimize cost-performance tradeoff on the cloud. Limousine takes a workload, a cloud budget, and optionally a target performance as input and outputs a Limousine *configuration* composed of the selected (i) storage engine design in terms of multiple data structures that interact through read-write algorithms, (ii) cloud provider and (iii) storage hardware and VMs for computation.

Limousine comprise of four key components (Fig. 2). First, Limousine creates a design space of quindecillion (10^{48}) storage engines (2A) by identifying the first principles of design for learned and classical data structures, diverse hardware possibilities, and three major cloud providers (AWS, GCP, and Azure). Limousine proposes novel algorithms to optimize read-write performance and introduces IO cost models to estimate the performance of any engine (2B). Limousine efficiently searches the exhaustive space to determine the near-optimal storage engine design (2C) and automatically materializes the code for the target engine (2D).

Our contributions are as follows.

(1) We show that existing learned (PGM, FITing-Tree, RadixSpline) and classical (BTree, LSM-tree, LSH-table) data structures share the same first principles of design. We identify two core properties that fundamentally differentiate learned structures from classical structures and show how a unified design space connects the two data structure paradigms (§2.2).
(2) We introduce storage engines with a new family of data structures, called *clearned structures*, created by blending design principles of learned and classical structures. We show storage engines with clearned structures are made up of diverse design possibilities leading to heterogeneous data layouts or *sub-designs* for different levels of the storage hierarchy (§2.3).
(3) We introduce a design space of quindecillion (10^{48}) storage engines comprising of (a) combinations of learned (PGM, FITing-Tree, RadixSpline), classical (LSM-trees, B-trees, LSH-tables) and clearned structures, (b) in-memory accelerators (buffers, filters, and indexes), (c) hardware and (d) cloud providers (§2.4).
(4) We empirically identify the source of inefficient writes within learned data structures and introduce a crammed out-of-place (COOP) write algorithm that increases storage engine performance by up to 70%, compared to the state-of-the-art (§3).
(5) We introduce a distribution-aware IO-model to evaluate storage engine designs accurately. We show how the model embeds data layout primitives, workloads, and data distribution within its estimation process to achieve an average accuracy of 90% over diverse designs, hardware, and application contexts (§4).
(6) We show how Limousine transforms the entire design space to a low-dimensional Pareto frontier of ranked storage engines to determine the near-optimal design (§6).
(7) We show how Limousine automatically generates the code for the target storage engine in Rust (§7).

Symbol	Explanation	Symbol	Explanation
B	#entries in a block	M_{B_i}	Memory allocated to buffers of L_i
E	entry size	M_{FP}	Memory allocated to index of L_i
N	total data entries	M_{BF}	Memory allocated to filters of L_i
T_i	size ratio at level i	G	Merge granularity
K_i	merge greediness (hot)	ϵ	Error bounds of models
Z_i	merge greediness (cold)	$\mathcal{D}_{\text{get}}$	Distribution over reads
D_i	Data at level i	$\mathcal{D}_{\text{put}}$	Distribution over writes
Ω_i	Design at level i	W	Workload

Table 1. List of notations used in the paper

(8) Using the YCSB benchmark, we demonstrate that on average, Limousine outperforms existing storage engines used in industry (RocksDB, WiredTiger, and FASTER) and academia (Cosine) by up to 3 orders of magnitude over diverse workloads, data sets, and cloud budgets (§8).

2 THE DESIGN SPACE OF LIMOUSINE

The core of storage engines is *data layouts* that control how data is physically organized and stored crucially affecting the end-to-end performance. A data layout is made up of several data structures each with a specific balance of the fundamental trade-offs of read, update, and memory amplification [18, 48]. Consequently, there is no single data structure and no single storage engine design that covers diverse performance requirements. Therefore, to realize Limousine's end goal of self-designing and materializing near-optimal storage engine architecture for different applications, it is crucial to explore the "design space" of data layouts. Effectively, such a design space of layouts controls the design space of storage engines by creating diverse architectural possibilities to cater to the needs of emerging and ever-changing data-driven applications.

In this section, we introduce the existing design space of all classical data structures that are used within modern key-value stores (§2.1). Next, we investigate the different learned layouts possible within storage engines (§2.2) and contrast the two seemingly different data structure paradigms. We show why it is beneficial to consolidate them as part of a single, unified design space (§2.2) that leads to the synthesis of a novel class of data layouts, namely *clearned layouts* within key-value engines (§2.3). Finally, we show how Limousine unifies all afore-mentioned data layout variants (pure classical, pure learned, and clearned) into a single exhaustive design space to create quindecillion (10^{48}) possibilities of storage engine designs (§2.4) to match the needs of diverse applications.

2.1 Background: Classical Data Structures

The Design Space for Classical Data Structures. The problem of knowing all possible data layouts within storage engines is open and perhaps even unsolvable [50, 51]. Toward this, the Data Calculator [52] introduced the concept of a design space for fine-grained design decisions and learned cost models to help with evaluating designs.

The Design Continuum for Classical Structures. Although calculator conceptualizes the design space of classical data layouts, it is intractable to navigate through this space to find optimal storage engine design as the cardinality of the space is exponential to the number of design principles. To this end, existing research proposes *design continuums* that use a small set of primitives to connect seemingly different data structures along a continuous performance hyperplane or a Pareto frontier (explained in Section 9). This enables efficient navigation through parts of the design space and opens up possibilities for self-designing and adaptive systems. In 2019, we see the first design continuum of modern key-value stores spanning across classical data structures – LSM-Trees, LSH-tables, and BTrees [48].

2.2 Learned Data Structure Designs

The Unexplored Design Space of Learned Data Structures. To construct the design space of storage engines with learned data structures, it is once again crucial to identify the fundamental primitives that capture the first principles of designing them. Further, to be able to use learned data layouts, we should have ways to efficiently navigate through this design space and evaluate their performance. Toward this, we first set out to contrast the design concepts for learned and classical structures. We use Btrees as the representative classical data structure for contrasting purposes as existing work [35, 40, 56] already establishes a link about how BTrees can be fundamentally seen as index structures with models which are also the inherent core of learned data structures. Table 1 lists the notation used in this work.

The Structural Similarity in BTrees and Learned Indexes. There is a significant similarity in how BTrees and learned indexes store and access data. Both these structures are hierarchical in nature with the last level containing the base data and the other upper levels containing indexing information. Within a level, the indexing information is stored within *nodes* containing *elements* in a key-value format i.e., the value of an element is a pointer for BTrees and the combination of models and pointers for learned indexes. Further, for both these structures, data within a level is always sorted and every element (i.e., a model for learned index or a key-pointer for BTree) of a node indexes non-overlapping data regions. For accessing base data, both these structures read once per level to navigate through the hierarchy and reach the base data.

Difference 1: Orders of Magnitude of Size Ratios. A crucial design principle in a BTree structure and a learned index is the fanout T indicating the branching factor of each node in a level and in turn, regulates the layout of the storage engine [22, 48]. In a BTree, T is in the order of the disk block size or $\Theta(B)$ so that data is read at the granularity of disk pages to optimize the cost of IOs. However, theoretically, T is bounded by the maximum data to be indexed, i.e., $O(N)$ and can fall anywhere within the range of $[2, N]$. For any Btree variants (pure Btrees [25], B^{ϵ}-Trees [19, 54], FD-Trees [59], CSB-Trees [66]), T never exceeds $\Theta(B)$ to higher order of block sizes ($O(B^2)$, $O(B^3)$, and so on) as these structures are storage-based indexing techniques that always store block-specific metadata information in the form of pointers, or offsets, or start key of a block. On the other hand, compute-based indexing techniques such as learned indexes can index larger chunks of data through learned models. This allows learned indexes to tap into higher size ratios. For instance, Ferragina *et al.* [39] proved that linear models can be as powerful as indexing data that quadratically scales with block size i.e., a single linear model can index a data chunk of size $O(B^2)$. Therefore, as the model complexity increases to higher order polynomials, T of a learned index is expected to gradually grow further till it reaches the extremity where a single model can index the whole data set. This brings us to the first design property unique to a learned data layout:

Property 1: *The size ratio T_i of any level i of a learned layout is typically higher than that of a classical structure with the maximum being the amount of information that it indexes (N) and the minimum being $\Theta(B)$ when it converges to a classical Btree.*

Difference 2: Variability of Size Ratios Across Levels. There is another major distinction in the variability of T across levels of storage. In Btrees, every level always grows by a factor of T which

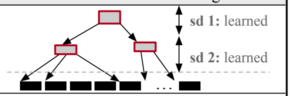
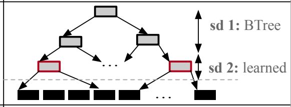
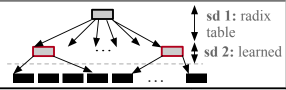
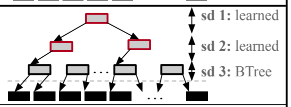
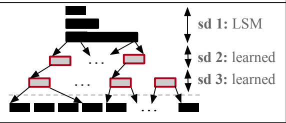
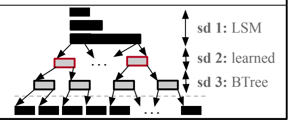
	Cleared layouts	Layout primitives and their values					Storage engine layouts in Limousine (▭ classical, 🟥 learned, ■ data node, **sd**: sub-design)
		M_{FP} **(index)**	M_B **(buffer)**	M_{BF} **(filter)**	**L1**	**L2 (and below)**	
Existing layouts in literature	**2-level PGM**	func() Rule: *if L1/L2 fits in memory*	0 (in-place write)		$T_1 = kB^2$ (*k based on N*) K1 = 1, Z1 = 1	$T_2 = kB^2$ (*k based on D1*) K2 = 1, Z2 = 1	
Existing layouts in literature	**2-level FIT-ing Tree**	func() Rule: *if L1/L2 fits in memory*	0 (in-place write)		$T_1 = kB^2$ (*k based on N*) K1 = 1, Z1 = 1	$T_2 = B$ K2 = 1, Z2 = 1	
Existing layouts in literature	**Radix Spline**	r bits (*default 18*) + func() Rule: *if L1 fits in memory*			$T_1 = kB^2$ (*k based on N*) K1 = 1, Z1 = 1		
New layouts in Limousine (quindecillion possibilities)	**New cleared layout (C1)**	func() Rule: *if L1/L2 fits in memory*	[0, 64] entries for each node *(in-place or out-of-place)*		$T_1 = kB^2$ (*k based on N*) K1 = 1, Z1 = 1	$T_2 = kB^2$ (*k based on D1*) or $T_2 = B$ K2 = 1, Z2 = 1	
New layouts in Limousine (quindecillion possibilities)	**New cleared layout (C2)**	Fence pointers of LSM + func() Rule: (*if L2 fit in memory*)	[64 MB, 128 MB, …] for LSM + func() *(in-place or out-of-place)*	10 bits/ key	T1 = [2, 32], K1 = [1, T1], Z1 = [1, T1]	$T_2 = k_2B^2$ $T_3 = k_3B^2$ K2 = 1, Z2 = 1 K3 = 1, Z3 = 1	
New layouts in Limousine (quindecillion possibilities)	**New cleared layout (C3)**	Fence pointers of LSM + func() Rule: (*if L2 fit in memory*)	[64 MB, 128 MB, …] for LSM + func() *(in-place or out-of-place)*	10 bits/ key	T1 = [2, 32], K1 = [1, T1], Z1 = [1, T1]	$T_2 = k_2B^2$ $T_3 = B$ K2 = 1, Z2 = 1 K3 = 1, Z3 = 1	

