# Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine

Subarna Chatterjee
Harvard University
subarna@seas.harvard.edu

Meena Jagadeesan
Harvard University
mjagadeesan@seas.harvard.edu

Wilson Qin
Harvard University
wilson@seas.harvard.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

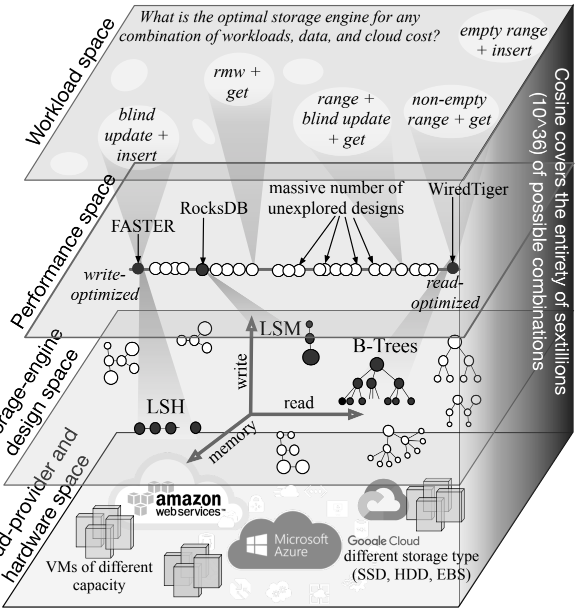
## ABSTRACT

We present a self-designing key-value storage engine, Cosine, which can always take the shape of the close to "perfect" engine architecture given an input workload, a cloud budget, a target performance, and required cloud SLAs. By identifying and formalizing the first principles of storage engine layouts and core key-value algorithms, Cosine constructs a massive design space comprising of sextillion ($10^{36}$) possible storage engine designs over a diverse space of hardware and cloud pricing policies for three cloud providers – AWS, GCP, and Azure. Cosine spans across diverse designs such as Log-Structured Merge-trees, B-trees, Log-Structured Hash-tables, in-memory accelerators for filters and indexes as well as trillions of hybrid designs that do not appear in the literature or industry but emerge as valid combinations of the above. Cosine includes a unified distribution-aware I/O model and a learned concurrency-aware CPU model that with high accuracy can calculate the performance and cloud cost of any possible design on any workload and virtual machines. Cosine can then search through that space in a matter of seconds to find the best design and materializes the actual code of the resulting storage engine design using a templated Rust implementation. We demonstrate that on average Cosine outperforms state-of-the-art storage engines such as write-optimized RocksDB, read-optimized WiredTiger, and very write-optimized FASTER by 53x, 25x, and 20x, respectively, for diverse workloads, data sizes, and cloud budgets across all YCSB core workloads and many variants.

**PVLDB Reference Format:**
Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos.
Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine.
PVLDB, 15(1): 112 - 126, 2022.
doi:10.14778/3485450.3485461

## 1 REASONING ABOUT CLOUD COSTS

**Application Diversity.** Key-value stores [71, 72, 87] serve as the storage backbone for a wide range of applications such as graph processing in social media [27, 59], event log processing [42], web applications [119], and online transaction processing [98]. Relational systems increasingly use key-value stores for core functionality such as the catalog and indexing [72]. Machine Learning pipelines deploy key-value stores for data exploration, storing features, and maintaining debugging data [116, 117]. Bitcoin uses a key-value engine as its primary node infrastructure [115].



Workload space
What is the optimal storage engine for any combination of workloads, data, and cloud cost?
empty range + insert
rmw + get
blind update + insert
range + blind update + get
non-empty range + get
Performance space
massive number of unexplored designs
FASTER
RocksDB
WiredTiger
write-optimized
read-optimized
Storage-engine design space
LSM
B-Trees
write
read
memory
LSH
Cloud-provider and hardware space
amazon web services
Microsoft Azure
Google Cloud
different storage type (SSD, HDD, EBS)
VMs of different capacity
Cosine covers the entirety of sextillions (10^36) of possible combinations

**Figure 1: Fixed-design systems capture only a small fraction of the possible storage-engine design space on the cloud.**

**Movement on the Cloud.** With the growing diversity of applications and data sizes, key-value stores are increasingly deployed on the cloud to take advantage of auto-scaling and on-demand pricing. For instance, Amazon Web Services (AWS) cloud [23], the Google Cloud Platform (GCP) [101], and Microsoft Azure [36] cloud provide support for widely used key-value stores such as MongoDB [95], CouchDB [48], RocksDB [56], and Cassandra [25].

**The Problem: Reasoning About Cloud Costs & Performance.** We show that existing key-value systems fail to scale in the face of the combined challenge: growing application diversity and growing data sizes, which in turn result in growing cloud budgets. The source of the problem is in the inherent complexity of data system design, opacity of cloud infrastructures, and the numerous metrics and factors that affect performance and cloud cost. As a result, organizations, systems administrators, and even expert data system

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:10.14778/3485450.3485461

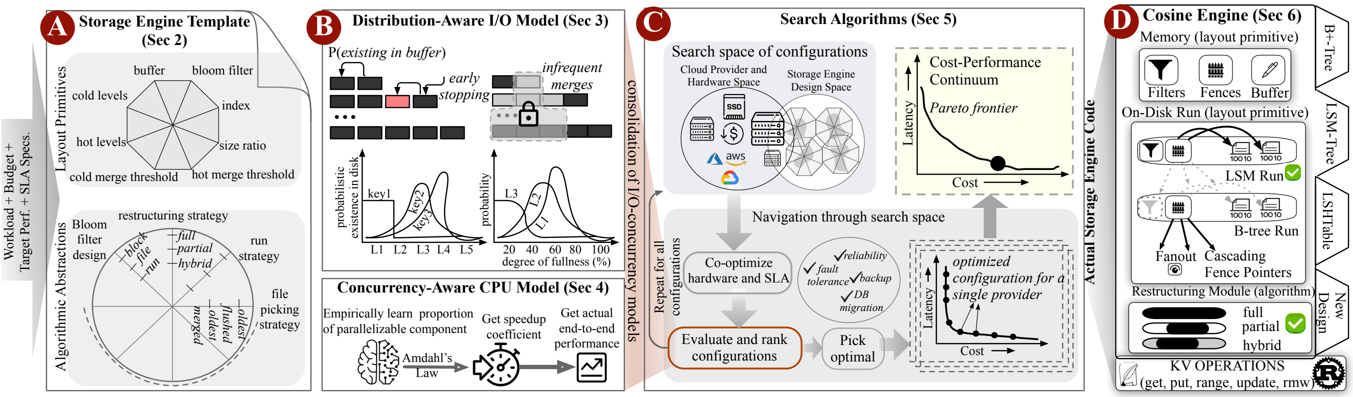

Workload + Budget + Target Perf. + SLA Specs.
A
Storage Engine Template (Sec 2)
Layout Primitives
buffer
bloom filter
cold levels
index
hot levels
size ratio
cold merge threshold
hot merge threshold
Algorithmic Abstractions
restructuring strategy
Bloom filter design
block
file
run
full
partial
hybrid
run strategy
file picking strategy
merged
oldest
flushed
oldest
B
Distribution-Aware I/O Model (Sec 3)
P(existing in buffer)
early stopping
infrequent merges
probabilistic existence in disk
key1
key2
key3
L1
L2
L3
L4
L5
probability
L3
L2
L1
20
40
60
80
100
degree of fullness (%)
Concurrency-Aware CPU Model (Sec 4)
Empirically learn proportion of parallelizable component
Amdahl's Law
Get speedup coefficient
Get actual end-to-end performance
consolidation of I/O-concurrency models
C
Search Algorithms (Sec 5)
Search space of configurations
Cloud Provider and Hardware Space
Storage Engine Design Space
SSD
aws
Cost-Performance Continuum
Latency
Pareto frontier
Cost
Navigation through search space
Repeat for all configurations
Co-optimize hardware and SLA
reliability
fault tolerance
backup
DB migration
Evaluate and rank configurations
Pick optimal
optimized configuration for a single provider
Latency
Cost
Actual Storage Engine Code
D
Cosine Engine (Sec 6)
Memory (layout primitive)
Filters
Fences
Buffer
On-Disk Run (layout primitive)
10010
10010
LSM Run
B-tree Run
Fanout
Cascading Fence Pointers
Restructuring Module (algorithm)
full
partial
hybrid
KV OPERATIONS (get, put, range, update, rmw)
B+-Tree
LSM-Tree
LSHTable
New Design

**Figure 2: Cosine produces optimal key-value storage engines given a budget, a workload, and a target performance.**

designers cannot predict how a specific combination of a *key-value store design*, a *cloud provider (their pricing policies and hardware)*, and a *specific workload (data and queries)* will ultimately behave in terms of end-to-end performance and cloud-cost requirements [55, 58, 93, 103]. This can lead to severe performance bottlenecks or cost requirements that are exceptionally hard to get out of given the time, effort, and risk involved in switching to a different system or investing in building a new one. There are numerous manifestations of this problem. We elaborate with two characteristic examples.

**1. Choosing from Off-the-Shelf Key-Value Stores.** State-of-the-art key-value systems are designed for specific workloads. Applications in turn need to choose a system from the limited set of options in terms of core design, more prominently systems that are based on B-trees [4, 10, 12, 47] for read-heavy workloads or on Log-Structured-Merge (LSM)-trees [9, 20, 25, 26, 108] for write-heavy workloads or Log-Structured-Hash (LSH)-tables [43] for systems with large memory. Utilizing such engines for any other workload type does not guarantee good performance [79, 92, 97], as shown in Figure 1. For example, Viber [17] had to switch from MongoDB to Couchbase due to growing datasets [1, 102]; applications need to make a hard choice between either being stuck with a sub-optimal system or an expensive and risky transition [39].

**2. The Choice of Cloud Provider and Hardware.** 77% of organizations face challenges choosing an appropriate cloud provider [112]. Even with a single cloud provider, it is imperative to choose the correct hardware resources to maximize performance and avoid paying extraneous costs. Currently, all these decisions are manually made based on past experience [96, 104, 111] and given the complexity, this often leads to wrong choices with drastic negative impact. For example, a 2022 study [54] illustrates how picking the wrong VM can be a catastrophic mistake.

**Cosine.** We present **Cosine**: a **c**loud-cost **o**ptimized **S**elf-designing key-value storage eng**ine**, that has the ability to self-design and instantiate *holistic configurations* given a workload, a cloud budget, and optionally performance goals and a set of Service Level Agreement (SLA) specifications. A *configuration* is composed of the exact storage engine design in terms of the individual data structures design (in-memory and on-disk) in the engine as well as their algorithms and interactions, a cloud provider and specific virtual machines. Figure 1 depicts the core Cosine concept and how existing systems are "locked" into a small fraction of the possible design space. Cosine is inspired by those systems and makes use of their innovative designs but instead of being locked in any particular design it can mix and match fine-grained storage engine design elements. This creates a space of sextillions ($10^{36}$) of storage engine configurations most of which do not exist in the industry or literature. Cosine automatically takes the perfect shape for the problem at hand making it possible to scale across all three challenges: data size, application (workload) diversity, and cloud budget.

**Our contributions are as follows.**

(1) We formalize the exhaustive search space of key-value storage engine designs comprising of combinations of (a) data structure designs (including LSM-trees, B-trees, LSH-tables, and trillions of valid new designs which are hybrids of those) and in-memory accelerators such as buffer, filters and indexes, (b) hardware for storage (such as HDD, SSD, or EBS) and computation (VMs), and (c) cloud providers (AWS, GCP, Azure) for an input workload (Figure 2 part A, C).
(2) We introduce a unified model that precisely estimates the expected cost of executing a given workload with a given engine design (Figure 2 part B). The model has two novel parts: a) an analytical distribution-aware I/O model that captures data movement across the exhaustive design space of possible key-value storage engines, and b) a learned cost model that captures CPU, query concurrency, and hardware parallelism through a training phase that is kept at minimum cost by selectively training for a few of the possible designs.
(3) We show how for a given workload Cosine collapses the massive possible design space into a Pareto frontier of ranked configurations that co-optimizes available cloud budget, required cloud SLAs, and required performance (Figure 2 part C). This enables choosing the best configuration for the current application, but it also facilitates interactive reasoning, e.g., what if I can afford more budget. Cosine selectively includes noise in the input workload so that the resulting engines are robust to workload drifts.
(4) Using a storage engine code template that allows structured descriptions of Cosine's data layout and algorithmic abstractions, the output of a search is a Rust implementation of the target storage engine design (Figure 2 part D).

| | | | | Design Abstractions of Template | Type/Domain | Example templates for diverse data structures | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | LSM variants | B-Tree variants | LSH variants | A new design |
| LAYOUT PRIMITIVES | Design and hardware specification | *initialized by search through engine design space* | 1. | **Key size:** Denotes the size of keys in the workload. | unsigned int | auto-configured from the sample workload | | | |
| | | | 2. | **Value size:** Denotes the size of values in the workload. All values are accepted as variable-length strings. | string/slice<br>*max size set to 1 GB* | auto-configured from the sample workload | | | |
| | | | 3. | **Size ratio (T):** The maximum number of entries in a block (e.g. growth factor in LSM trees or fanout of B-trees. | unsigned integer \| function (func) | [2,.. 32] | [32, 64, 128, 256, ..] | [1000, 1001, …] (T is large) | 2 |
| | | | 4. | **Runs per hot level (K):** At what capacity hot levels are compacted.<br>Rule: should be less than size ratio. | unsigned int | [1.. T] | | [T-1] | 7 |
| | | | 5. | **Runs per cold level (Z):** At what capacity cold levels are compacted.<br>Rule: should be less than size ratio. | unsigned int | [1.. T] | [1] | | 32 |
| | | | 6. | **Logical block size (B):** Number of consecutive disk blocks. | unsigned int | [2048, 4096, …] | | | |
| | | | 7. | **Buffer capacity** $(M_B)$**:** Denotes the amount of memory allocated to in-memory buffer/memtables. Configurable w.r.t file size. | 64-bit floating point \| function (func) | [64 MB, 128 MB, …] | [1 MB, 2 MB, …] | [64 MB, 128 MB, …] | h/w dependent |
| | | | 8. | **Indexes**$(M_{FP})$**:** Amount of memory allocated to indexes (fence pointers/hashtables). | 64-bit floating point \| function (func) | memory to cover L | memory for first level | memory for hash table | h/w dependent |
| | | | 9. | **Bloom filter memory** $(M_{BF})$**:** Denotes the bits/entry assigned to Bloom filters. | 64-bit float \| func(FPR) | 10 bits/key | | | func(FPR) |
| ALGORITHMIC ABSTRACTIONS | Data access | *derived with empirically verified rules* | 10. | **Bloom filter design:** Denotes the granularity of Bloom filters, e.g., one Bloom filter instance per block or per file or per run. The default is file. | block \| file \| run | file | | | file |
| | | | 11. | **Compaction/Restructuring algorithm:** Full does level-to-level compaction; partial is file-to-file; and hybrid uses both full and partial at separate levels. | partial \| full \| hybrid | full, partial | partial | partial | hybrid |
| | | | 12. | **Run strategy:** Denotes which run to be picked for compaction (only for partial/ hybrid compaction). | first \| last_full \| fullest | first, fullest, last_full | | first | fullest |
| | | | 13. | **File picking strategy:** Denotes which file to be picked for compaction (for partial/ hybrid compaction). For LSM-trees we set default to dense_fp as it empirically works the best. B-trees pick the first file found to be full. LSH-table restructures at the granularity of runs. | oldest_merged \| oldest_flushed \| dense_fp \| sparse_fp \| choose_first | dense_fp | choose_first | | dense_fp (hot), choose_first (cold) |
| | Parallelism | | 14. | **Merge threshold:** If a level is more than x% full, a compaction is triggered. | 64-bit floating point | [0.7..1] | 0.5 | | 0.75 |
| | | | 15. | **Full compaction levels:** Denotes how many levels will have full compaction (only for hybrid compaction). The default is set to 2. | unsigned integer \| function (func) | [1..L] | | | L-Y (from optimal config) |
| | | | 16. | **No. of CPUs:** Number of available cores to use in a VM. | unsigned int | Use all available cores | | | |
| | | | 17. | **No of threads:** Denotes how many threads are used to process the workload. | unsigned int | Use 1 thread per CPU core | | | |

**Table 1: Storage engine template in Cosine and example initializations for diverse storage engine designs.**

(5) We demonstrate that the Cosine cost model captures diverse workloads, hardware, and storage engine contexts with up to 91% accuracy. We verify this using diverse state-of-the-art key-value engines: RocksDB (LSM-tree), WiredTiger (B-tree), and FASTER (LSH-table) as well as with numerous hybrid/new designs with Cosine's templated Rust engine.

(6) We demonstrate that Cosine improves throughput by 53x, 25x, and 20x on average over RocksDB, WiredTiger, and FASTER, over the diverse workloads of the YCSB benchmark and varying cloud budgets and data sizes, while providing a robust behavior that is within 2% of the optimal one even when the workload fluctuates by more than 15%.

**Online Demo and Technical Report.** Cosine is part of the emerging space of **instance optimized data systems** [76]. We focus on key-value stores and use fine-grained first principles that form a massive design space which includes trillions of previously unknown designs out of which we pick the best one for the problem at hand. Given the massive space of possible configurations, to give a better sense of Cosine's potential, we also provide an interactive demo at http://daslab.seas.harvard.edu/cosine/. One can use the demo to perform what-if design questions and compare system designs in terms of budget and performance, for arbitrary workloads across the three major cloud providers and against state-of-the-art systems. Interested readers can also find additional experiments, proofs and model derivations in an online technical report [2].

## 2 BACKGROUND: LAYOUT PRIMITIVES

Cosine has its roots in the Data Calculator project [77] which structures a first-principle driven search of a wide design space of known and unknown data structures. Unlike the Calculator that focuses on a single data structure at a time, Cosine is about complete storage engine configurations considering the interactions of many data structures in a full engine, hardware space, as well as the cloud provider space. To realize the end-goal of self-designing at its core Cosine implements a *storage engine template* – a dictionary of design abstractions to allow structured descriptions of arbitrary key-value storage engine designs. The Cosine template has multiple parts as shown in Table 1. We first presented the data layout part in a vision paper [73] and we give a brief summary of that in this section before continuing with the contributions of this paper.

The Cosine storage engine template spans trillions of possible designs. They are all derived by combining elements from three designs which span the extremes of performance from read to write optimized: B-trees [64], LSM-trees [89], and LSH-tables [43]. The layout primitives in Table 1 help describe key-value storage engines based on the design of their core data structures, in-memory and on-disk. For example, $M_B$, $M_{BF}$, $M_{FP}$ define the memory allocated to the buffer, Bloom filters, and fence pointers. The template provides a lot of flexibility, e.g., it is possible to choose between the construction policies of the filters, e.g., reducing the false positive rate (FPR), optimizing the number of internal hash functions, or even controlling the granularity of filters per block, file, or run.

Based on the memory footprint of filters and fence pointers, Cosine divides $L$ disk-levels of a storage engine into $L - Y$ hot and $Y$ cold levels. Data residing at hot levels are quickly accessed with in-memory filters and fence pointers whereas cold levels have to be accessed through cascading fence pointers on-disk. Other layout primitives include the size ratio (T) denoting the factor by which disk-levels grow in capacity, and merge thresholds (K and Z) denoting how greedily merges happen within a level (hot and cold). In [73] it is explained how these few primitives can be enough by showing there is another set of rules that help derive additional layout design elements of an engine. For example, to figure out exactly how many bits to give to each Bloom filter of each level and run we use the equations from [51, 52]: this only requires knowing $M_{BF}$. For brevity we do not explain these rules further as they are not necessary for following the rest of the paper.

**Describing Existing and New Storage Engines.** Overall, these primitives allow Cosine to take the shape of arbitrary and diverse designs in terms of storage engine data layouts including LSM-Trees, B-Trees, LSH-Tables and several hybrids in between. For example, as shown in Table 1, the layout of LSM-tree based RocksDB is described as $T = 10, K = 1, Z = 1, M_{BF} = 10$, and B-Tree based WiredTiger as $T = 32, K = 1, Z = 1, M_{BF} = 0$. For both engines, the memory footprint of indexes, $M_{FP}$ decides the number of hot and cold levels of the tree. On the other hand, a storage engine such as FASTER which is a flat data structure is described by setting $T = \frac{N.E}{M_B}$ enforcing the first level to never run out of capacity. This also implies that the merge thresholds are set to maximum such that the level is never compacted, i.e. $K = T - 1$, $Z = T - 1$. As FASTER uses in-memory hashtables, we allow $M_{FP}$ to take that into account. By default, FASTER creates hash-bucket entries amounting to $\frac{1}{8^{\text{th}}}$ of the keys, hence $M_{FP} = \frac{N.F}{8} * (1 + \frac{1}{B})$. Table 1 also includes an example of a new design with a log-structured layout at the hot levels and a B-tree at the cold levels.

**Toward a Self-Designing System.** On top of the basic data layouts, for Cosine to achieve its end goal we need to be able to consider additional critical storage engine design components such as the ones shown at the bottom of Table 1 for hardware parallelism and maintainance strategies. In addition, as shown in Figure 2, we need a series of innovations beyond engine specifications such as being able to judge different designs (without implementing them first) (Sec 3), consider the effect of query and hardware parallelism on each design (Sec 4), search over the massive possible space efficiently for the best (yet robust) design given a cloud budget, workload, and SLAs (Sec 5), and finally materialize the code of the resulting storage engine so that it is ready for deployment (Sec 6).