Table 2. Clearned layout instantiations in Limousine

is a constant for the entire storage engine hierarchy. On the other hand, for learned layouts, this is untrue. Levels of learned layouts can be recursively constructed thereby indexing different size and distribution of data at every level. This fundamentally brings in the new behavior of having variable growth factor for every level depending on what data they index. For instance, in a k (e.g., 2)-level storage engine with learned layouts indexing N (e.g., 1M) integers with model size being s (e.g., 20 bytes for linear models) and p bytes for pointers (e.g., 4 bytes), the last level of the storage engine may contain $m1$ (e.g., 100) models and the second last level may contain $m2$ (e.g., 2) models to index the $m1$ (100) models below it. Thus, the size ratios for the last and the second last levels are respectively $\frac{N}{m1\times(s+p)}$ $\left(\frac{1M}{100\times(16+4)} = 500\right)$ and $\frac{m1}{m2\times(s+p)}$ $\left(\frac{100}{2\times(16+4)} = 2.5\right)$.

Property 2: *Unlike classical storage engines, every level L_i of learned storage engines has its own size ratio T_i (subjected to the indexing capabilities at L_i) and hence, each level may have a different data layout.*

Unifying the Design Space of Classical and Learned Data Structures. After contrasting the two paradigms, we observe that although there are two fundamental properties that distinctly differentiate learned layouts, a large part of the design principles still overlap. Therefore, if we can unify the design principles of classical and learned layouts into a single design space, it automatically opens up a diverse space of possibilities and design opportunities.

2.3 Introducing Clearned Data Layouts

Implications of Property 1. Toward unifying the design space of classical and learned layouts, the implications of Property 1 is to extend the domain of the existing primitive, T (size ratio) from

$[2, O(B)]$ to $[2, O(N)]$. This way the resulting design continuum [48] can accommodate more possibilities of size ratios to be combined with the other primitives to generate new data layouts.

Implications of Property 2. Property 2 inherently introduces the possibility of designing every level of a storage engine differently by setting different values of T at each level of a data layout. This naturally brings up several new design questions such as,

(1) How can we know the best layout at each storage engine level?
(2) Can we possibly create and combine data layouts of each level such that the resulting storage engine design can be any permutation of learned and classical components?
(3) Can we translate the impact of any permutation of design decisions to end goals such as performance or cloud cost?

As we set out to answer the above questions, we discover that this leads to the instantiation of a family of new data layouts within storage engines that are not only different in their primitive values but more importantly, by virtue of their construction process.

A New Class of Data Layouts: Clearned Layouts. We introduce *clearned data layouts* by blending design principles of classical and learned layouts (Table 2). To describe a clearned layout, we introduce *sub-designs* or data layouts corresponding to a subspace of the storage engine hierarchy. A sub-design is *learned* if the corresponding storage engine levels use learned models to index information below it, otherwise, we call it a *classical sub-design.* A *clearned storage engine* is a combination of one or more sub-designs covering non-overlapping parts of the overall storage hierarchy with at least one of the sub-designs being learned. Therefore, by definition, all storage engines with pure classical data layouts are not clearned systems as they do not contain learned components, however, any engine with pure learned layouts qualifies as a clearned system.

Examples of Clearned Data Layouts. The right-most column of Table 2 shows examples of existing and new clearned layouts, and in §2.4, we discuss the configuration details and definitions of clearned sub-designs. A simple example of a *clearned layout* is the existing PGM [40] index comprising of L levels. Each level can contain any number of learned models affecting the T and data layout of that level, thereby leading to numerous distinct *sub-designs* in the overall structure. Another example of an existing clearned structure is a FIT-ing tree [42] comprising of two sub-designs: the first is a classical BTree that can span across multiple levels and the second is the last index layer comprising of learned models.

In addition to the few existing structures, we permute and combine different classical and learned components along the storage engine hierarchy to synthesize innumerable possibilities of clearned layouts that we have not seen in the literature – C1, C2, and C3 in Table 2 are three such new data layouts. C1 is a 3-level clearned structure comprising of two learned sub-designs at the top followed by a BTree layer at the bottom. C2 and C3 are more complex clearned examples beyond the realm of tree-based indexing with LSM sub-designs with implicit indexing on top [68, 70–72] and combinations of explicit indexes with learned and BTree sub-designs at the bottom. Later, in §8.3, we show and discuss how storage engines with these new layouts can outperform existing storage engines e.g., storage engines with layout C1 can optimize write-performance in a read-intensive workload and those with C2 and C3 can optimize the cost-performance tradeoff for low-budget, mixed workloads.

2.4 Data Layouts of Limousine

We now introduce the superstructure (Fig. 3) of Limousine that contains all design primitives necessary to create structured definitions of arbitrary data layouts and thus, arbitrary key-value storage engines that fall in this design space. This design space comprise of storage engines with

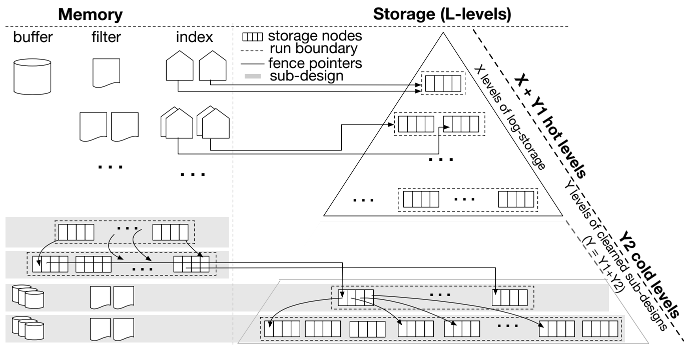


Fig. 3. Superstructure of data layouts in Limousine

not only all major classical (BTrees, LSM-Trees, LSH-Tables) and learned (FIT-ing Trees, PGM, and Radix Spline) layouts but also several new clearned layouts that are uniquely distinctive in how they blend learned and classical concepts as core design elements within modern key-value stores.

Data Layout Primitives. We use the same set of data layout primitives [22, 48] that include the size ratio (T) with expanded domain, merge thresholds (K and Z) along with in-memory accelerators such as buffers (M_B), filters (M_{BF}), and indexes (M_{FP}) (Table 1).

Design Rules within Limousine's Superstructure. The L-level superstructure (Fig. 3) is conceptually designed to contain a top sub-design utilizing log-structured storage (LSM-trees, LSH-tables, hybrids) over X levels and a bottom component using explicit indexing across Y sub-designs of learned structures and Btrees. This hierarchy facilitates the realization of over quindecillion (10^{48}) storage engine layouts such as (i) pure log-based data layouts (for write-heavy workloads), (ii) clearned layouts excluding log-structured sub-designs (read-heavy workloads), and (iii) other combinations of classical and learned structures (mixed workloads). Additionally, the superstructure allows data classification into hot and cold categories. The upper X levels remain hot owing to the use of filters and indexes by classical log-based designs. The dominant primitives in this part are T, K, and the memory allocation across these levels. Within the remaining Y levels, each may be configured as its own sub-design i.e., we may set the combination of the primitives T, K, Z, M_B, M_{BF}, and M_{FP} differently at every level. As learned models can be significantly more succinct compared to classical Btrees, depending on the available memory budget and the model size, it might be possible to cache a part of Y levels, say $Y1$, in-memory. The rest of the $Y2 = Y - Y1$ levels can be accessed through cascading fence pointers from the in-memory levels. There are different possibilities about how to partition data between hot and cold levels which control the exact values of X, $Y1$, and $Y2$. In the subsequent sections, we show given any workload, how Limousine can automatically set these values appropriately using its internal IO models (§4, §6).

Describing Existing and New Storage Engine Layouts. With the Limousine's superstructure, the definitions of classical storage engines remain the same as before [22]. For clearned storage engines, we define them in Table 2. For example, a storage engine with a PGM sub-design i and linear models can be described by $T_i = kB^2$, k being a data-specific constant [39]. M_{FP} and M_B are set based on available memory and the write strategy (§3) and K and Z are set to 1 for sorted data. Table 2 includes additional definitions of storage engine designs with existing and new layouts.

3 CLEARNED WRITES: NO MORE ACHILLES HEEL

In this section, we address writes, one of the fundamental limitations of learned structures [39, 40, 42, 55]. We begin by analyzing and identifying the root cause of the problem on pure learned structures. Next, we propose a new write algorithm that significantly enhances write performance of learned structures without degrading the read performance. Finally, we show how the proposed algorithm can be extended beyond pure learned structures and can be generalized for the entire space of clearned engines.

3.1 In-Place Writes on Learned Structures

The most common technique to insert data on a learned structure is to write in place (IP) [40, 42] similar to that of a Btree. We investigate the overhead of IP writes through empirical analysis.

Problem 1: Enormous Data Movement. IP insert incurs massive data movement. First, each insert may lead to moving several (all, in the worst-case) entries within a segment to preserve the sorted order across each level. Second, an insert may break the ϵ guarantee of a model thereby, necessitating to split the affected segment (or model) and train multiple small models to preserve ϵ. This, in turn, may trigger series of splits or merges up the storage hierarchy. To demonstrate this, we experiment (Fig. 4A) by varying number of inserts on the existing PGM index build on top of 1M uniformly distributed entries, each of size 64 bytes and record the number of splits because of the afore-mentioned reasons. We observe that the probability of a split followed by a subsequent merge is significantly high averaging at 0.9. This incurs data movement (Fig. 4(B)), that in the worst-case can be 1000× more than the data being inserted.

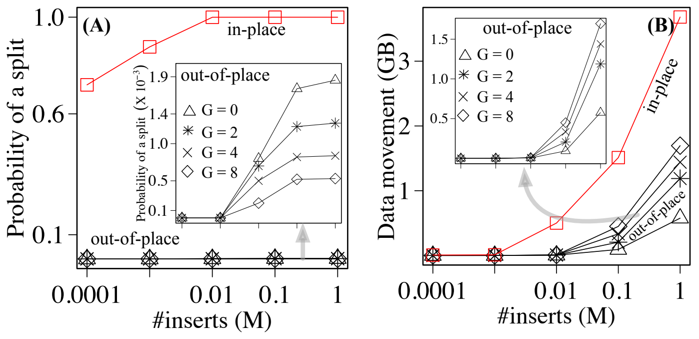


Fig. 4. With IP inserts, the probability of splitting a learned model can be as high as 0.9 incurring up to 1000× data movement for merging and reorganization of the data structure.