## 3 DISTRIBUTION-AWARE I/O MODELING

Given a workload Cosine needs to evaluate the massive number of possible storage engine designs in a practical way, i.e., without actually running the workload with all possible designs, cloud providers and VMs. For this to be possible Cosine needs to be able to *calculate* the expected performance on a given hardware for any candidate storage engine design. Sections 3 and 4 introduce the Cosine model for I/O and CPU respectively which achieves this with high accuracy. Then Section 5 uses the model to build the search algorithm for the best design given a workload. We start by describing the properties that the model should have.

| | Symbols | Explanation | Cost | Symbols | Explanation | Cost |
|---|---|---|---|---|---|---|
| Workload + Performance | $\theta_r$ | fraction of total lookups | – | $\theta_w$ | fraction of rmws | $\sigma_w$ |
| | $\theta_l.\theta_r$ | fraction of single-result lookups | $\sigma_l$ | $\theta_s$ | fraction of scans | $\sigma_s$ |
| | $(1 - \theta_l)\theta_r$ | fraction of no-result lookups | $\sigma_n$ | $\mathcal{D}_{get}$ | get key distribution | – |
| | $\theta_p$ | fraction of inserts | $\sigma_p$ | $\mathcal{D}_{put}$ | put key distribution | – |
| | $\theta_u$ | fraction of updates | $\sigma_u$ | $\chi$ | queries in workload | – |

| | | |
|---|---|---|
| Cost-Model | $\sigma_n$ | $K\left(\sum_{i=1}^{L-Y-1} p_i\right) + (Y+1)Z$ |
| | $\sigma_l$ (uniform) | $C_0 + \sum_{i=1}^{L-Y-1} p_i\left(\sum_{r=1}^{K} C_{r,i}\right) + \sum_{i=\max(1,L-Y)}^{L}\left(\sum_{r=1}^{Z} C_{r,i}\right)$ |
| | $\sigma_l$ (skew) | $p_{get}\left[C_0^1 + \sum_{i=1}^{L-Y-1} p_i\left(\sum_{r=1}^{K} C_{r,i}^1\right) + \sum_{i=\max(1,L-Y)}^{L}\left(\sum_{r=1}^{Z} C_{r,i}^1\right)\right] +$<br>$(1-p_{get})\left[C_0^2 + \sum_{i=1}^{L-Y-1} p_i\left(\sum_{r=1}^{K} C_{r,i}^2\right) + \sum_{i=\max(1,L-Y)}^{L}\left(\sum_{r=1}^{Z} C_{r,i}^2\right)\right]$ |
| | $\sigma_p$ | $\frac{1}{B}\left(\sum_{i=1}^{L-Y-1} \frac{E_B \cdot T^{i-1}(T/K+1)}{Q_{i-1}}\right) + \frac{1}{B}\left(\frac{E_B \cdot T^{L-Y-1}(T/Z+1)}{Q_{L-Y-1}}\right)$<br>$+ \frac{1}{Q_{L-Y-1}} \sum_{i=L-Y+1}^{L} \min(E_B \cdot T^{L-Y-1} \cdot \frac{\max(1, 1+\min(B-T,T))}{\max(1,B-T)}, \frac{E_B \cdot T^i}{B})$ |
| | $\sigma_u$ | $I_{M_{BF=0}}(\sigma_l + \frac{1}{B}) + I_{M_{BF!=0}}\sigma_p$ |
| | $\sigma_w$ | $I_{M_{BF=0}}\sigma_p + I_{M_{BF!=0}}(\sigma_l + \sigma_p)$ |
| | $\sigma_s$ | $\frac{2s}{B}\left(\sum_{i=1}^{L-Y} E_B \cdot T^i + \sum_{i=L-Y+1}^{L-1} I_{T<B} E_B \cdot T^i + E_B \cdot T^L\right) + I_{T=B}\max(0,\ Y-1)$ |

| | | |
|---|---|---|
| Building Blocks of Model | $p_i$ | Probability of a false positive at a bloom filter at a run at hot level i |
| | $C_0, C_0^1, C_0^2$ | Probability of not being in buffer for uniform key, skew special key, skew regular key |
| | $C_{r,i}, C_{r,i}^1, C_{r,i}^2$ | For a hot level, probability of not being in run r , level i or previous runs for uniform key, skew special key, skew regular key. For a cold level, probability of not being in any node at level i or nodes in previous levels for uniform key, skew special key, skew regular key |

**Table 2: Distribution-Aware I/O Cost Model**

**1. Precision.** The I/O cost estimates should be within a $1 + r$ factor of the actual I/O, where $r$ is a small error parameter, regardless of data properties, data size, query patterns, and engine design.

**2. Conservation.** The model should generate only positive errors that is only over-estimating of I/O costs. This property lowers the chances of exceeding the desired cloud budget or breaching the performance target as a result of estimation errors.

**3. Consistency.** If a storage engine design outperforms another design in practice, this should be reflected in the model estimates.

**Workload Characterization.** Table 2 describes the notation used. Cosine describes workloads as a set of operations over a universe of key-value pairs, using (a) the distribution of the keys and workload operations and (b) the proportion of each operation type: single-result lookups, no-result lookups, range queries, inserts, blind updates (updating the value of an entry regardless of its current value), and read-modify-writes (rmws) (updating the value of an entry based on the current value). This forms a *workload feature vector*. Ideally, we would feed all information about a workload (i.e. the exact sequence of operations and keys-value pairs) into the cost model to get an exact estimate of I/O cost. However, such a strategy would be intractable, since the space of workloads is enormous. Thus, we introduce a low-dimensional "workload summary" to strike a balance between tractability and precision.

**Uniform Distribution.** We let $\mathcal{D}_{get}$ and $\mathcal{D}_{put}$ be distributions over which keys are drawn for reads and writes respectively. In a uniform distribution any possible key is equally likely to be drawn. Hence, $\mathcal{D}_{get}$ is uniform over the universe $U$ of keys and $\mathcal{D}_{put}$ is uniform over the keys that have been inserted/updated. We assume querying keys are drawn i.i.d from these distributions.

**Skewed Distribution.** We use uniform distribution as a building block. The intuition is to partition the key-space of a workload such that (i) for any partition, the keys are uniformly distributed, and (ii) the width of partitions may differ which controls the degree and region of skew. Our definition of a skew distribution exactly matches the definition in [40] and it creates a sharper truncation between hot and cold keys for ease of analysis. For instance, a Zipfian distribution consists of two distinct uniform distributions with universes $U_1$ and $U_2$. $U_1$ contains a set of *special* keys that are likely to be accessed more than the set of remaining *regular* keys residing within $U_2$. More precisely, for lookups, a special key is accessed with probability $p_{get}$ from $U_1$ and a regular key is drawn from $U_2$ with probability $1 - p_{get}$. The same definition holds for writes, with the respective probabilities being $p_{put}$ and $1 - p_{put}$.

**Other Distributions.** The above methodology can be extended to describe other distributions such as normal and exponential distributions. Examples can be seen at the technical report [2]. The core idea is that instead of having two partitions we have $k$, each with its distinct universe spread across the key-space such that the probability of accessing a special and regular key from any partition $i$ is $\frac{p_i}{|U_i|}$ and $\frac{1-p_i}{|U_i|}$, respectively, and that the probabilities to access any partition add up to unity.

**Overview.** The model is shown in Table 2. It shows how Cosine estimates the cost of each type of key-value operation. This is a **unified model** which means that it works across all possible storage engine designs (LSM-trees, B-Trees, LSH-Tables and anything in between) defined by the primitives of Cosine. For a workload of $\chi$ operations, we use the per-operation I/O cost and the proportion of different type of operations in the workload to compute the total I/O cost of the workload (see also definitions in Table 2):

$$\text{IO}_{\text{total}} = \chi(\theta_p\sigma_p + \theta_r\theta_l\sigma_l + \theta_r(1-\theta_l)\sigma_n + \theta_u\sigma_u + \theta_w\sigma_w + \theta_s\sigma_s) \quad (1)$$

While we cannot provide the full details on how the model is derived due to space restrictions, we instead provide in the rest of this section the core intuitions that lead to the model construction using examples for specific storage engine designs and operations. The exact derivations for each operation are described in detail in [2]. A **core insight** is that for all designs supported by Cosine, the disk part of the data layout is effectively built from arrays and pointers connected in a hierarchical format. The cost-model leverages this structure to decompose the I/O cost into inter-dependent per-level quantities and embeds the fundamental read/write behavior of each core design class (LSM-tree/B-tree/LSH-tables) within its cost computation. This allows the resulting unified model to work for any storage engine design possible within the possible space.

**Intuition for Lookup Cost ($\sigma_l$, $\sigma_n$).** Let us assume an LSM-tree-like design and a single-result lookup: when a key is found in the buffer or the first few levels the query terminates and no data from the lower levels is brought into memory. We call this effect *early stopping*, shown in Figure 2 part B. Early stopping is not of much relevance when the distribution of writes is uniform over a large universe as the most recent copies of almost all keys live at the last level. On the other hand, when the read distribution cycles through a small number of keys, as in the case of skew distributions, those keys are very likely to live in the buffer or at a high level, and early stopping can significantly impact the single-result lookup estimate, $\sigma_l$. The distribution-aware model precisely captures the impact of early stopping on $\sigma_l$ as a function of $\mathcal{D}_{put}$ and $\mathcal{D}_{get}$ through the distribution-dependent quantities $C_0$ and $C_{r,i}$, which capture the probability that an access to a given run is attempted. On the other hand, Cosine's models for the no-result lookup cost account for the fact that early stopping does not occur for these queries.

**Intuition for Write Cost ($\sigma_p$, $\sigma_u$, $\sigma_w$).** Cosine captures the phenomenon that with a high proportion of updates merges between any two levels of the tree happen less frequently since an updates is handled in place at the level or buffer it is found. We call this effect *infrequent merging*, shown in Figure 2 part B. Infrequent merging is less prominent when the distribution of writes is uniform over a large universe, where almost all writes are insertions. On the other hand, when the write distribution cycles through a small number of keys many of the writes are updates, leading to infrequent merging. The model is able to capture the impact of infrequent merging on $\sigma_p$ as a function of $\mathcal{D}_{put}$ through the distribution-dependent quantities $Q_i$, which estimate the expected number of writes to fill a run in level $i$. Rmws are modeled as a composition of other operations, e.g., for LSMs, the cost of an rmw is the summation of a lookup and an insert whereas for B-trees or LSH it is same as an update.

**Intuition for Range Cost ($\sigma_s$).** For modeling range queries, key distribution is not needed because the effect of early stopping does not occur: range queries need to access every hot level. With a selectivity of $s$, for hot levels, roughly $s$ fraction of the entries at each level will be touched using the in-memory fence pointers. For internal nodes at cold levels, Cosine differentiates between the case where the size ratio $T$ equals to the block size $B$ ($T = B$) and where it does not ($T < B$) which impacts whether data at internal nodes needs to be scanned as well. For the last cold level, Cosine's models account for touching all relevant leaf nodes.

**Derivation of Lookup Cost $\sigma_l$ for Skew.** We give a sketch for one of the derivations as an example: for a single-result lookup following skew distribution when a design only has hot levels, i.e., $Y = 0$. We can express the expected cost as $\sigma_p = p_{get} \times \sigma_{ls} + \sigma_{ln} \times (1 - p_{get})$ where $\sigma_{ls}$ is the expected cost of a special key lookup and $\sigma_{ln}$ is the expected cost of a normal key lookup. Given that the skew distributions are uniform over special keys, it suffices to consider a generic special key $k$ and its expected cost over the randomness of $\mathcal{D}_{put}$. The cost is the sum of the I/O cost of accessing the disk block containing the key and the I/Os due to bloom filter false positives. If the key is in the buffer or the block cache, the I/O cost is 0, otherwise it is 1. We show that, $C_0^1$ captures this cost. For the false-positive incurred costs, by linearity of expectation, it suffices to compute the probability that a block in a given run will be touched and then add up the probabilities. The expression $p_i C_{r,i}^1$ precisely captures this cost for run $r$ at level $i$, where $p_i$ is the probability of obtaining a false positive and $C_{r,i}$ is the probability that the actual key is not in the current run or any previous runs (so the access has not terminated yet). The argument is similar for $\sigma_{ln}$.