Problem 2: Eventual Degradation of Tree Structure. Even after a small number of IP inserts, the model indexability significantly decreases. From our previous experiment, we show the number of keys indexed per model as 1M inserts progress over time (Fig. 5). After only 20% of inserts, the indexability drops by 5× and by the time all inserts finish, the average indexability reduces to that of a BTree. This, in turn, defeats the purpose of learned structures by (a) increasing the number of levels, and its overall size, and more crucially (b) degrading read performance due to more number of accesses to reach base data. To obtain the BTree baseline, we constructed a BTree with fanout 128 on top of the base data and measured the size of each node.

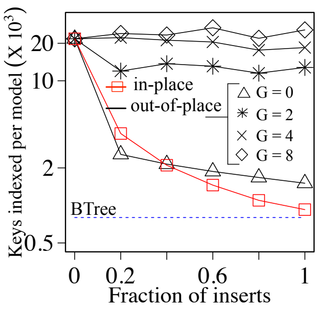


Fig. 5. IP writes reduce model indexability to Btrees.

3.2 Out-of-Place Writes on Learned Structures

We propose **COOP**, a crammed out-of-place write algorithm that enhances performance of learned indexes by (i) facilitating faster inserts, (ii) without hurting the read performance or the index structure, and (iii) without introducing a massive memory overhead.

Our Core Idea. There are two core ideas COOP. Firstly, for every node j in level i, we keep an in-memory buffer $M_{B_{ij}}$ to accept incoming inserts. However, we carefully configure the buffer capacity based on the data and the workload (explained below). We use this subscript notation to differentiate

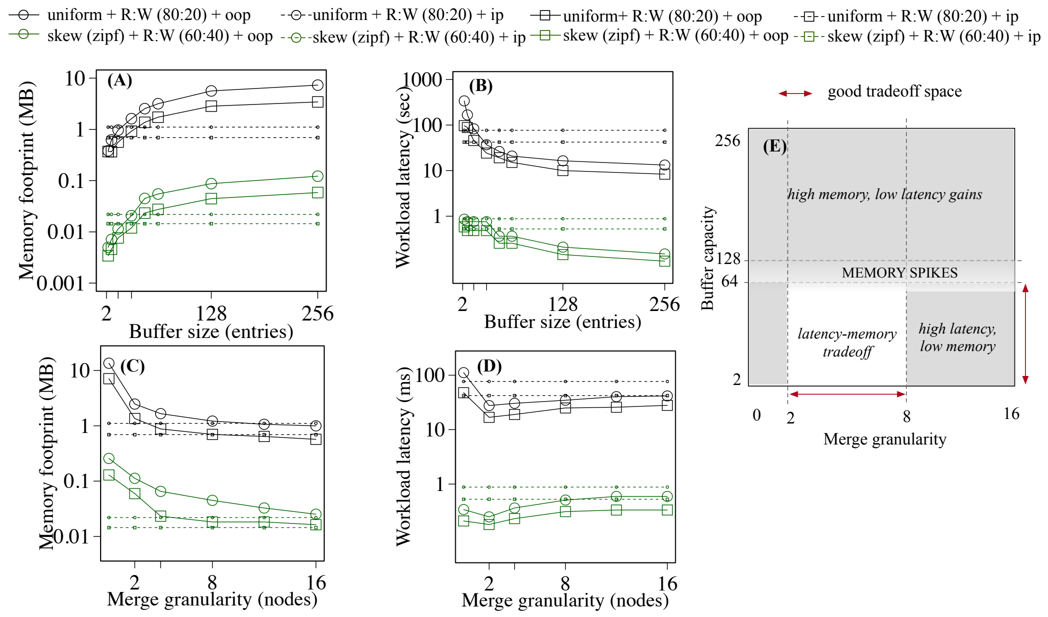


Fig. 6. Increasing buffer capacity improves performance up to a point beyond which it leads to diminishing returns at the cost of high memory footprint (A, B). Although higher merge granularity reduces memory footprint, it can bring the overall latency down only till a threshold beyond which the merge overhead surpasses the benefits (C, D). Out of all possibilities of $M_{B_{ij}}$ and G, only the intersection of $M_{B_{ij}} \leq 64$ and $2 \leq G \leq 8$ offer good tradeoff properties between performance and memory (E).

node buffers ($M_{B_{ij}}$) from the level buffer (M_{B_i}) introduced earlier in §2 as a layout primitive. The footprint of node buffers adds up to the buffer capacity of a level, i.e., $M_{B_i} = \sum_{\text{all nodes at level } i} M_{B_{ij}}$. When a node buffer reaches capacity, we merge the buffer with the node and propagate the changes up the hierarchy. Secondly, we introduce another design consideration in COOP called merge granularity G, that captures the number of neighboring segments or models to be merged together during a split. This parameter enables learned structures to leverage their inherent weakness into strength by leveraging incoming inserts to reunite split models and discover new easily-approximated patterns thereby maintaining the initial tree structure.

Investigating the Tradeoff for Buffer Capacity and Merge Granularity. Theoretically, $M_{B_{ij}}$ can take up any non-negative value. While increasing $M_{B_{ij}}$ can speed up inserts by accommodating more entries, it can significantly increase the memory footprint. Further, it also slows down reads as data can stay both within buffers and the nodes. For merge granularity G, the minimum possible value is 0 (involving no neighbors) and the maximum can be the fanout of the previous level indicating a maximum number of nodes in the current level. While higher merge granularity can amortize the insert cost, it may also increase the total data movement and memory used for merges. Therefore, it is important to explore the tradeoff space of both parameters to understand their implications on performance and memory.

Empirical Study of $M_{B_{ij}}$ and G. We use two data sets with uniform and skewed (Zipfian) distribution on 10M entries and two workloads with read-write ratios being 80:20 and 60:40. We use both COOP and IP for comparison and measure the memory footprint and performance (Fig. 6). We first analyze the tradeoff space of the in-memory buffers. As buffer size increases, memory footprint grows (A), and latency reduces (B). However, beyond the size of 64, the memory usage steeply rises, but the drop in latency becomes negligible (less than 2%) suggesting diminishing returns for both the dataset and the workloads. As G increases, the memory footprint reduces owing to fewer resulting models and hence, a smaller number of node buffers (Fig. 6C). This drop in footprint is more than 10% till $G = 8$ beyond which the change becomes negligible (less than 1%) as the working memory also increases owing to larger merges. On the other hand, for all data-workload combinations, as we increase G, the latency of workload steeply falls till $G = 2$ after which it rises but marginally and between $G = 8$ and $G = 16$ the slope tends to become zero (D). This is because, after a point, the cost of a single merge starts showing up as an overhead in the overall performance instead of a benefit. Thus, we infer that $2 \leq G \leq 8$ are potential values that could provide noticeable tradeoff properties between memory and performance for any workload.

Pruning Suboptimal Values of $M_{B_{ij}}$ and G. We observe that uniformly distributed datasets exhibit minimal pruning of COOP parameters and hence result in a wider effective design space of storage engines with $M_{B_{ij}} \leq 64$, $2 \leq G \leq 8$. With skewed datasets, workloads tend to get localized to specific segments of the data thereby pruning the design space drastically with qualified values being $M_{B_{ij}} = 32$, $G = \{2, 4\}$, while still operating within the broader superset selected from the uniform data. This demonstrates the generalizability of these insights across a wider spectrum of data distribution and workloads. The remaining parameter space is either sub-optimal or with diminishing returns and therefore, we prune them from Limousine's design space.

3.3 COOP Writes for Clearned Structures

Learned and BTree Sub-Designs. For learned sub-designs, integrating COOP writes is straightforward as nodes of such a sub-design comprise of either multiple models pointing to data in the next level or the base data itself. Each node contains its own buffer to accept inserts such that the addition of new models or data is periodically reflected in the main structure after the buffer is merged. Similarly, including COOP writes for BTree sub-designs directly follows as in principle, a Btree layer is similar to that of learned sub-design with just a different indexability and error bound.

Other Classical Sub-designs. Other classical sub-designs may also appear in a clearned structure, such as an LSM sub-design or an LSH sub-design. For LSMs, the inherent write policy is already out-of-place as the data is fundamentally stored in an immutable format. On the other hand, LSH sub-designs follow a mix of IP updates (leveraging the hash table indexability) and out-of-place inserts. Therefore, inserts within LSH sub-designs also remain unaffected.

4 IO MODELS IN LIMOUSINE

We now describe how Limousine estimates the IO cost of any data layout. For classical layouts, Limousine relies on existing distribution-aware IO model [22] while for the clearned storage engines, we propose new models. Given a storage engine design specification, Limousine can differentiate designs based on the values of the layout primitives and invoke the right model accordingly.

Workload Characterization and Key Distribution. In Limousine, any workload W is expressed using (a) w number of operations over a universe of key-value pairs, (b) the proportion θ_{op} of each operation type: single-result lookups, inserts, blind updates (updating the value of an entry regardless of its current value), read-modify-writes (rmws) (updating the value of an entry based on the current value) range queries and deletes, and (c) the distribution of the keys and workload operations. This forms a workload feature vector.

It is crucial to express workload and data distributions in a consistent and standardized manner such that the resulting workload feature vector can be applied to both the existing cost models of classical storage engines and the new models of clearned engines. To this end, we follow the same distribution parameters as Cosine [22]: $\mathcal{D}_{get}$ and $\mathcal{D}_{put}$ denote uniform distribution over the universe U of keys for lookups and inserts, respectively. Further, skew is defined as a combination of *regular* keys and *special* keys (likely to be accessed more than regular keys) over two distinct universe of U_2 and U_1, respectively. p_{get} and p_{put} denote the respective probabilities of drawing a special key for lookups and inserts (from U_1) while $1 - p_{get}$ and $1 - p_{put}$ denote the counter-part probabilities for regular keys (from U_2).

Overview. Let Ω be an L-level clearned storage engine comprising of k sub-designs. We start with the case when a sub-design is comprised of either learned models or storage-based indexes, such as BTrees. Note that a Btree sub-component can be either thought of as a single sub-design or an ensemble of multiple sub-designs (as many as the number of consecutive BTree layers) each with identical specifications of the data layout primitives. We will use the latter conceptualization to simplify the model intuition with the assumption that each sub-design only affects a single-level of the structure and in this case, $k = L$. The total IO cost of any clearned storage engine can be estimated as the summation of the individual IO cost emanating from each of the component sub-designs $\Omega_i \in \Omega$. Therefore, for running any workload W on D data entries stored with Ω data layout, we estimate the performance $\sigma_W(\Omega, D)$ as:

$$\sigma_W(\Omega, D) = \sum_{\Omega_i \in \Omega} \sum_{op \in W} \theta_{op} . \sigma_{op}(\Omega_i, D_i) \tag{1}$$

where σ_{op} denotes the per-operation IO cost $\forall op \in W$. A sub-design can only index data residing below it and hence, we denote the data indexed by Ω_i as D_{i+1}. Similarly, the data held by Ω_i becomes indexable to the immediate sub-design above i.e., Ω_{i-1}. At any level i, the minimum total size of data to be indexed can be derived from the layout primitives as $D_i = \frac{q \times D_{i+1}}{T_i}$ where q is the fill-factor of node ($0 \leq q \leq 1$). Values of q less than 1 accommodate several design variants such as BTrees with partially full nodes [26, 43] and designs with gapped arrays [35]. Despite the inherent complexity due to the strategic placement of gaps in gapped array designs, Limousine can still approximate the impact on writes through discounted data movement (by a factor of q) by amortizing it over a large number of writes (Table 3). For reads, the model stays the same but the IO cost changes as q affects the height of the tree. For the rest of the section, we discuss the process of estimating the IO cost of any sub-design which is the building block for Eq. 1.

Intuition for Modeling Reads. For both lookups and range queries (σ_{get}, σ_{range}), regardless of other layout parameters, one access per sub-design is always necessary to reach the base data. Further, as mentioned in §3, a sub-design Ω_i of a clearned storage engine may contain in-memory buffers $M_{B_{i+1}}$ the value of which depends on whether the write-policy of Ω_i is IP ($M_{B_i} = 0$) or COOP ($M_{B_i} > 0$). Generalizing this layout specification, we model the cost of a read (σ_{get}) as shown in Table 3. We use the index memory M_{FP_i} of Ω_i to clearly indicate if the sub-design is in-memory or on-disk, i.e., Ω_i is in-memory is $M_{FP_i} \neq 0$. Although we do not explicitly consider the effects of caching, by taking into account portions of data that fit in-memory, we prevent over-estimation of IOs and accurately estimate the cost for a very large number of queries (§8.1).

Intuition for Modeling In-Place Writes. We model the cost of IP inserts and use it as a building block to model COOP writes. When modeling IP writes, it is crucial to consider the data movement involved with restructuring the layout e.g., split and merge for BTree sub-designs and cost of retraining models for learned sub-designs. As we restrict the scope of calculations only at per-level basis, we assume that the insert cost at a given node only affects its parent, however, not every single insert triggers merge. To model this behaviour, we first estimate the average indexing granularity of a sub-design i.e., the average number of entries indexed by a single element (such as key-pointer

Operation type, Proportion	IO Cost Model
get/lookup, θ_{get}	$\sigma_{get}(\Omega_i, D_i) = \begin{cases} 0 & \text{if } M_{FP_i} \neq 0 \\ (1 - P(\text{buffer}_i)) \times P(\text{base}_i) & \text{otherwise} \end{cases}$
put/insert, θ_{put}	$\sigma_{put}(\Omega_i, D_i) = \sigma_{get}(\Omega_i, D_i) + q.\frac{D_{i+1}}{T_i \times B}.\frac{G^{\alpha}}{\min(1, \lvert M_{B_i} \rvert)}$
rmw, θ_{rmw}	$\sigma_{rmw}(\Omega_i, D_i) = \sigma_{get}(\Omega_i, D_i) + \sigma_{put}(\Omega_i, D_i)$
range, θ_{range}	$\sigma_{range}(\Omega_i, D_i) = \begin{cases} \sigma_{get} & \text{for indexing layers} \\ \frac{\lvert\text{high}-\text{low}\rvert}{\lvert U \rvert \times B} & \text{for data layer, uniform distribution} \\ \frac{\lvert\text{high}-\text{low}\rvert}{B}\left(\frac{p_{get}}{\lvert U_1 \rvert} + \frac{1-p_{get}}{U_2}\right) & \text{for data layer, skew} \end{cases}$
delete, θ_{delete}	$\sigma_{delete}(\Omega_i, D_i) = \begin{cases} \sigma_{get}(\Omega_i, D_i) + q.\frac{D_{i+1}}{T_i \times B}, \text{in-place} \\ \sigma_{rmw}(\Omega_i, D_i) + \frac{D_{i+1}}{T_i \times B}.\frac{G^{\alpha}}{\min(1, \lvert M_{B_i} \rvert)}, \text{out-of-place} \end{cases}$
building blocks of the model	$P(\text{buffer}_i) = \begin{cases} \frac{\lvert M_{B_i} \rvert}{\lvert U \rvert} & \text{for uniform distribution} \\ \lvert M_{B_i} \rvert \times \left(\frac{p_{get}}{\lvert U_1 \rvert} + \frac{1-p_{get}}{\lvert U_2 \rvert}\right) & \text{for skew} \end{cases}$ $P(\text{base}_i) = \begin{cases} \frac{\lvert D_i \rvert}{\lvert U \rvert} & \text{for uniform distribution} \\ \lvert D_i \rvert \times \left(\frac{p_{get}}{\lvert U_1 \rvert} + \frac{1-p_{get}}{\lvert U_2 \rvert}\right) & \text{for skew} \end{cases}$

Table 3. IO models in Limousine

pair for a Btree and a model for learned structure) of a sub-design. Within a level, any BTree key-pointer pair indexes $O(B)$ amount of data whereas the indexability of a linear model can typically range between $O(B)$ to $O(k.B^2)$ where k is a distribution-specific constant [39]. This is essentially captured by primitive T_i. However, every single insert may not trigger a merge and inserts may be distributed to affect different indexing elements within a level. We approximate the effect of this behavior with the key insight that, in any sub-design, if there are $\mathcal{I}$ elements with T_i being the average size of data indexed by each element, it would approximately take another T_i inserts to have a total of $\mathcal{I} + 1$ elements in that level. If we take the example of a piecewise-linear model on uniform data (a and b are minimum and maximum of the distribution), the average number of keys indexed by the model is $3\frac{(a+b)^2}{(b-a)^2}B^2$ if the keys are drawn i.i.d [39]. Therefore approximately after every T_i inserts, there will be a data movement of $\frac{3\frac{(a+b)^2}{(b-a)^2}B^2}{B}$ (to generalize D_{i+1}/B) IOs. As we amortize this cost over T_i inserts, we get $\frac{D_{i+1}}{T_i \times B}$ IO per merge operation. The impact of cascading merges is automatically accounted for as we cumulate (Equation 1) the cost of all sub-designs within a storage engine.

Intuition for Modeling Out-of-Place Writes. We model COOP writes as merge operations similar to that of an IP insert but periodically triggered once the buffer reaches capacity i.e., after every $|M_{B_i}|$ inserts. Further, the cost of merge is regulated by the merge granularity G (§3) with α denoting the factor by which G is multiplicative on performance. In principle, an IP strategy is essentially an out-of-place algorithm in absence of a buffer or to ensure mathematical correctness $|M_{B_i}| = 1, G^{\alpha} = 1$ (σ_{put} in Table 3).

Tuning α. We empirically benchmark the insert performance to tune α appropriately. For various data sizes ranging from 10M to 100M, we measure the per-insert data movement by

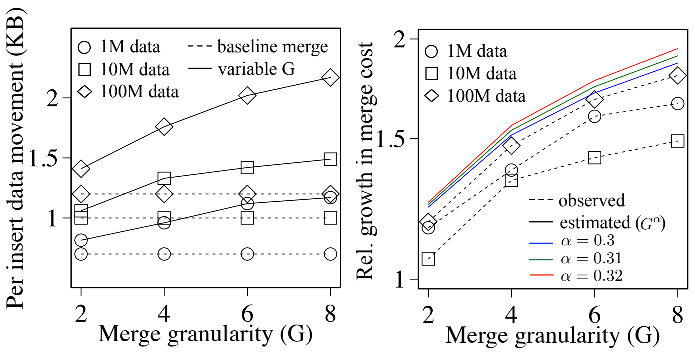


Fig. 7. G^α captures the multiplicative factor of merge cost at $\alpha = 0.3$.

varying G and compare the observed values with the baseline of a single merge $G = 0$ (Fig. 7A). We plot the relative growth in merge cost by measuring the ratio of merge costs when $G > 0$ and that of the baseline. The observed values and the predicted values of G^α are respectively denoted by the dotted and the solid, colored lines in Fig. 7B. The accuracy is the highest at $\alpha = 0.3$.

The Cost of Accessing Base Data. The base data layer is a special sub-design that stores N entries but has no data to index below it i.e., while Ω_i is accessed for reads and writes, it does not experience merge cost. This is because, as mentioned before merge cost emanates from data below the current layer and hence, merges in the last level are costed in the level above. For this level, while the cost of reads and writes stay the same, range queries incur additional costs due to sequential reads of large number of data blocks depending on the selectivity of the query (σ_{range} mentioned in Table 3).

Workload Simplicity Vs. Accuracy. For a search space with cardinality over 10^{48}, it is crucial that we strike a balance between simplicity (which is related to runtime of computing a near-optimal solution) and accuracy while evaluating designs during the search process. Therefore, we simplify the workload parameters to prevent creating an extremely high dimensional workload feature vector: (i) query keys are drawn i.i.d from our data distributions and (ii) we abstain from modeling intricate correlations and interleaving so that we amortize the cost of large workloads rather than an individual query. These simplifications help maintain practical search times while still leading to near-optimal designs as we demonstrate during our verification analysis.

5 CONCURRENCY MODEL IN LIMOUSINE

In addition to the IO cost, the performance of storage engines largely depends on the concurrency support of the underlying hardware due to the number of available CPU cores. In this section, we show how Limousine uses a learned concurrency model to capture the CPU performance of storage engine designs and consolidate it with the IO cost to accurately estimate the end-to-end performance.

Concurrency with Amdahls' Law. The key idea of concurrency logic in Limousine is to conceive the cost of concurrency as a separate layer that augments a speedup factor on top of the estimated latency from the IOs model. The concurrency model is based on Amdahl's Law [16, 44, 45] that provides the theoretical upper bound of the speedup g, any workload can obtain for a given number of cores η as $g = \frac{1}{1-\phi(1-\frac{1}{\eta})}$ where ϕ denotes the coefficient of parallelizable components within any storage engine design. Within Limousine, we derive ϕ for every storage engine design and obtain η from the hardware configuration (§6) to compute speedup using Amdahls' law. The end-to-end latency of a storage engine is then obtained as (latency due to IOs $\times \frac{1}{g}$).