Now, we provide intuition for the specific formulas for $C_0$ and $C_{r,i}^1$. To compute these quantities, we determine the distributions over queried special keys as uniform over $U_1 \cap K_{\text{special}}$ (where $K_{\text{special}}$ is the set of keys in the data structure). This can be viewed as the conditional distribution of $U_1$ conditioned on the key being in $K_{\text{special}}$. Therefore, the probability that a key $k$ is not in any of the runs up to a given run is, $\frac{P[k \in K_{\text{special}}, k \text{ is not in any runs up to a given run}]}{P[k \in K_{\text{special}}]}$.

We now use the independence of keys across different runs and condition on the data structure being full. We add “weights” to designate that runs are likely to be a certain fraction full “on average”. The numerator can be expressed as, $\left(\prod_{\text{current run}}^{\text{up to}} P[k \text{ is not in the run}]\right) \times (1 - P[k \text{ is in a later run}])$. For a run $r$ at level $i \geq 1$, the first term is $(1 - \alpha^{0,l})(1 - \alpha^{BC,l}) \times \left(\prod_{h=1}^{i-1}(1 - \alpha^{h,l})^K\right)(1 - \alpha^{i,l})^r$. Similarly, the second term is $1 - (1 - \alpha^{i,l})^{K-r}\left(\prod_{h=i+1}^{L-Y-1}(1 - \alpha^{h,l})^K\right)$. In these expressions, $\alpha^{i,l}$ captures the probability that $k$ appears in a given run at level $i$. The calculation of $\alpha^{i,l}$ can be obtained from skew properties of $\mathcal{D}_{put}$.

## 4 CONCURRENCY-AWARE CPU MODELING

In addition to I/Os, performance of storage engines is significantly affected by CPU costs and hardware parallelism. Writing analytical models, as we did for I/O in the previous section, that capture in-memory and hardware effects is extremely complicated and error prone, even for a single design, as we have found in our prior work [82]. The challenge is that there are many factors that affect CPU performance and are tightly connected compared to the single factor of I/O when data comes from disk. Cosine uses learning in a hardware-conscious way, as shown in Figure 2, part B.

**Utilizing Amdahl’s Law.** Amdahl’s Law [24, 67, 68] theoretically reasons about how much speedup we can achieve for a given number of cores. Every program has a sequential component and a parallelizable component and with more cores, it is only the runtime of the parallelizable component that improves. If $\phi$ proportion of a program is parallelizable and it takes $T$ time units to execute it with 1 core, then for $\eta$ cores, the speedup $g$ is as follows:

$$g = \frac{T}{T - \phi T + \frac{\phi T}{\eta}} = \frac{1}{1 - \phi(1 - 1/\eta)} \tag{2}$$

**Learning $\phi$.** Cosine takes the value of $\eta$ directly from the hardware where the engine is to be deployed. On the other hand, $\phi$ is learned as it relies on many interconnected factors. While the possible designs are too many, we observed that $\phi$ has similar values across designs that share core design elements and so Cosine only needs to learn $\phi$ for four drastically distinct design classes (LSM, LSH, B-Tree, and Hybrid: LSM-like hot levels and B-tree-like cold levels). The process is seen in Algorithm 1. For each class of design $r$ and for each operation type in $q$, we benchmark (using the Rust code template discussed in Section 6) the speedup as we increase the number of queries executed in parallel and CPU cores used (one per query). This is done for all distinct VM types $v$ for each cloud provider. The observed speedup $\hat{g}$ is fed to Equation 2 to generate multiple values of $\phi$ ($\phi_{q,r,v,D,c}$) for different VMs, numbers of total queries, and data size, to aggregate and derive a robust $\phi$ for this combination of $q$ and $r$. Then, for any workload $W$ to run with a design of class $r$, we calculate $\phi$ as a weighted average of the $\phi$ of each operation type in $W$. Then, the end-to-end latency and throughput of running $W$ on a given VM, while maximizing utilization of $\eta$ cores, are given by combining Equations 1 and 2:

$$\text{latency} = \frac{\text{IO}_{\text{total}}}{\text{IOPS}} \times \frac{1}{g}, \text{throughput} = \frac{\#\text{operations in } W}{\text{latency}} \tag{3}$$

where IOPS is a cloud-provider specific constant (Sec 5). All steps for learning happen independently so that there are no modeling errors due side-effects of one process on another.

**Algorithm 1** Algorithm for learning $\phi$

```
Input: $S$, Output: $\phi^{s_i}_{q,r}$ // value of $\phi$ for op $q$ on class $r$ for $s_i$
$R$ : {LSM, BTree, LSH, Hybrid}
$\chi$ : {lookup, insert, blind update, rmw, range}
for each provider $s_i \in S$ do
    for each design class $r \in R$ do
        for each op type $q \in \chi$ do
            // distinct combinations of $(r, q)$ benchmarked
            $\Phi^{s_i}_{q,r}$ = NULL
            for each VM $v$ of distinct type do
                for data $D \in$ [1M..100M] do
                    // varying data-h/w combinations
                    populate Cosine with $D$ on layout $r$
                    for op count $oc \in$ [1M..10M] do
                        // using actual speedup in Amdahl's law
                        for $c \in [1..v^{\text{CPU}}]$ do
                            set $c$ as query parallelism
                            $T(c)$ = run $oc$ ops of type $q$ on $c$ cores
                            $\hat{g} = \frac{T(c)}{T(1)}$
                            get $\phi_{q,r,v,D,c}$ using Eq (2)
                            $\Phi_{q,r} = \Phi_{q,r} \cup \{\phi_{q,r,v,D,c}\}$
            $\phi^{s_i}_{q,r}$ = average of values in $\Phi_{q,r}$
function getPhiForWorkload($W$, $r$, $s_i$)
    $\phi^{s_i}_{W,r} = \sum_{o=1}^{\text{all op types in} W} \phi^{s_i}_{o,r} \times$ proportion of o in W
```

**Training Cost.** $\phi$ depends on the hardware. Training only for four design classes as opposed to for every possible design reduces the cost by several orders of magnitude for Cosine. For each VM, learning can still take several hours though. To reduce the cost further, we observe that VMs overlap in hardware properties and thus we only need to train for a small subset of VMs (AWS alone has more than 100 VMs). For example, m5zn.large and m5n.large are AWS instances with a similar configuration (2 vCPUs, 8 GB memory, EBS-only storage, 25 Gbps network bandwidth) with the only difference in EBS bandwidth [18]. Such differences translate to marginal impact on $\phi$ at the third or fourth places of decimal, which plays a negligible role in the cost-performance optimization. We curate a list of distinct VM types and Cosine ships already with trained $\phi$ values for those while similar VMs use these $\phi$ values [2].

**Changes in Pricing Policy or VM Specifications.** Cloud providers frequently update their pricing models [99] but this does not require any retraining as Cosine only needs to pull the new prices. Cloud providers may also add new VMs types [83]. If a new VM offers distinct hardware properties than all VMs Cosine has trained for in the past, then Cosine has to train on this VM and make it part of its learned $\phi$ library but this is a one time operation. When training does need to happen, Cosine can train in parallel for every distinct VM; this saves time, not cloud cost.

## 5 SEARCHING FOR THE BEST DESIGN

We now use the models of the previous two sections to describe how Cosine searches for the best storage engine design given a workload $W$, a desired cloud budget $b$, and optionally performance requirements (latency/throughput) $pr$. The output is a storage engine design (expressed in the primitives of Table 2), specific VM and cloud provider choices, along with the expected cloud cost and performance to run $W$ with the resulting engine and VMs.

**Hardware Space.** The hardware on which a storage engine is deployed drives the cloud cost and performance. For any input workload-budget combination, Cosine constructs the space of possible hardware configurations (VMs and storage) for the set of all cloud providers $S$, as shown to the left of Figure 2 part C. Cosine uses a range of cloud costs $C = [c_{\min}, c_{\max}]$ such that the input budget $b$ falls within that range - we explain in a few paragraphs why we need that range. VMs are discretely priced per time unit and hence, there cannot exist a unique hardware configuration for each $c \in C$. To prevent including redundant configurations in the hardware space, Cosine incrementally adds resources to generate hardware configurations that have a distinct combination of VMs and storage and keeping only these costs in $C$. For each cloud provider $s_i$, computing resources are combinations of VM instances of $k_i$ distinct types. Every VM $v_{i,j}$, indicating the $j^{\text{th}}$ VM instance of $s_i$, contains $v_{i,j}^{\text{mem}}$ GB of memory and $v_{i,j}^{\text{CPU}}$ vCPU cores. Storage can be attached externally to VMs and determine the number of I/O operations per second, $v_{i,j}^{\text{IOPS}}$. For any $c \in C$, there are multiple deterministic possibilities of combining storage and compute resources and that varies with the pricing policy of each cloud provider, Amazon [3], Azure [11] and Google Cloud [8]. The unification of all of these possibilities for all cloud costs in $C$ makes up the hardware space.

**Storage Engine Design Space.** For each VM in the candidate hardware space, a storage engine design space is constructed. A storage engine design is mathematically represented using the data layout primitives from Table 1 as $\Omega$: $(T, K, Z, M_B, M_{BF}, M_{FP}, \eta)$ where $\eta$ denotes the number of physical cores ($\eta \in \{1..v_{i,j}^{\text{CPU}}\}$). Given the input workload $W$, the distinct possibilities of allocating memory across buffers, bloom filters, and fence pointers using the pricing policy of $s_i$ for every cost in $C$, is $\Omega^{W,s_i,C}_{M_B \times M_{BF} \times M_{FP}}$. Then, the design space over all cloud providers for cost range $C$ is,

$$\Omega^{W,S,C} = \Omega^{W}_{T \times K \times Z \times \eta} \times \cup_{s_i \in S} \Omega^{W,s_i,C}_{M_B \times M_{BF} \times M_{FP}} \tag{4}$$

**Performance Space.** For every candidate storage engine design $\Omega \in \Omega^{W,S,C}$ Cosine computes the expected latency for $W$ using the models of Sections 3 and 4. This results in the performance space $P$.

**The Overall Search Space.** A massive space of *configurations*, $\Delta^{W}_{C,\Omega,P}$ is generated. It consists of ordered triples $(\Gamma, c, p)$ where $\Gamma$ denotes a configuration comprising of a storage engine design, a hardware, and a cloud provider combination that yields performance $p$ in terms of latency and needs cloud cost $c$ to run $W$.

**The Cardinality of the Search Space.** Given the number of distinct VM types $k_i$ offered by provider $s_i$, for all $c \in C$, this leads to a set, $H_{s_i}$ of VM combinations. Every combination is of the form $< \lambda_{i,1}, \lambda_{i,1}, \cdots, \lambda_{i,k_i} >$, where $\lambda_{i,j}$ determines the number of instances of VM type $j$ that can be purchased. Therefore, for a total of $m_i$ combinations under a single provider, we have, $H_{s_i} = \{< \lambda_{i,1}, \lambda_{i,1}, \cdots, \lambda_{i,k_i} >^q\}$, $1 \leq q \leq m_i$. For each VM, $T$, $K$, $Z$, and $\eta$ possess an integral domain space, whereas the domain space of memory allocated across buffers, bloom filters, and fence pointers is non-integral. For navigating through the memory space, Cosine uses "*memory hopping*" setting $M_B$ to a small value and then incrementing it by a fixed amount equal to $\epsilon$ fraction of the total memory $M$. Therefore, the cardinality of storage engine designs possible within a single VM is $|\Omega^{W,s_i,c}| = T \times K \times Z \times \eta \times \frac{1}{\epsilon}$. The cardinality of designs over $k_i$ VM types that can be purchased with a given $c$ is $(T \times K \times Z \times \frac{1}{\epsilon})^{k_i}$. If $m_i$ distinct configurations result from different cost values in $C$, the cardinality of all possible configurations with provider $s_i$ is $m_i \times (T \times K \times Z \times \eta \times \frac{1}{\epsilon})^{k_i}$. For example, with \$$50K$ monthly budget only for one provider (AWS) and only with 6 distinct VMs, and even if we statically assign the highest degrees of parallelism to $\eta$, we have $m_i = 74612$, $T \times K \times Z \times \frac{1}{\epsilon} = 30752$ ($\epsilon = 0.2$). This leads to a total design space $74612x(30752^6)$.