Learning ϕ for Cleaned Storage Engines. Learning ϕ within Limousine has two-fold challenges: (i) as there are more than 10^{48} design possibilities, it is impossible to learn ϕ for every storage engine for different workloads and hardware and (ii) a clearned storage engine can comprise of

Algorithm 1 Algorithm for learning ϕ

```
Input: $S$, Output: $\phi^{s_i}_{op,\Omega_j}$ // $\phi$ for $op$ on sub-design $\Omega_j$ for $s_i$
1:  $R$ : {Learned (in-place), Learned (COOP), LSM, BTree, LSH}
2:  $\chi$ : {get, put, rmw, range, delete}
3:  for each provider $s_i \in S$ do
4:      for each sub-design class $\Omega_j \in R$ do
5:          for each op type $op \in \chi$ do
6:              $\Phi^{s_i}_{op,\Omega_j}$ = NULL // set of all intermediate $\phi$
7:              for each VM $v$ of distinct type do
8:                  for data $D \in [1M..100M]$ do
9:                      initialize Limousine with single layout $\Omega_j$
10:                     bulk-load the design with $D$ entries
11:                     for op count $oc \in [1M..10M]$ do
12:                         for $c \in [1..v^{CPU}]$ do
13:                             $T(c)$ = run $oc$ ops of type $q$ on $c$ cores
14:                             $\hat{g} = \frac{T(c)}{T(1)}$
15:                             get $\phi_{op,\Omega_j,v,D,c}$ using Eq (2)
16:                             $\Phi_{op,\Omega_j} = \Phi_{q,r} \cup \{\phi_{op,\Omega,v,D,c}$
17:             $\phi^{s_i}_{q,r}$ = average of values in $\Phi_{q,r}$
18: function getPhiForOp($op$, $\Omega$, $s_i$) // $\Omega$ is full storage engine
        $\phi^{s_i}_{op,\Omega} = \frac{\sum_{\Omega_j \in \Omega}(\phi^{s_i}_{op,\Omega_j} \times \text{no. of levels in}\Omega_j)}{\sum_{\Omega_j \in \Omega} \text{no. of levels in } \Omega_j}$
    function getPhiForWorkload($W$, $\Omega$, $s_i$)
        $\phi^{s_i}_{W,\Omega}$=$\sum_{\forall op \in W}$ getPhiForOp($op, \Omega, s_i$)$\times$ fraction of $op$ in $W$
```

multiple sub-designs making it possible to potentially have several values of ϕ for different parts of the storage hierarchy. To this end, instead of learning ϕ for all storage engines, we identify five drastically different sub-designs and learn ϕ for each distinct sub-design. Further, for learned sub-designs, we obtain ϕ for both in-place and COOP writes. To learn ϕ (Algorithm 1), for each sub-design Ω and operation type op, we benchmark g with different VMs, different degrees of parallelism, and data size. Every observed speedup is fed to Amdahls' law to obtain the aggregated final value of ϕ. The technical report [23] contains the learned values of ϕ with a walkthrough to derive speedup for an example workload.

6 SEARCHING FOR THE OPTIMAL DESIGN

The Cloud Provider and Hardware Space. For any input of a workload-budget combination, Limousine constructs the space of possible hardware configurations. For any cloud provider $s_i \in S$, Limousine uses the respective pricing policies [1–3] to compute the storage cost $C^{\text{storage}}_{s_i}$ of the data and then incrementally adds compute resources (VMs) to generate hardware configurations. If s_i offers k distinct VM types, we create the set of hardware configuration H_{s_i} comprising of m_i individual configurations. Each configuration $h \in H_{s_i}$ is a k-dimensional vector $\{\lambda_{i,j}\}, 1 \leq j \leq k$ where $\lambda_{i,j}$ denotes the number of instances of VM type j. Limousine adds the cost of every VM instance in h to get the compute cost $C^{\text{compute}}_{h,s_i}$ and the total cost $c_{h,s_i} = C^{\text{storage}}_{s_i} + C^{\text{compute}}_{h,s_i}$. By unifying all hardware configurations, Limousine creates the cloud-cost space C and the hardware space H, $\forall s_i \in S$, $\forall c_{h,s_i} \in C$.

The Storage Engine Design Space. In any hardware configuration h, for each VM instance $v_{i,j}$ with $v_{i,j}^{\text{mem}}$ memory and $v_{i,j}^{\text{CPU}}$ physical CPU cores, a storage engine design space is constructed to execute workload W. For each storage engine design Ω within that space, Limousine generates the possibilities for each component sub-designs Ω_u. Limousine computes the minimum and the maximum number of levels, denoted by L_{min} and L_{max}, respectively. In principle, the minimum number of sub-designs can happen only if all sub-designs are learned (due to maximum indexability of models), whereas, Btree sub-designs can maximally increase the depth of a storage engine layout. Therefore, clearned layouts can have varied depths ranging between $L_{\text{min}} = \log_{B^2}\lceil\frac{D}{B}\rceil$ and $L_{\text{max}} = \log_{B}\lceil\frac{D}{B}\rceil$. Using the domain of layout primitives, Limousine generates the individual design space of L_{max} sub-designs without knowing the actual depth a clearned layout can eventually attain after materialization. The generated design space of sub-designs of level l is denoted as Δ_{L_l}. For any $\Omega_u \in \Delta_{L_l}$, the possibilities of T, K, and Z are directly obtained from the domain values. For generating all possible allocation of memory values in Ω_u, Limousine utilizes the knowledge about the intermediate structure of Ω_u. Considering $v_{i,\text{res}}^{\text{mem}}$ as the available memory of a VM, if $(D_u \times E) - v_{i,\text{res}}^{\text{mem}} \leq 0$, it implies this sub-design cannot be placed in-memory as there will be no residual memory to construct the subsequent levels. Thus, possible values of M_{FP_u} is $\max(0, (D_u \times E) - v_{i,\text{res}}^{\text{mem}})$ and Ω_u is placed in-memory if $v_{i,\text{res}}^{\text{mem}} > 0$ after accommodating $L_{\text{max}} - l$ sub-designs. Similarly, for M_{B_u}, if $T_u > B$ (learned sub-design), Limousine assigns D_{u+1}/T_u node buffers and sets $M_{B_u} = \sum_{j=1}^{D_{u+1}/T_u} M_{B_{uj}}$ where $M_{B_{uj}}$ denotes the node buffers each configured to hold between 0 to 64 entries. At any given level, Limousine can recognize invalid states of memory allocation within sub-designs and automatically remove the sub-design possibility.

Search Space and Optimization. For a workload W, the overall search space $\Delta_{C,P}$ is expressed as triplets $(\Gamma, c, p) \in \Delta_{C,P}$ implying it takes cloud cost of c to execute W on storage engine Γ to get performance p. For an input budget b and latency l, we minimize b and maximize l through

$$\underset{(\Gamma,c,p)\in\Delta_{C,P}^{W} \text{ such that } p\leq l}{\operatorname{argmin}} (c), \qquad \underset{(\Gamma,c,p)\in\Delta_{C,P}^{W} \text{ such that } c\leq b}{\operatorname{argmin}} (p) \tag{2}$$

Although Equation 2 formalizes the cost-performance optimization for storage engines, finding the point of optimality is intrinsically NP-hard. This stems from the facts that (i) the domain space of multiple design primitives, such as memory allocation across various data structures (filters, indexes, and buffers) is inherently non-integral resulting in a countably infinite space and (ii) this characteristic also extends to the large cardinal property [36] of overall decision space resulting from primitive combinations. To address this complexity, Limousine focuses on discretizing the search space for efficient navigation to find superior designs, if not the absolute optimal one. Consequently, we refer to Limousine sub-designs and storage engine configurations as "near-optimal".

The Cardinality of the Search Space. It is impossible to know the exact cardinality as the number of possibilities of clearned structures is tightly coupled to factors such as, the size and distribution of the underlying base data and the complexity and accuracy of the models. For instance, for a level u, the number of design possibilities of T_u is directly influenced by the data Ω_u indexes and the number of models at u. This affects the number of sub-designs and combinatorial possibilities at the level. To simplify our calculations, we estimate the tightest lower-bound of the cardinality by limiting to those layouts where every level has its own sub-design comprising of linear models and BTrees only so that the maximum value of T_u is B^2. For an l-level structure, we can have $O(B^{2l})$ possibilities (in the order of 10^7 for $l = 2$ and $B = 64$, 4KB page size containing 64 byte records). On top of this, if we consider node buffers for COOP and or include the simplest binary design primitive (e.g., whether to persist a level in-memory or on disk), the cardinality of the design space blows up exponentially. Further, as we combine the hardware space within a single cloud provider s_i,

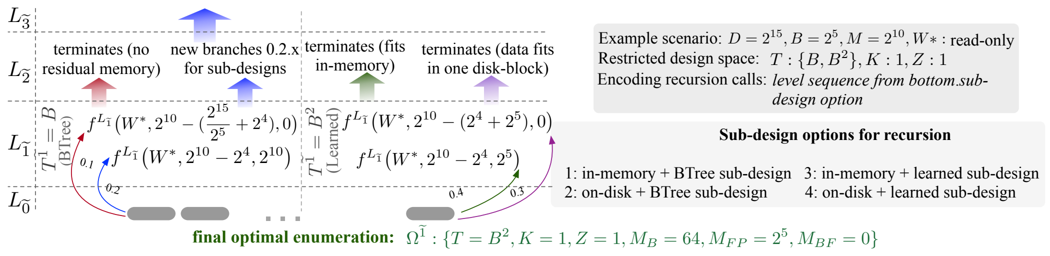

$L_{\tilde{3}}$
$L_{\tilde{2}}$
$L_{\tilde{1}}$
$L_{\tilde{0}}$
terminates (no residual memory)
new branches 0.2.x for sub-designs
terminates (fits in-memory)
terminates (data fits in one disk-block)
$T^{\tilde{1}} = B$ (BTree)
$T^{\tilde{1}} = B^2$ (Learned)
$f^{L_{\tilde{1}}}(W^*, 2^{10} - (\frac{2^{15}}{2^5} + 2^4), 0)$
$f^{L_{\tilde{1}}}(W^*, 2^{10} - 2^4, 2^{10})$
$f^{L_{\tilde{1}}}(W^*, 2^{10} - (2^4 + 2^5), 0)$
$f^{L_{\tilde{1}}}(W^*, 2^{10} - 2^4, 2^5)$
0.1
0.2
0.4
0.3
Example scenario: $D = 2^{15}, B = 2^5, M = 2^{10}, W*$: read-only
Restricted design space: $T : \{B, B^2\}, K : 1, Z : 1$
Encoding recursion calls: level sequence from bottom.sub-design option
Sub-design options for recursion
1: in-memory + BTree sub-design
2: on-disk + BTree sub-design
3: in-memory + learned sub-design
4: on-disk + learned sub-design
final optimal enumeration: $\Omega^{\tilde{1}} : \{T = B^2, K = 1, Z = 1, M_B = 64, M_{FP} = 2^5, M_{BF} = 0\}$

Fig. 8. The enumeration and greedy materialization of the optimal storage engine.

the cardinality of the search space becomes $m_i \times$(number of possible layouts within a single VM)^k [22]. Therefore for AWS, with $m_i = 74612$ and $k = 6$, the cardinality of the search space goes beyond 10^{48} or quindecillion of possibilities. Note that, this lower-bound estimation of the cardinality excludes several other design considerations such as, several variants for clearned layouts when combined with classical sub-designs as well as the entire subspace of pure classical structures.

Navigating the Search Space. Once the search space is generated, Limousine navigates all possible configurations and constructs the continuum. For each storage engine configuration $h \in H_{s_i}$, Limousine first shards the workload and data using off-the-shelf sharding algorithms [34]. Next, it searches through all possible layouts within each VM and partially prunes the space by selecting the near-optimal configuration for each h.

Greedy Search of Optimal Layout. Limousine greedily constructs the near-optimal sub-design for any level L_i i.e., Limousine incrementally constructs the engine design by recursively materializing the sub-designs in a level-by-level manner. To mathematically express the recursion of greedy materialization, we introduce the notation of $\sim$ to denote levels and sub-designs in a bottom-up manner. e.g., $L_{\tilde{1}}$ is the 1st level closest to the data i.e., the last index level in a layout. We formulate the greedy search f:

$$f(W^*, v_{i,j}^{\text{mem}}, D^*) = f^{L_{\tilde{1}}}(W^*, v_{i,j}^{\text{mem}}, D^*) + f\left(W^*, v_{i,j}^{\text{mem}} - (M_{B_{\tilde{1}}} + M_{FP_{\tilde{1}}}), D_{\tilde{1}}\right)$$

where $f^{L_{\tilde{l}}}(W^*, v_{i,j}^{\text{mem}}, D^*)$ generates the design space and selects the near-optimal sub-design for $L_{\tilde{l}}$ for the specified workload, memory, and data arguments. The recursion terminates when for a certain level $L_{\tilde{j}}$, $D_{\tilde{j}}$ fits in a single block and $L_{\tilde{j}}$ becomes L_1 for the resulting layout. We show an example with a read-only workload to illustrate the greedy function calls as Limousine materializes the near-optimal engine (Fig. 8). Once the space is partially pruned, Limousine follows a similar technique to Cosine [22] to generate the continuum or Pareto frontier of cost and performance.

7 CODE GENERATION AND SELF-DESIGNING

The code of the target storage engine is materialized with a Rust-based templated key-value engine – Limousine-engine (Fig. 2D).

Procedural Macros: The Core of Code Generation in Limousine. As Limousine-engine is engineered to instantiate code for more than sextillions of storage engines, it is critical to ensure that the code generation process does not pollute the namespace or the module (where it was invoked) with boilerplate codes. Therefore, the core of Limousine-engine is designed with procedural macros in Rust. These macros are essentially compile-time Rust codes that take a simplistic layout specification in the form of an input stream of tokens. When the macro is invoked at compile-time, it automatically parses this stream and materializes the abstract syntax tree in the form of succinct and optimized code blocks tailored for the data layout of the target storage engine.

AWS			
Instance Type	vCPU	Memory (GB)	Hourly Rate ($)
r5d.large	2	16	0.091
r5d.xlarge	4	32	0.182
r5d.2xlarge	8	64	0.364
r5d.4xlarge	16	128	0.727
r5d.12xlarge	48	384	2.181
r5d.24xlarge	96	768	4.362
r5d.metal	96	768	4.362

GCP			
Instance Type	vCPU	Memory (GB)	Hourly Rate ($)
n1-highmem-2	2	13	0.0745
n1-highmem-4	4	26	0.1491
n1-highmem-8	8	52	0.2981
n1-highmem-16	16	104	0.5962
n1-highmem-32	32	208	1.1924
n1-highmem-64	64	416	2.3849
n1-highmem-96	96	624	3.5773

AZURE			
Instance Type	vCPU	Memory (GB)	Hourly Rate ($)
E2 v3	2	16	0.0782
E4 v3	4	32	0.1564
E8 v3	8	64	0.3128
E16 v3	16	128	0.6256
E20 v3	20	160	0.7409
E32 v3	32	256	1.2512
E64 v3	64	512	2.5024

Table 4. Parameters used for cloud hardware and pricing

Converting Design Primitives to Rust Generics. For procedural macros to work efficiently, Limousine-engine is based on Rust's *parametric polymorphism* or *generics* to allow parameterization of traits, functions, structures, and enumerations to prevent code duplication and type safety. These generics are used to translate the storage engine design primitives to unit-testable code blocks. They allow flexibility and reusability in the code structure and simplify the process of code generation for diverse storage engine architectures. For instance, the `Model` generic in Limousine is a design primitive to include diverse methods of learning and segmentation such as `PiecewiseLinear`, `LinearSpline`, and `LinearRegression`. Other examples of Limousine generics are `Search` that accommodates `ApproximationSearch`, or `BinarySearch`, or `LinearSearch` and `Inserts` to materialize both `InPlace` or `OutOfPlace` algorithms.

Materizalizing Code for Full Storage Engines. To realize the implementation of any storage engine design, Limousine generalizes the smallest building block in the form of a Rust trait called `NodeLayer`. A `NodeLayer` is a generic responsible for indexing any bounded set of keys through either models (as used in clearned structures) or storage-based indexing (as in classical data structures such as BTree variants). Regardless of the sub-design type, every node contains an ϵ error bound which is set to B for a classical sub-design and is set to $\frac{B}{2}$ for learned sub-designs to bound the data movement of the binary search to a single disk block. Several `NodeLayers` can be consolidated to form more complex traits of a storage engine such as an `InternalLayer` for storing the index layers and `BaseLayer` for storing the base data. To allow for a common access interface for classical and learned sub-designs, nodes are realized with pointer-to-pointer mappings i.e., each node points to another node in the `InternalLayer` or a `BaseLayer` below. Any `InternalLayer` can reside in memory or be persisted on disk. Persistence of nodes of any layer is realized in separate files with a fill-factor so that during inserts (i) both in-place and out-of-place inserts can be seamlessly materialized, and (ii) splitting a node is as simple as allocating a node and updating the pointers. Limousine also includes additional optimizations such as a `CacheLayer`, a special `NodeLayer` to cache the maximal keys within nodes to reduce data movement and memory overhead during reads and writes.

Characteristics and maintainability of generated storage engine code. The length of the generated code is in the order of $O(n)$, where n is the number of specified sub-designs of a storage engine. Since code generation generally can grow quite complex, we made it a general design principle of Limousine to keep the macro as simple as possible, and move most of the implementation complexity to core Limousine components. These components can evolve independently of the macro system, ensuring adaptability. At present, the lines of code (LoC) of Limousine core is about 3K. In contrast, the macro system remains relatively stable at approximately 1K LoC, regardless of

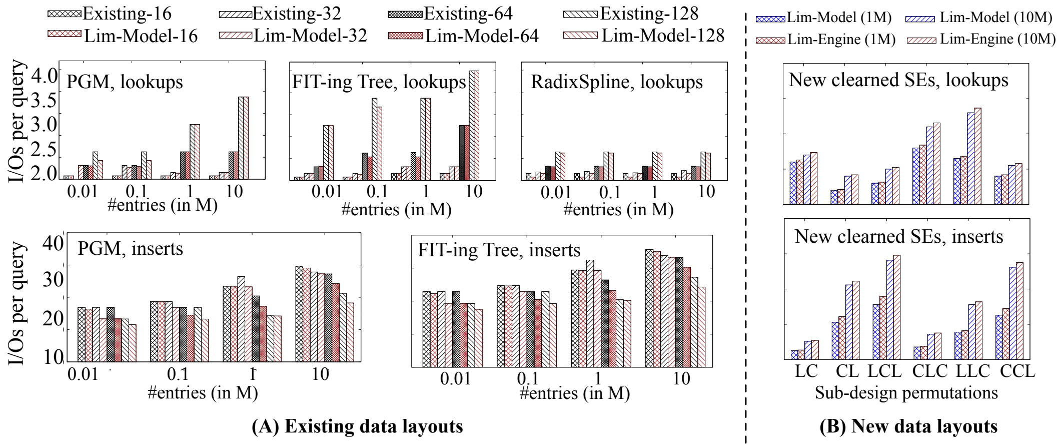


Fig. 9. Limousine models are consistent and accurate across data, error-margins, and operations on new and existing storage engines.

the specific data layout within the storage engine. For example, the LoC generated for a 2-level pure learned engine is 1248 resulting in a total of about 4228 LoC with the core component. An example of the structure of the generated code can be found in the technical report [23].

8 EXPERIMENTAL EVALUATION

We now demonstrate the self-designing ability of Limousine. First, we verify Limousine's cost models with diverse storage engine designs and workloads. Next, we show how the proposed COOP algorithm outperforms existing IP algorithms on clearned structures. Finally, we show that with its self-designing abilities Limousine scales with data, workload diversity, and cloud budgets, outperforming state-of-the-art engines by up to three orders of magnitude.

Baselines. We compare Limousine against four classical baselines: RocksDB (LSM-tree) [9], WiredTiger (B-tree) [11], FASTER (LSH-table) [21], Cosine [22]. For learned storage engines, as there are no existing systems, we integrate the layout specification of PGM [40], FIT-ing Tree [42], and RadixSpline [55] with Limousine-engine to generate the corresponding storage engine. Other than Cosine, we set each of these baselines to their default configuration.

Datasets. We use two synthetic and two real-world datasets comprising of $10 - 100$M records: D1. synthetic data with uniform and Zipfian distributions generated with YCSB generator, D2. *lognormal* dataset with values generated using a lognormal distribution with $\mu = 0, \sigma = 2$ [35], D3. *longitude* dataset sampled from the Open Street Maps [6] comprising of longitudes of locations around the world and D4. *longlat* dataset, also sampled from the Open Street Maps [6] containing non-linearly distributed keys generated by applying point conversion to every pair of longitude and latitude.

Workloads. We cover all core YCSB workloads A-F, and also test with several variations including mixed workloads with and without range queries across uniform, zipfian, and normal distributions.

Cloud Parameters for Hardware and Pricing. Table 4 specifies the VM specifications and pricing model for evaluation and experimentation. Experiments involving high cloud budgets have been done on a machine with Core i5 processor and 16GB DDR4 RAM.

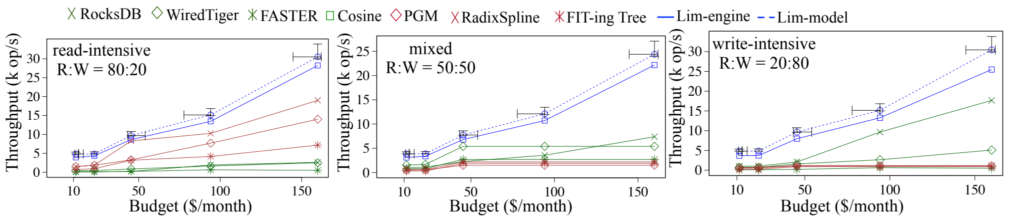


Fig. 10. Limousine's cost models retain accuracy on diverse workloads on Azure cloud.