**Cost-Performance Optimization.** Cosine needs to solve two optimization problems to find the best storage engine design that minimizes cloud cost and latency $l$.

$$\underset{(\Gamma,c,p) \in \Delta^{W}_{C,\Omega,P} \text{ such that } p \leq l}{\operatorname{argmin}} (c), \quad \underset{(\Gamma,c,p) \in \Delta^{W}_{C,\Omega,P} \text{ such that } c \leq b}{\operatorname{argmin}} (p) \tag{5}$$

**Range of Cloud Budgets and Interactive Design.** We cannot be certain that any combination of desired performance requirement $pr$ and budget $b$ is possible, i.e., that indeed there exists a storage engine design, a set of VMs and a cloud provider that can achieve performance $pr$ with $b$ on $W$. Thus, Cosine searches simultanously not only for the best configuration for $b$ and $pr$ but also for neighboring values of $b$ and $pr$. This is why we need the cloud range defined in the Hardware Space paragraph. By default, we set this range from \$1-\$500,000 per month, but it is also exposed as a knob. This range covers the monthly budgets of diverse real-life applications, e.g., early- to mid-stage tech startups [19, 57, 86]. If $b$ does not fall within this range, Cosine updates the range to \$1-\$($b$+500,000). If the desired performance cannot be achieved, both the fastest configuration with the desired budget and the cheapest configuration with the desired performance are included in the result. Furthermore, Cosine enables a what-if design process where designers can search and explore alternative storage engine designs and budget/performance balances in an interactive way with instance system responses. Interested readers may visit https://cloud-demo-2021.github.io/ to interact with Cosine directly.

**Cost-Performance Continuum.** To enable all of the above, for every design search session Cosine collapses the engine design space on the cost-performance plane (given $W$, $b$). This transforms the trilateral tradeoff among engine designs, hardware and cloud cost, into a cost-performance Pareto frontier. This is a continuum 1) with an optimal configuration at every point, and 2) where a higher cloud cost maps to better or at least the same performance. The process of generating the Pareto frontier is shown in Algorithm 2. We provide a stepwise description below.

**Step 1: Partially-Pruned Configurations for Each Provider.** We first construct the performance space of each provider $s_i$ for a single cloud cost $c$ in $C$ at a time. For each storage engine configuration in $\Lambda \in H_{s_i}$, Cosine shards the workload and data using off-the-shelf sharding algorithms [53] across all VMs of the configuration proportionally to their memory capacity. For each distinct VM type $v_{i,j}$ within $h$, Cosine generates the intra-VM design space specific to its workload shard by spanning through all combinations of $T$, $K$, $Z$, $\eta$, and using memory hopping for $M_B$, $M_{BF}$, and $M_{FP}$. Further, for each possible value of $M_B$, Cosine splits the residual memory between $M_{BF}$, and $M_{FP}$ to maximize as many hot levels as possible for the design. Once all designs specific to the VM type are generated, Cosine uses the I/O model to predict the I/O cost of these designs. Next, it looks up the learned coefficients (obtained

**Algorithm 2** The Cost-Performance Optimization

```
**Input**: $D$, $W$, $S$ **Output**: Cost-performance continuum ($F$)
budget$^{\min}$ = $\infty$, budget$^{\max}$ = 0
**for all** $s_i \in S$ **do**    // for every cloud provider
    Generate $H_{s_i}$
    cost$_{\text{storage}}$ = getStorageCost($s_i$, $D$) // storage cost of config
    **for** all configurations $\Lambda \in H_{s_i}$ **do**    // searching through several VM configs. Each config may have instances of different VM types
        $M_\Lambda = \sum_{j=1}^{k_i} v_{i,j}^{\text{mem}}, \forall v_{i,j} \in \Lambda$ // total memory of the configuration
        cost$_{\text{compute}}$ = $\sum_{j=1}^{k_i} \lambda_{i,j} \times$ monthly_price($v_{i,j}$) // compute cost of config
        cost$_{\text{SLA}}$ = getSLACost($D$, $s_i$, $\Lambda$) // SLA cost of config, shown in [2]
        cost$_{\Lambda,i}$ = cost$_{\text{compute}}$ + cost$_{\text{storage}}$ + cost$_{\text{SLA}}$ // cost to afford the config
        budget$^{\min}$ = min(budget$^{\min}$, cost$_{\Lambda,i}$)
        budget$^{\max}$ = max(budget$^{\max}$, cost$_{\Lambda,i}$)
        Shard W into $(W_1, W_2, \cdots, W_{|k_i|})$ s.t. #ops in $W_j \propto \frac{\lambda_{i,j}.v_{i,j}^{\text{mem}}}{M_\Lambda}$
        Add 10% noise to $W$ using noise model [2]   // for robustness
        **for** all VM types $v_{i,j}$ in $\Lambda$ and $\lambda_{i,j} > 0$ **do**    // get optimal design for each distinct VM type of the configuration
            $M_B = \epsilon.M$, latency$_{v_{i,j}}^{\min} = \infty$
            **while** $T \in [2..B_i]$ **do**    // size ratio    // navigating through the storage engine design space
                **while** $K \in [1..T-1]$ **do**    // hot merge threshold
                    **while** $Z \in [1..T-1]$ **do**    // cold merge threshold
                        **while** $M_B \leq M$ **do**    // buffer
                            Set $\Omega : (T, K, Z, M_B, M_{BF}, M_{FP}, v_{i,j}^{\text{CPU}})$
                            Get no. of levels $L$ using [74]
                            Find min($Y$) s.t. $\sum_{l=1}^{L-Y}(M_{BF}^l + M_{FP}^l) \leq (M - M_B)$
                            IO$_{v_{i,j}}$ = get IOs using Equation (1)  // I/O model
                            Find design class $D$ of $\Omega$ and get $\phi$ of $W$ on $D$
                            Compute speedup $g_{v_{i,j}}$ with Equation (2)
                            latency$_{v_{i,j}}$ = $\frac{\text{IO}_{v_{i,j}}}{\text{IOPS}_{s_i}} \cdot \frac{1}{g_{v_{i,j}}}$   // latency from I/Os
                            **if** latency$_{v_{i,j}}$ < latency$_{v_{i,j}}^{\min}$ **then**
                                latency$_{v_{i,j}}^{\min}$ = latency$_{v_{i,j}}$
                                update optimal design for $v_{i,j}$ with $\Omega$
                            Increment $M_B$ by $\epsilon.M$
            build tuple $F_{\Lambda,i,j}$: $(\Omega, v_{i,j}, W_j, \text{latency}_{v_{i,j}}^{\min})$
            $F_{\Lambda,i} = F_{\Lambda,i} \cup F_{\Lambda,i,j}$
            latency$_{\Lambda,i}$ = max(latency$_{v_{i,j}}$) $\forall v_{i,j} \in \Lambda$
    $F_{s_i} = F_{s_i} \cup (F_{\Lambda,i}, \text{latency}_{\Lambda,i}, \text{cost}_{\Lambda,i})$   // constructing the cost-perf mapping for each provider
$F = F \cup F_{s_i}$
**for** $c \in$ [budget$^{\min}$..budget$^{\max}$] and all $s_i \in S$ **do**   // constructing the Pareto
    Find $F_{\Lambda,i}$ s.t. argmin$_{\text{cost}_{\Lambda,i} \leq c}$(latency$_{\Lambda,i}$)
```

from Algorithm 1 of concurrency model) specific to the design class of each design and the operations in $W$. Using these coefficients, Cosine computes the overall $\phi$ of workload $W$, derives the speedup using Equation 2, and computes the predicted end-to-end latency using Equation 3. Once all designs specific to $v_{i,j}$ are evaluated, Cosine ranks them and picks the one with the minimum end-to-end latency for this VM. Cosine repeats this across the VM types of $h$ and for all configurations within $H_{s_i}$. At this step, we have partially pruned inferior configurations within intra-VM design space but we neither have a (i) continuum yet as every cloud cost may still map to multiple performance points emanating from different configurations nor (ii) a Pareto frontier because all performance points are locally optimized for a certain cloud cost, i.e., increasing cloud cost does not necessarily map to equal or better performance.

**Navigating the Non-Integral Domain Space of Memory.** With memory hopping Cosine considers designs around the non-integral memory space of a VM, left and right of the best $M_B$ value so far (by a fixed amount equal to $\epsilon$ fraction of the total memory $M$) using binary search. Cosine examines the expected performance of the new $M_B$ values (and derived $M_{BF}$, and $M_{FP}$) with the previously obtained values of $T$, $K$, and $Z$ of the best design so far. Then, the memory allocation with the best resulting performance is chosen. With smaller $\epsilon$ the width of the adjacent hop regions shrinks at the cost of increasing the number of different buffer values to be checked. Cosine exposes $\epsilon$ as a tuning parameter and adopts a default value of 0.1 which we find provides consistently a good balance among search time and quality of results (shown in [2]).

**Step 2: Generating Continuum for Each Provider.** Cosine further prunes configurations by iterating over all performance points for all $c \in C$ and $s_i \in S$. We partition the resulting mapping from Step 1 into as many equal-sized disjoint partitions as the number of cores of the machine where we run the cost-performance optimization. We prune redundant configurations within each partition to generate sub-continuums where a configuration $\Lambda$ mapped to cloud cost $c$ is pruned if (i) there is at least one configuration $\Lambda'$ at $c$ that is better than $\Lambda$ or (ii) Cosine has already seen a better configuration $\Lambda'$ that offers a lower latency at a lower cost $c'$ in which case it augments $\Lambda'$ as a valid configuration at $c$. After this process finishes for all cloud costs within every partition, Cosine examines the junction points of each partition to ensure Pareto optimality. Let the current state of the cost-performance mapping be denoted as $\Delta$ partitioned into $k$ partitions $\Delta_1, \Delta_2, \cdots \Delta_k$, then we ensure that for any two consecutive partition $\Delta_i$ and $\Delta_{i+1}$, the best (or the rightmost) latency point of $\Delta_i$ is greater than or equal to the worst (or the leftmost) latency point of $\Delta_{i+1}$, i.e.,

$$\min_{\forall c \in \Delta_i} f(c) \geq \max_{\forall c \in \Delta_{i+1}} f(c), \forall 0 < i < k - 1 \qquad (6)$$

If this condition fails, (i) we identify all points in $\Delta_{i+1}$ that have latency worse than that of the best latency point of $\Delta_i$ and (ii) update these points to $\min_{\forall c \in \Delta_i} f(c)$ thereby guaranteeing the property of increased (or at least equal) performance with increased cost. We repeat this step for each pair of consecutive partitions for each provider. Thus, we obtain three different continuums which are Pareto frontiers (one for each provider) where each cloud cost maps to exactly one performance point.