8.1 Verifying Accuracy of Limousine's Models

Limousine's ability to self-design and materialize scalable storage engines largely depends on the accuracy of its models across the entire space of storage engine designs, hardware, data, and operations. We design the process of verification in three steps.

Step 1: Verifying Models for Existing Storage Engines. To evaluate the model accuracy on existing learned baselines i.e., PGM, FIT-ing Tree, and RadixSpline, we use dataset D1 with varying size from 0.01M to 1M for verifying 10M reads and writes each (Fig. 9A). For each baseline, we vary the error parameter ϵ and record the IO per operation. This is indicated as "Existing-ϵ" for different designs and values of ϵ. We provide the corresponding layout specification as input to Limousine's models and estimate IO of every operation as indicated by "Lim-Model-ϵ" in Fig. 9A. We observe that with varying ϵ and data size, Limousine's models are consistent across all operations as the shape of the prediction matches with that of the actual pattern. For lookups, as data increases, the models have an accuracy of more than 90%. This is because the IO cost of lookups is regulated by the number of levels and with more data, although the estimated number of models per level can differ, this does not affect the total number of levels as the size ratio of levels differ exponentially. For inserts, the models are 95% accurate when $\epsilon \leq 32$ beyond which the accuracy drops below 80%. This is because larger ϵ allows longer data segments thereby creating higher degrees of variance in the indexability of models within a single level which, in turn, makes the average-case analysis of the models less accurate. In practice, this inaccuracy is not a concern as for storage engines, ϵ is usually set to $\frac{B}{2}$ [40] to ensure that the data movement for every model does not go beyond 1 I/O.

Step 2: Verification on New Storage Engines. We extend the experiment to evaluate the correctness and accuracy of these models on new storage engines within Limousine's design space. For this purpose, we generate 6 definitions of data layouts of new designs comprising of 2-3 sub-designs permuted differently (Fig. 9B). For each of these new layouts, we materialize the actual implementation of Limousine-engine and repeat a similar process to obtain model estimations. We use our proposed COOP algorithm for inserts. We notice that on average across data and operation, the model retains an accuracy of 92% over the new designs.

Step 3: Verifying Full Worklow of Limousine on Cloud. We run the entire workflow of Limousine on Azure cloud to verify the holistic correctness and accuracy of the models, the search algorithm, and the eventual generation of the continuum across diverse budgets and workloads. For this experiment, we use the first five VM types (Table 4) and workloads comprising of 90M data entries and 30M queries with different read-write ratios. Each experiment is run on a 3VM-cluster thereby limiting the budget to $150/month [4, 5] (Fig. 10). For each workload-budget combination, we compare the estimated performance of Limousine's suggested storage engine configuration with the observed performance of the corresponding instantiation when deployed through Limousine-engine. We observe that within Azure data centers, based on the availability of VMs, there can be a variability of up to 15% in cloud-cost as VMs may be charged differently across different data centers. Similarly, for performance estimations, we observe variability due to the shape estimation

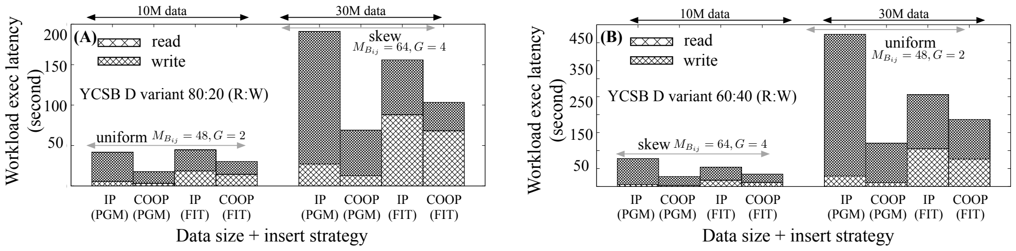


Fig. 11. COOP improves not only writes but also the overall structure leading up to 70% increase in overall performance.

of the storage hierarchy and aligned with our observations from steps 1 and 2, this variability can range between 7% to 20% from read-heavy to write-heavy workloads. We show this with error bars along both the cost and performance axes. Overall, the shape of the predicted Pareto curve matches that of the observed performance and when averaged over workload types and cloud cost, Limousine never exceeds the budget by more than 15% or misses the performance target by more than 18%.

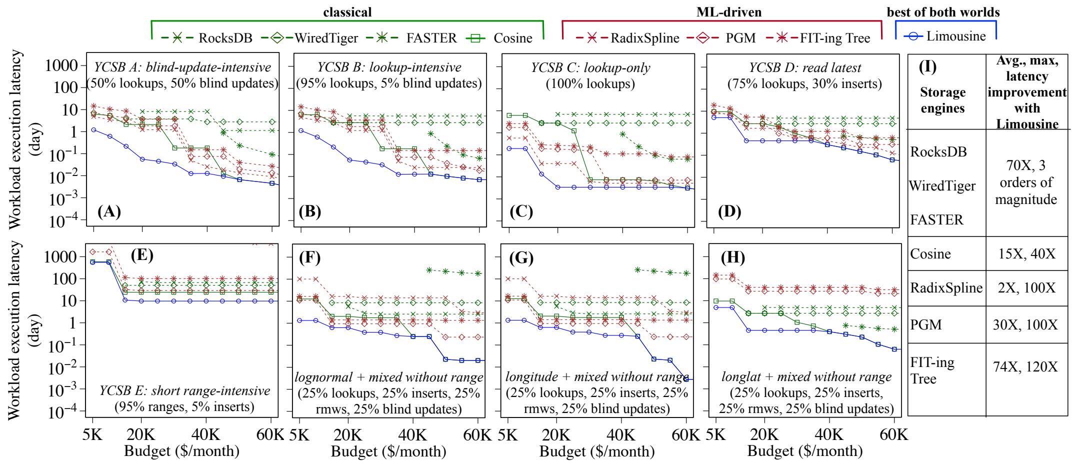
8.2 COOP Outperforms In-place Writes

We provide a comprehensive performance analysis of COOP with IP strategies (Fig. 11) on PGM and FIT-ing Tree[1]. Although we run this experiment on YCSB D with 95% lookups and 5% inserts [27], we slightly alter the composition so that we can stress-test with more inserts making up to 20% and 40% of the workload (Fig. 11A and B). Although Fig. 6E presents the effective values of $M_{B_{ij}}$ and G to strike a balance between latency and memory consumption, the resulting parameter still space encompass a significant number of possibilities with 441 COOP tunings (7 merge granularities X 63 buffer sizes). To streamline our results and maintain judicious experimentation, we assign informed defaults of $M_{B_{ij}} = \{48, 64\}$ and $G = \{2, 4\}$ so that, on average, we can maximize performance benefits across a spectrum of data sizes for both uniform and skewed workloads. We observe that with diverse data sizes, COOP not only improves the write performance by up to 3.6× but it does so without hurting the reads. In fact, in most cases, we observe that the read performance improves by up to 2×. This is because COOP prevents the eventual degradation of the tree structure (§3) in addition to reducing the data movement significantly. For mixed workloads, this leads to an overall benefit of 4×, especially with the increase of both the data size and the number of inserts in the workload.

8.3 Limousine Dominates Across Diverse Workloads and Cloud Budgets

We now demonstrate that Limousine can search over the exhaustive design space of storage engines, hardware, and cloud providers and self-design the near-optimal storage engine that outperforms all baselines across diverse workload-budget combinations. For each provider, we use the experimental parameters used for cloud pricing policies and VM specifications (Table 4). We set AWS as the default cloud provider for all baselines except Cosine. Fig. 12 shows 8 different experiments with varied monthly cloud budgets ranging from \$5K to \$60K – Fig. 12A through 12E use the corresponding core YCSB workload on data set D1 while Fig. 12F through 12H use a mixed workload (25% lookups, 25% inserts, 25% rmws, 25% blind updates) on data sets D2, D3, and D4, respectively. As we have already proven Limousine's model accuracy, we derive the results and error margins from the models as paying for the actually experimented budgets up to \$60K/month would not be practical.

[1]RadixSpline is a learned index that does not support inserts.



(I) Storage engines	Avg., max, latency improvement with Limousine
RocksDB WiredTiger FASTER	70X, 3 orders of magnitude
Cosine	15X, 40X
RadixSpline	2X, 100X
PGM	30X, 100X
FIT-ing Tree	74X, 120X

Fig. 12. Limousine self-designs to the near-optimal storage engine architecture across diverse workload-budget combinations.

Classical Storage Engines. For all workloads and budgets in Fig. 12, Limousine outperforms or matches the performance of the baselines. Table 12(I) shows that for industrial classical storage engines (RocksDB, WiredTiger, FASTER) the reduction in latency can be up to three orders of magnitude lower. Even compared to a self-designing academic storage engine (Cosine), Limousine achieves a latency up to 40× less as Cosine is restricted to only classical data layouts thereby being unable to look at trillions of other design possibilities that cater to diverse workloads. For smaller budgets (e.g., less $40K/month for A, B, D, F, G, H), Limousine scales better by self-designing storage engines with succinct data layouts and co-optimizing hardware and budget to minimize latency. For higher budgets, as data can be indexed within an in-memory hash table (LSH), Limousine mostly converges to Cosine with only two exceptions. First, when the workload is lookup-only (Fig. 12C), Limousine creates pure learned storage engines and index with a succinct radix table (inspired by RadixSpline) on top for faster lookups. As budget grows and memory becomes affordable, Limousine also generates non-tree-like storage designs with flat logs indexed by hash table (inspired by FASTER). Second, when workloads are range-intensive (Fig. 12E), the data movement between Limousine and Cosine have similar patterns with a shallower index structure in Limousine.

Learned Baselines. For learned baselines, the performance varies with the workload-budget combinations. While RadixSpline outperforms PGM and FIT-ing tree for lookups (A-D), it does not support inserts (E-H). PGM being a pure learned index outperforms FIT-ing tree with succinct linear models, however, the latter takes over with inserts in the workload. On the other hand, Limousine significantly minimizes latency with its uniquely generated clearned layouts, COOP writes, and search algorithm to find the best design. On average, Limousine outperforms RadixSpline, PGM, and FIT-ing tree by 2×, 30×, and 74×, respectively.

9 RELATED WORK

As Limousine is an ongoing effort in the intersection of machine learning and databases, we discuss areas related to instance-optimization, self-designing systems, data structures, and algorithms.

Database Tuning and Instance-Optimization. Database engineers have a long history of manually operating physical, technical, and design aspects of databases [13, 14, 24, 29, 32, 49, 65, 71]. With growing data volumes and new cloud applications, DBMS configurations are regulated through semi-automatic tuning through rule-based [30, 31, 33, 57, 68–70, 72] and ML-based techniques

[37, 60, 61, 64, 75]. However, tuning marginally contributes to scalability as it does not alter the fundamental behavior of the systems. Limousine, on the other hand, is based on a drastically different methodology to discover and create an extensive design space for the core design decisions of storage for fast materialization of designs that scale with cloud cost and performance.

Learned Data Structures. The space of learned data structures is growing and evolving fast [15, 35, 40, 42, 55, 58, 74, 76, 82]. While these structures offer superior read performance with a low memory footprint, they still suffer when it comes to writes. Consequently, a comprehensive approach to crafting holistic learned data systems that seamlessly integrate these advanced structures as fundamental design elements remains elusive. Limousine makes it possible to realize holistic learned systems through a wider design space of data layouts, efficient read/write algorithms on these structures, as well as a reflection of the system design on cloud deployment and hardware.

Self-Designing Layouts. The vision of a self-designing system is to automatically create the perfect storage and algorithms for the whole system using the first principles of design [50, 51]. Idreos *et al.* proposes the Data Calculator [52] to identify the first-order design primitives that capture the design principles of designing classical data structures. Cosine [22] took the self-designing concept at the level of a whole system showing that it is possible to self-design full key-value storage engines. However, Cosine does not scale with data and cloud cost. Limousine allows self-design in a way that is scalable with data size and cloud budgets by expanding the design space of key-value engines to include learned components that are efficiently updated.

10 OPPORTUNITIES AND DISCUSSION

Limousine showcases that it is possible to self-design scalable storage engines on the cloud with cloud cost and performance controls. Realizing the full vision of self-designing storage engines towards industry-ready systems includes numerous additional and exciting research challenges. For example, such research themes include adapting storage engine designs for workload fluctuations, expanding for more hardware and providers, and ensuring robustness guarantees as part of system design. Furthermore, an exciting path is to consider more complex data models beyond key-value such as the full relational model. Towards this goal, the Image Calculator takes the self-designing concept to the space of image storage data models for AI [73].

ACKNOWLEDGMENTS. This work is partially funded by the USA Department of Energy project DE-SC0020200.

REFERENCES

[1] 2019. AWS Calculator. https://calculator.s3.amazonaws.com/index.html.

[2] 2019. Azure Calculator. https://azure.microsoft.com/en-us/pricing/.

[3] 2019. GCP Calculator. https://cloud.google.com/products/calculator/.

[4] 2019. Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.

[5] 2020. General purpose Azure VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-general?toc=/azure/virtual-machines/linux/toc.json&bc=/azure/virtual-machines/linux/breadcrumb/toc.json.

[6] 2024. Amazon OpenStreetMap on AWS. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[7] 2024. Apache Cassandra. http://cassandra.apache.org.

[8] 2024. CouchDB. http://couchdb.apache.org/.

[9] 2024. Facebook RocksDB. https://github.com/facebook/rocksdb.

[10] 2024. MongoDB. http://www.mongodb.com/.

[11] 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[12] Hussam Abu-Libdeh, Deniz Altinbuken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. 2020. Learned Indexes for a Google-scale Disk-based Database. *NeurIPS* abs/2012.12501 (2020).

[13] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505.

[14] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 359–370. https://doi.org/10.1145/1007568.1007609

[15] Abdullah Al-Mamun, Hao Wu, and Walid G. Aref. 2020. A Tutorial on Learned Multi-Dimensional Indexes. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems* (Seattle, WA, USA) *(SIGSPATIAL '20)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3397536.3426358

[16] G. Amdahl. 1967. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS spring joint computer conference*.

[17] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1185–1196. https://doi.org/10.1145/2463676.2465296

[18] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark D. Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *International Conference on Extending Database Technology*.

[19] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Maryland) *(SODA '03)*. Society for Industrial and Applied Mathematics, USA, 546–554.

[20] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and Xiaoyang Sean Wang. 2013. LogKV: Exploiting Key-Value Stores for Log Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2013/Papers/CIDR13{_}Paper46.pdf

[21] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. 2018. Faster: A Concurrent Key-Value Store with In-Place Updates. In *ACM SIGMOD*.

[22] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2022. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. In *Proceedings of the VLDB Endowment*.

[23] Subarna Chatterjee, Mark Pekala, Lev Kruglyak, and Stratos Idreos. 2024. Technical Report for Limousine.

[24] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 3–14.

[25] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (jun 1979), 121–137. https://doi.org/10.1145/356770.356776

[26] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (jun 1979), 121–137. https://doi.org/10.1145/356770.356776

[27] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[28] G. Cudahy. 2022. The Key To Intelligent Connectivity In A World Awash With IoT Data? Making Decisions At The Edge. https://www.forbes.com/sites/forbestechcouncil/2022/03/04/the-key-to-intelligent-connectivity-in-a-world-awash-with-iot-data-making-decisions-at-the-edge/?sh=7521b0794b13

[29] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1 (2010), 48–57. https://doi.org/10.14778/1920841.1920853

[30] Benoît Dageville and Mohamed Zait. 2002. Chapter 88 - SQL Memory Management in Oracle9i. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, and Dimitris Papadias (Eds.). Morgan Kaufmann, San Francisco, 962–973. https://doi.org/10.1016/B978-155860869-6/50095-0

[31] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[32] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. https://doi.org/10.1145/3183713.3196927

[33] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 449–466. https://doi.org/10.1145/3299869.3319903

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available

Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[35] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969–984. https://doi.org/10.1145/3318464.3389711

[36] F. R. Drake. 1974. Set Theory: An Introduction to Large Cardinals. *Studies in Logic & the Foundations of Mathematics, Elsevier Science Ltd.* 76 (1974).

[37] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with ITuned. *Proc. VLDB Endow.* 2, 1 (aug 2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[38] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* IEEE Computer Society, 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[39] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes so Effective?. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20).* JMLR.org, Article 293, 10 pages.

[40] 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 (apr 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[41] W. Fokoue, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *In Proceedings of the International Conference on Management of Data, SIGMOD.*

[42] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[43] Goetz Graefe. 2010. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2010). https://doi.org/10.1561/1900000028

[44] J. L. Hennessy and D. A. Patterson. 2003. *Computer Architecture: A Quantitative Approach.* Morgan Kauffman.

[45] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (July 2008), 33–38. https://doi.org/10.1109/MC.2008.209

[46] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. https://doi.org/10.1145/3299869.3314041

[47] S. Idreos and M. Callaghan. 2020. Key-Value Storage Engines. In *ACM SIGMOD Tutorial.*

[48] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Biennial Conference on Innovative Data Systems Research (CIDR).*

[49] Stratos Idreos and Tim Kraska. 2019. From Auto-tuning One Size Fits All to Self-designed and Learned Data-intensive Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

[50] Stratos Idreos, Konstantinos Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41 (2018), 64–75.

[51] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Michael S. Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. 2019. Learning Data Structure Alchemy. *IEEE Data Eng. Bull.* 42 (2019), 47–58.

[52] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 535–550. https://doi.org/10.1145/3183713.3199671

[53] M. R. Jain. 2019. Why we choose Badger over RocksDB in Dgraph. https://blog.dgraph.io/post/badger-over-rocksdb-in-dgraph/.

[54] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 301–315.

[55] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing

Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[56] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[57] Eva Kwan, Sam Lightstone, Berni Schiefer, Adam Storm, and Leanne Wu. 2003. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. 620–629.

[58] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-Grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (oct 2021), 321–334. https://doi.org/10.14778/3489496.3489512

[59] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree Indexing on Solid State Drives. *Proc. VLDB Endow.* 3, 1–2, 1195–1206. https://doi.org/10.14778/1920841.1920990

[60] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.3196908

[61] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1248–1261. https://doi.org/10.1145/3448016.3457276

[62] P. Malkowski. 2018. MyRocks Disk Full Edge Case. https://www.percona.com/blog/2018/09/20/myrocks-disk-full-edge-case/.

[63] W. Oledzki. 2013. memcached is a weird creature. http://hoborglabs.com/en/blog/2013/memcached-php.

[64] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3211–3221. https://doi.org/10.14778/3476311.3476411

[65] Andrew Pavlo, Carlo Curino, and Stan Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (05 2012). https://doi.org/10.1145/2213836.2213844

[66] Jun Rao and Kenneth A. Ross. 2000. Making B+- Trees Cache Conscious in Main Memory. *SIGMOD Rec.* 29, 2 (may 2000), 475–486. https://doi.org/10.1145/335191.335449

[67] J. Rydning. 2022. Worldwide IDC Global DataSphere Forecast, 2022–2026: Enterprise Organizations Driving Most of the Data Growth. https://www.idc.com/getdoc.jsp?containerId=US49018922

[68] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-Based Data Stores. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2489–2497. https://doi.org/10.1145/3514221.3522563

[69] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-Based Data Stores. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2489–2497. https://doi.org/10.1145/3514221.3522563

[70] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2429–2432. https://doi.org/10.1145/3514221.3520169

[71] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 893–908. https://doi.org/10.1145/3318464.3389757

[72] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2216–2229. https://doi.org/10.14778/3476249.3476274

[73] Utku Sirin and Stratos Idreos. 2024. The Image Calculator: 10x Faster Image-AI Inference by Replacing JPEG with Self-designing Storage Format. In *Proceedings of the ACM on Management of Data (PACMMOD) (SIGMOD '24)*. Association for Computing Machinery. https://doi.org/10.1145/3639307

[74] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (apr 2023), 1992–2004. https://doi.org/10.14778/3594512.3594528

[75] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[76] 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* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/3409963.3410496

[77] Z. Wei, G. Pierre, and C. H. Chi. 2011. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *IEEE Transactions on Services Computing* (2011), 525–539.

[78] C. Weinschenk. 2018. Deeper Dive—Will growth in data traffic ever slow down? https://www.fiercewireless.com/wireless/special-report-will-growth-data-traffic-ever-slow-down

[79] J. Wise. 2022. How Much Data Is Created Every Day In 2022? http://tinyurl.com/dataCreatedEveryday

[80] Haoyuan Xing, Sofoklis Floratos, Spyros Blanas, Suren Byna, M. Prabhat, Kesheng Wu, and Paul Brown. 2018. Array-Bridge: Interweaving Declarative Array Processing in SciDB with Imperative HDF5-Based Programs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 977–988. https://doi.org/10.1109/ICDE.2018.00092

[81] Cong Yan and Alvin Cheung. 2019. Generating Application-Specific Data Layouts for in-Memory Databases. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1513–1525. https://doi.org/10.14778/3342263.3342630

[82] Hadian Ali Heinis Thomas Yang Guang, Liang Liang. 2023. FLIRT: A Fast Learned Index for Rolling Time frames. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT) (EDBT '23)*. Association for Computing Machinery.

Received July 2023; revised October 2023; accepted November 2023