**Step 3: Generating Continuum Across Providers.** Cosine iterates once again over all the cloud costs in $C$ and ranks these configurations at each cost to generate a single cost-performance continuum optimized across all providers, as also indicated in Figure 2 part C. At this point, we keep all three configurations from different providers and do not prune any as it helps probing the continuum for provider-specific questions that may arise as part of Cosine's what-if interactive reasoning.

**Cloud Service Level Agreements.** Cosine takes as an optional input Cloud SLA requirements. SLAs are provisioning and monitoring services that guarantee a threshold level of cloud service quality [45, 66, 78]. Cosine supports five SLA features – (i) *DB migration* [28, 32, 63], (ii) *operational and tooling support* [31, 34, 60], (iii) *backup* [13–15], (iv) *reliability* [29, 33, 61], and (v) *availability* [30, 35, 62]. SLAs are mathematically quantified and exposed as computable pricing models. If any SLA is required, then every cloud cost $c \in C$ is co-optimized for purchasing of hardware and SLAs. As seen in Algorithm 2, Cosine ensures that (a) the combined price of hardware and SLAs never exceeds $c$ and (b) all resource-SLA permutations are considered.

**Algorithm 3** Mapping layout to algo abstractions

```
    Input: $\Omega, W, s_i$ Output: initialized primitives
if $\Omega \rightarrow M_{BF}! = 0$ then                    ⎤
    if $\Omega \rightarrow Y > 0$ then                      ⎥ // Bloom filters + cold levels
        class = "Hybrid"                                    ⎦
    else
        class = "LSM"   // Bloom filters + no cold levels
else if $\Omega \rightarrow T > B$ then                     ⎤ // T is so large that
    class = "LSH"                                           ⎦ there is only 1 level
else
    class = "BTree"
$\phi$ = getPhiForWorkload($W$, class, $s_i$)
set algo primitives based on class and $\phi$
```

**Robustness.** The actual workload may vary from the input workload $W$. To ensure robust designs Cosine adds "noise" by changing the proportion of operations in $W$ and by adding missing operations. This is set by default to 10% of noise which we find to give consistent results across numerous diverse workloads [2].

**Result: Navigating the Pareto Frontier.** Creating the continuum takes on average 40-50 seconds on our test machine (Sec. 7) for all three cloud providers. Once the continuum is constructed, Cosine can instantly navigate it using a binary-search to generate the optimal configuration $\Gamma_{\text{optimal}}$ (storage engine design, VMs, and provider) for the available cloud budget and to also suggest neighboring configurations with attractive budget-performance balances. Finally, the continuum enables what-if capability: designers can interactively query Cosine to get the optimal configuration for any budget or performance point (given $W$).

## 6 RUST CODE TEMPLATE

Once the search process terminates, Cosine uses the resulting design to setup the code for the target storage engine. Cloud logistics have to be addressed a priori, e.g., creating and setting up cloud contracts. Cosine includes a templated key-value engine which consists of Rust library crates for every component of the storage engine template in Table 1. Each crate containerizes the structure and capacity of a storage component and also offers its own set of Rust `Traits` that define how it can be *created* or accessed for *reads* and *writes*, *garbage collection*, *capacity* checks, and *inter-crate interactions*. Cosine also maintains a wrapper crate for the entire storage engine the traits of which directly link to that of the buffer and the main tree. By using the values of the layout primitives in the target design Cosine initializes the code template.

In addition to the layout primitives, Cosine's engine template contains a set of *algorithmic abstractions*. Each algorithmic abstraction not only indicates a core functionality of the storage engine but also controls the granularity at which the functionality is induced within the engine. Table 1 shows all such primitives and their definitions. For example, `<restructuring strategy>` denotes how data is restructured (B-trees), merged (LSH-tables), or compacted (LSM-trees) across diverse designs. Based on the resulting engine design from the search process, Cosine determines which algorithmic primitives are the best fit. For instance, the absence of Bloom filters ($M_{BF} = 0$) means that the design is not in the LSM class. If the class is LSM, merging starts from the first run of a level and so Cosine sets `run_strategy = full` whereas in B-trees, run and level are synonymous as merges happen at the granularity of files (`run_strategy = none, file_picking_strategy = choose_first`). Similarly, for concurrency, Cosine uses the learned coefficient ($\phi$) of the workload on the chosen design class to set the degrees of parallelism to the point where Equation 2 converges or speedup does not improve by adding more cores. Algorithm 3 shows the primary rules to setup the template given a data layout.

## 7 EXPERIMENTAL EVALUATION

We now demonstrate the self-designing ability of Cosine. First, we verify Cosine's cost model with diverse storage engine designs, workloads and cloud budgets. Then, we show that Cosine scales with data, workload diversity, and cloud budget, outperforming state-of-the-art engines by up to an order of magnitude.

**Baselines.** We compare Cosine against three state-of-the-art storage engines of diverse designs, RocksDB (LSM-tree) [56], WiredTiger (B-tree) [120], and FASTER (LSH-table) [43]. We set each of these baselines to their default configuration. As RocksDB does not support the existence of cold levels, we allocate as much memory for fence pointers as needed to cover to all disk-resident blocks. We use the default size ratio ($T = 10$) and the Bloom filter policy with 10 bits per key. For WiredTiger, we set the size of the leaf nodes and the internal nodes to the default of 32KB and 4KB, respectively. We use the default fanout of 32 ($T = 32$). For FASTER, we assign the default memory to accommodate $\frac{1}{8^t h}$ keys in the hashtable. For all baselines, we allocate 1 GB to the in-memory buffer.

**Workloads.** We extend the standard key-value benchmark, YCSB [46]; we cover all core YCSB workloads A-F, but we also test with several variations with different distributions such as, uniform, zipfian, and normal corresponding to diverse real-life applications such as graph processing in social media [27, 59], web applications [119] and event log processing [42]. We include the workloads which are known to be favorable for the baselines, i.e., write-heavy workloads which favor FASTER [43], range query heavy workloads which favor WiredTiger [82], and mixed workloads which favor RocksDB [56]. We experiment with 100M records each of length 1024 bytes (128 for integer keys & 896 for values).

**Cloud Pricing.** We use 7 VMs from each cloud provider: `r5` from AWS, `n1-highmem` from GCP, and `E2-64 v3` from Azure. The VMs have diverse CPU and memory properties with hourly rates ranging from \$0.09 to \$4.36 in AWS, \$0.07 to \$3.57 in GCP, and \$0.07 to \$2.50 in Azure. For SSD storage, we set \$0.1 per GB-month for AWS (beyond 75 GB which is a free slab) and \$0.24 per GB-month for GCP with maximum allowed throughput set at $15K$ IOPS and $30K$ IOPS, respectively. Azure employs flat storage prices for different storage slabs from \$5.28 to \$259.05 for storage slabs ranging from 32 GB (120 IOPS) to 2 TB (7500 IOPS), respectively. We use the publicly available quotations of the three cloud providers [3, 8, 11].

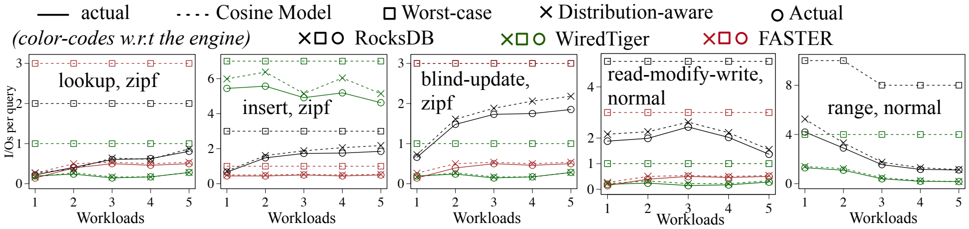
**Hardware.** For computing Cosine's storage engine configurations, we use a machine with Core i5 processor and 16GB DDR4 RAM.

### 7.1 Verifying the Accuracy of Cosine's Models

Cosine's potential as a self-designing storage engine depends on the accuracy of its cost models across the entire space of storage engine designs, hardware, data, operations, and cloud budgets.

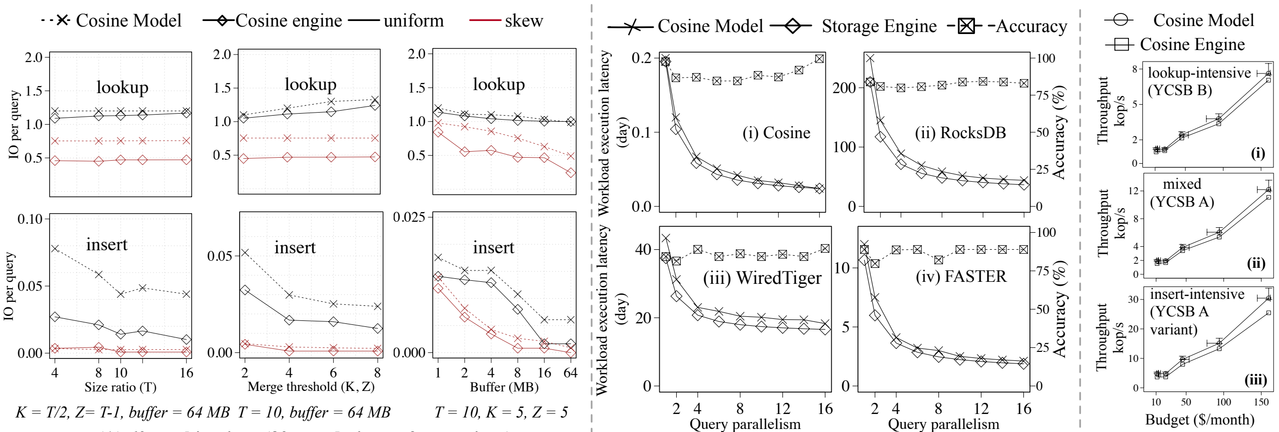
| Workload ID | Zipfian |  | Normal |  |
|---|---|---|---|---|
|  | p_put | p_get | $\mu$ | $\sigma$ |
| 1. | 0.8 | 0.8 | 2M | 1000 |
| 2. | 0.5 | 0.8 | 3M | 10000 |
| 3. | 0.2 | 0.8 | 5M | 10000 |
| 4. | 0.5 | 0.5 | 5M | 1000 |
| 5. | 0.2 | 0.2 | 5M | 100 |

**(A) Distribution parameters used for (B)**



**(B) 75 combinations with existing designs (5 workloads x 5 operations x 3 storage engines)**

**Figure 3: Accurately capturing the cost of running diverse operations on existing storage engine designs.**



**(A) 60 combinations (30 new designs x 2 operations)** **(B)** **(C)**

**Figure 4: Accurately predicting I/O, CPU, and cloud cost with query and h/w parallelism for diverse new and existing designs.**

**Step 1: State-of-the Art Engines.** First, we show that Cosine can accurately capture the I/Os of diverse engine designs for all core key-value operations. For each operation we compare the actual performance of each engine - we measure the mean incurred I/Os - against the predicted I/Os by the Cosine model, using the initializations of Table 1 for each engine. We test with 100M entries, 10M queries (for each operation type) and various skewed distributions varying $\mathcal{D}_{get}$ and $\mathcal{D}_{put}$ to represent zipfian and normal distributions as shown in Figure 3(A). Figure 3(B) shows that as we vary workloads and distributions, (1) the shape of the predicted I/Os matches that of the actual I/Os for all engines and distributions and (2) the error is extremely small with an average accuracy of 89%, 86%, and 93% for RocksDB, WiredTiger, FASTER, respectively. For a better understanding of the need for distribution-aware models, we also compare against the worst case models we introduced [73] for the data layout primitives. The Cosine model is drastically superior, e.g., for zipfian lookups with $p_{get} = 0.8$ or normally distributed rmws on FASTER or WiredTiger, most of the keys are in cache or buffer thereby leading to less I/Os per operation, whereas, the worst-case model still accounts at least 1 I/O for each lookup.

**Step 2: New/Hybrid Engines.** Next, we verify the model against several new engine designs which are hybrids of Cosine's core design classes. We generate 30 new engine designs by varying one design primitive at a time. To test the performance of these new engines, we deploy the Rust implementation of the Cosine template. Similar to the observations in Figure 3(B), Figure 4(A) shows an average accuracy of 94% in predicting the performance of diverse storage engines across two of the core key-value operations.

**Step 3: Concurrency-Aware CPU Modeling.** We now switch from individual operations to full-fledged workloads and move on from I/Os to end-to-end latency using also Cosine's learned CPU model. Figure 4(B) depicts results using YCSB A with 25% of each operation: lookups, inserts, blind updates, and rmws, and as we vary query parallelism and CPU cores. It shows results with all baselines as well as with Cosine's optimal auto-designed engine. The results include both the actual runtimes and the predicted performance by the Cosine model. For all degrees of parallelism in Figure 4(B) and for all engine designs, Cosine's prediction accuracy never falls below 82% and achieves a maximum of 90%.

**Step 4: Cloud Cost.** We now demonstrate that Cosine can accurately translate performance estimations to cloud cost. We use Azure and compare the cloud cost and performance estimations of Cosine with the actual monetary cost incurred and the throughput achieved when running the codebase produced by Cosine on the cloud. We use five VM types: B1|s, B1s, B1ms, A1_v2, and D1_v2 and a YCSB A workload variant. We run this experiment on a distributed setup with a cluster size of 3 VMs. Then, the cloud cost varies from $11.4/month to $159.87/month [11, 16] for running the same workload with the optimal configurations in each case. Figure 4(C) shows that the averaging over all budget points and YCSB A variants, the predicted performance is accurate to up to 91%. Depending on the availability of VMs, there can be a variability of up to 15% on cloud-cost for the same performance (each VM is

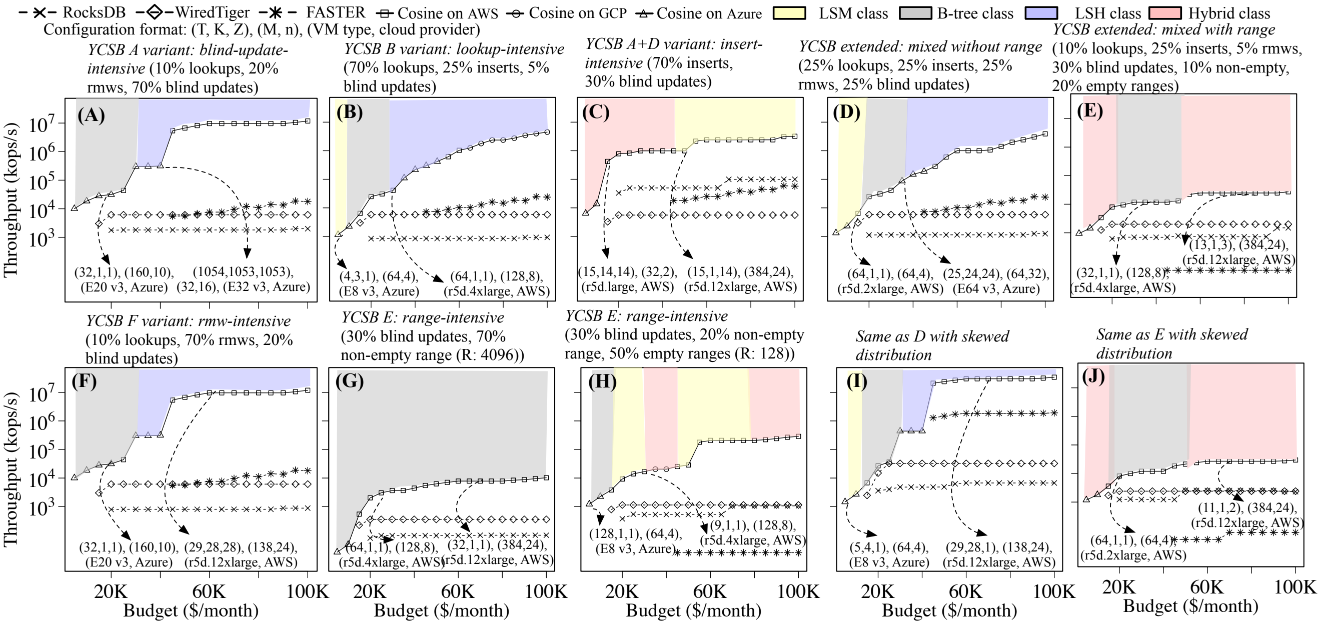


**Figure 5: Cosine outperforms existing storage engine across diverse workloads and cloud budgets.**

charged differently across different Azure data centers). To include this variability, we plot with error bars the minimum and maximum price of each VM type across 6 data centers in the United States.

**Overall** the results in this section show that the Cosine model satisfies all quality properties: (1) **precision** as it maintains an average accuracy of 89% across storage engine designs, operations, and distributions, (2) **consistency** as the shape of estimation matches that of the actual curve in all experiments, and (3) **conservation** as in almost all scenarios, we notice errors due to over-estimation and not under-estimation. Cosine never exceeds the budget by more than 15% or misses the performance target by more than 12%.

## 7.2 Outperforming State-of-the-art Engines

Next we demonstrate that given a workload and budget, Cosine can search over a massive space of possible storage engine designs to find and take the shape of the best design that significantly outperforms state-of-the-art systems while also choosing the best cloud provider and hardware (VM) for deployment to balance both cloud cost and performance. Now that we have proven the accuracy of the Cosine models, we use the models to generate the results of these experiments as we would otherwise need a budget in the order of 8 million dollars. We verify the cost-performance mapping for each provider using their price calculators [5–7].

**Scaling with diverse Workloads and Cloud Budgets.** Figure 5 shows results from ten workloads. (A)-(C) and (F)-(H) are workloads with a one dominant query type whereas (D)-(E) and (I)-(J) are mixed workloads with varied composition of operations, as indicated on top of each graph. For each experiment, we vary the budget in ($/month) and for each budget, we plot the throughput for RocksDB, WiredTiger, FASTER, and Cosine (best design chosen for each budget/workload). We also explicitly show some of the resulting Cosine designs and the dominant class of the chosen Cosine design at each point by color coding (e.g., B-tree based). The resulting cloud provider is indicated by the point used for each budget and we set AWS as the default cloud provider for the baselines.

For all workloads in Figure 5, Cosine either outperforms or matches RocksDB, WiredTiger, and FASTER. Cosine's performance gain grows with more budget because hardware and design are not co-optimized in fixed engines while Cosine morphs to the best design to co-optimize for the workload, budget, and available hardware. For example, by capturing early-stopping and infrequent merging within its distribution-aware cost model and by finding out the optimal level of parallelism with its learned model, Cosine is able to allocate memory across the buffer, bloom filters, and fence pointers optimally. For instance, as the budget increases for mixed workloads (5(D)), Cosine's optimal configuration uses the same memory footprint but switches the storage engine design from B-tree-like to LSH-like and the cloud provider from AWS to Azure. This is because (i) unlike B-trees, LSH-tables are fundamentally read-write-optimized at the cost of high memory and (ii) memory is cheaper in Azure, compared to AWS. Even with a reduced IOPS rate in Azure, the LSH-Azure combination emerges as the optimal one for higher budgets. Cosine also brings out observations that enhance conventional wisdom. For instance, Cosine pickes B-tree class of designs as the workload consists of range-intensive operations (5(G)). However, when the workload changes such that a significant proportion of those operations are empty (5(H)), then Cosine chose the LSM-class of design by utilizing an in-memory range filter (Rosetta [90]). Making such choices manually is extremely complex and non-intuitive as conventional wisdom suggests that LSM designs are not optimized for range queries.

**Scaling With Data.** We now show that Cosine scales better not only with budget and workloads but also with data. In this experiment, data grows from 1 to 100 billion entries; for every 10%

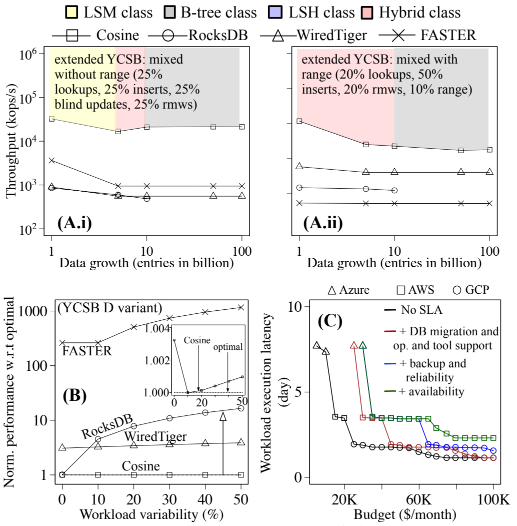


**Figure 6: Cosine scales (A); is robust (B); adapts to SLAs (C).**

growth in data we increase the budget by $10K to accommodate additional storage and computation requirements. We test for two YCSB variants. Figure 6(A.i) shows that Cosine scales better by up to 80X, 40X, and 7X compared to RocksDB, WiredTiger, and FASTER respectively. This is because when searching for the optimal engine, Cosine's synthesis assumes that the underlying data structures in the candidate designs are full. This allows Cosine to (i) eliminate configurations that are likely to grow disproportionately (or proportionately) for uniform (or skewed) workloads and (ii) pick configurations that amortize the cost over large data. When workloads have range queries as in Figure 6(A.ii), the benefit is slightly smaller as a range query involves reading multiple pages which dominate cost; Cosine still improves by 40x and 15x over RocksDB and WiredTiger, respectively.

**Overall**, across all workloads, budgets and data sizes in Figures 5 and 6, Cosine improves on average by 53x, 25x, and 20x compared to RocksDB, WiredTiger, and FASTER, respectively and for many contexts the benefit can be be up to 100x. For certain points across all figures, there are no numbers for RocksDB, WiredTiger, or FASTER. This is due to insufficient budget. For example, with AWS, RocksDB needs to always use at least $20K/month to maintain good performance by keeping its bloom filters in memory (with 10 bits/entry). Similarly, FASTER needs at least $45K/month for good performance to be able to hash about $\frac{1}{8^{\text{th}}}$ of the data in-memory which requires 7400 GB of memory ($\frac{10^{12}}{8} * 64 * (1 + \frac{1}{2048})$ with $B = 2048$) which is 21 VMs of type $r5d.12x$large in AWS. Instead, Cosine is able to find a viable storage engine design for any budget.

**Cosine is Robust.** We now show that Cosine provides robust performance if the expected workload varies from the actual workload. Here Cosine expects a YCSB D variant workload with 50% lookups and 50% inserts and is given a budget of $20K per month. The actual workload that arrives includes $10xV\%$ new operations: 45% lookups, 45% inserts, and $3.33xV\%$ blind updates, $3.33xV\%$ rmws, and $3.33xV\%$ range queries. We vary the actual workload by either altering the proportion of operations ($V$) and track the performance loss by testing against the optimal design that Cosine would create if it was given the actual workload. Figure 6(B) shows that Cosine can accommodate up to 40% of workload changes in composition of operations. It has a negligible performance loss of 0.3%, whereas RocksDB and WiredTiger incur a loss of 16x and 3x, respectively. FASTER incurs a 100x loss due to range queries.

**SLAs.** Finally, Figure 6(C) shows that as we add SLA requirements, Cosine utilizes the budget to maximize perfomance by coming up with new engine designs and cloud configurations.

## 8 RELATED WORK

Cosine is part of an ongoing effort in the systems community to design systems that learn and adapt: Learned systems [75, 84, 85], adaptive indexing [74], and learned-tuning [100, 110] are all related areas of research but orthogonal directions. First, Cosine offers a drastically different methodology by exhaustively utilizing our knowledge as system designers to craft an extensive design space for the core design decisions of storage engines and uses both analytical and learned cost models to rank those designs. This brings fast and robust design space navigation to achieve self-designing properties. Second, Cosine focuses specifically on key-value storage engines [70, 105, 106, 114, 121, 124] and co-designs the whole self-designing process with cloud cost policies and deployment.

Bourbon [49] is a modification of WiscKey [88] that replaces the fence pointers in an LSM-tree based key-value store with a learned model. Distributed key-value stores can also use learned components for purposes such as caching hot key-value data closer to the client for less network cost [118], and responsively adjusting in-memory component tuning to changing workload conditions [110, 122]. Learned components are orthogonal to Cosine which offers a framework to model the whole key-value store engine space. For example an LSM-tree system such as Bourbon that replaces an existing component with a learned component that does the same work is still an LSM-tree system with the same core properties. Instead, Cosine's first principles approach gives the ability to span design classes and offers scalability in workload diversity. Learned components can be integrated into the Cosine template and models so that they can be considered as part of the engine design space.

**Other Related Areas.** In [2], we discuss the related areas of multi-node cloud cluster management [21, 25, 37, 65, 81, 123], cloud resource pricing, utilization and guarantees [38, 69, 94, 113], and other data management work in cost optimization [22, 44, 50, 107, 109].

## 9 LIMITATIONS & OPPORTUNITIES

Given the vast space of considerations with storage engines there are several limitations and open topics we do not cover in this paper but offer exciting areas for future work towards full blown self-designing storage engines such as workload forecasting, distributed processing, on-the-fly adaptation, adoption of learned components, and even extending to data models beyond the key-value paradigm. For example, in the spirit of online database tuning [41] online adaptivity for self-designing storage engines can utilize the Cosine models to reevaluate the design periodically and then there is the additional problem of transitioning the engine design with minimum data movement. Our early results for online-adaptivity show promising signs across diverse designs [80, 91].

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

# REFERENCES

[1] 2014. Viber Replacing MongoDB with Couchbase. https://www.youtube.com/watch?v=mMuMAjgXWIc.

[2] 2019. http://daslab.seas.harvard.edu/cosine/appendix.pdf. , Cosine Technical Report pages.

[3] 2019. Amazon Web Services. https://aws.amazon.com/ec2/pricing/on-demand/.

[4] 2019. Aria Storage Engine. http://mariadb.com/kb/en/library/aria-storage-engine.

[5] 2019. AWS Calculator. https://calculator.s3.amazonaws.com/index.html.

[6] 2019. Azure Calculator. https://azure.microsoft.com/en-us/pricing/.

[7] 2019. GCP Calculator. https://cloud.google.com/products/calculator/.

[8] 2019. Google Cloud Pricing. https://cloud.google.com/compute/all-pricing.

[9] 2019. Influx. https://influxdata.com.

[10] 2019. InnoDB. https://dev.mysql.com/.

[11] 2019. Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.

[12] 2019. PostgreSQL. https://www.postgresql.org.

[13] 2020. AWS Backup pricing. https://aws.amazon.com/backup/pricing/.

[14] 2020. Azure Backup pricing. https://azure.microsoft.com/en-us/pricing/details/backup/.

[15] 2020. Cloud Storage for data archiving. https://cloud.google.com/storage/archival.

[16] 2020. General purpose Azure VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-general.

[17] 2020. Viber. https://www.viber.com/en/.

[18] 2021. Amazon EC2 Instance Types. (2021).

[19] 2021. How much are startups spending for their top needs? *T. C. Brand Studio* (2021).

[20] 2021. Riak KV. https://docs.riak.com/riak/kv/latest/index.html.

[21] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated Data Placement for Geo-distributed Cloud Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, Berkeley, CA, USA, 2–2.

[22] 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 ACM SIGMOD International Conference on Management of Data*. 1009–1024.

[23] Amazon. 2020. Cloud Storage. *https://aws.amazon.com/what-is-cloud-storage/* (2020).

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

[25] Apache. 2020. Cassandra. *http://cassandra.apache.org* (2020).

[26] Apache. 2020. HBase. *http://hbase.apache.org/* (2020).

[27] 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.

[28] AWS. 2019. AWS Database Migration Service pricing. https://aws.amazon.com/dms/pricing/.

[29] AWS. 2019. CloudEndure Disaster Recovery. https://aws.amazon.com/marketplace/pp/Amazon-Web-Services-CloudEndure-Disaster-Recovery/B073V2KBXM.

[30] AWS. 2019. Prior Version(s) of Amazon EC2 Service Level Agreement - Not Currently In Effect. https://aws.amazon.com/ec2/sla/historical/.

[31] AWS. 2019. What is DevOps? https://aws.amazon.com/devops/what-is-devops/.

[32] Azure. 2019. Azure Database Migration Service pricing. https://azure.microsoft.com/en-us/pricing/details/database-migration/.

[33] Azure. 2019. Azure Site Recovery pricing. https://azure.microsoft.com/en-us/pricing/details/site-recovery/.

[34] Azure. 2019. Pricing for Azure DevOps. https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/.

[35] Azure. 2019. SLA for Virtual Machines. https://cloud.google.com/functions/sla.

[36] Microsoft Azure. 2019. .

[37] Nicholas Ball and Peter Pietzuch. 2013. Skyler: Dynamic, Workload-Aware Data Sharding across Multiple Data Centres. (2013).

[38] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. The Price Is Right: Towards Location-Independent Costs in Datacenters. ACM HotNets.

[39] F. Brazeal. 2017. Why Amazon DynamoDB isn't for everyone. https://read.acloud.guru/why-amazon-dynamodb-isnt-for-everyone-and-how-to-decide-when-it-s-for-you-aefc52ea9476.

[40] J. Bruck, J. Gao, and A. Jiang. 2006. Weighted Bloom Filter. In *In Proceedings of the International Symposium on InformationTheory (ISIT). 2304–2308*.

[41] Nicolas Bruno, Surajit Chaudhuri, and Gerhard Weikum. 2018. Database Tuning Using Online Algorithms. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer.

[42] 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)*.

[43] 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*.

[44] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*. 34–43.

[45] M. Cooney. 2016. 10 best cloud SLA practices. https://www.networkworld.com/article/3053920/10-best-cloud-sla-practices.html.

[46] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 143–154.

[47] Couchbase. 2020. Online reference. *http://www.couchbase.com/* (2020).

[48] CouchDB. 2020. Online reference. *http://couchdb.apache.org/* (2020).

[49] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 155–171.

[50] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 666–679.

[51] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94.

[52] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 16:1–16:48.

[53] 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. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.

[54] Ditto. 2022. How do you choose the right VM size in Azure? *Accubits* (2022).

[55] DSM. 2018. AWS and Azure: Offering Cloud and Confusion. https://www.dsm.net/it-solutions-blog/aws-and-azure-offering-cloud-service-and-pricing-confusion.

[56] Facebook. 2020. RocksDB. *https://github.com/facebook/rocksdb* (2020).

[57] V. Finkle. 2015. This Startup Let Us Snoop Through Its Finances. Here's What We Found. (2015).

[58] N. Fisk. 2019. Opinion: Clearing up multi-cloud confusion. https://www.cloudcomputing-news.net/news/2019/apr/12/opinion-clearing-multi-cloud-confusion/.

[59] 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*.

[60] GCP. 2019. DevOps. https://cloud.google.com/devops/.

[61] GCP. 2019. Disaster Recovery Planning Guide. https://cloud.google.com/solutions/dr-scenarios-planning-guide.

[62] GCP. 2019. Google Cloud Functions Service Level Agreement (SLA). https://cloud.google.com/functions/sla.

[63] GCP. 2019. Pricing for Migrated Workloads. https://cloud.google.com/migrate/compute-engine/pricing.

[64] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010).

[65] Bram Gruneir. 2017. *Scalable SQL Made Easy: How CockroachDB Automates Operations* .

[66] D. Hein. 2019. 5 Things to Look For in a Cloud Service Level Agreement. https://solutionsreview.com/cloud-platforms/5-things-to-look-for-in-a-cloud-service-level-agreement/.

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

[68] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (July 2008), 33–38.

[69] Darrell Hoy, Nicole Immorlica, and Brendan Lucier. 2016. On-Demand or Spot? Selling the Cloud to Risk-Averse Customers. In *Proceedings of the 12th International Conference on Web and Internet Economics - Volume 10123* (Montreal, Canada) *(WINE 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 73–86.

[70] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. *CoRR* abs/2104.01305 (2021). arXiv:2104.01305

[71] Kecheng Huang, Zhiping Jia, Zhaoyan Shen, Zili Shao, and Feng Chen. 2021. Less is More: De-amplifying I/Os for Key-value Stores with a Log-assisted LSM-tree. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 612–623.

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

[73] 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)*.

[74] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[75] 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*.

[76] Stratos Idreos, Tim Kraska, and Umar Farooq Minhas. 2021. A Tutorial Workshop on Learned Algorithms, Data Structures, and Instance-Optimized Systems. In *VLDB*.

[77] 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.

[78] WIRED INSIDER. 2011. Service Level Agreements in the Cloud: Who cares? https://www.wired.com/insights/2011/12/service-level-agreements-in-the-cloud-who-cares/.

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

[80] Varun Jain, James Lennon, and Harshita Gupta. 2019. LSM-Trees and B-Trees: The Best of Both Worlds. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1829–1831.

[81] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. ACM, New York, NY, USA, 654–663.

[82] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 715–730.

[83] A. Kicinski and H. Souiri. 2019. Forecasting Future Amazon Web Services Pricing. In *ICEAA Professional Development & Training Workshop*.

[84] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *Biennial Conference on Innovative Data Systems Research (CIDR)*.

[85] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 489–504.

[86] M. Lahn. 2019. How much does a server cost for app hosting? (2019).

[87] J. Liang and Y. Chai. 2021. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1032–1043.

[88] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 133–148.

[89] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (Jul 2019), 393–418.

[90] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. 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, 2071–2086.

[91] Graham Lustiber. 2017. E-Tree: An Ever Evolving Tree for Evolving Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Student Research Competition*. 13–15.

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

[93] Metafilter. 2010. The cloud might run me dry. https://ask.metafilter.com/148869/The-cloud-might-run-me-dry.

[94] Jeffrey C. Mogul and Lucian Popa. 2012. What We Talk About when We Talk About Cloud Network Performance. *SIGCOMM Comput. Commun. Rev.* 42, 5 (Sept. 2012), 44–48.

[95] MongoDB. 2020. Online reference. *http://www.mongodb.com/* (2020).

[96] NordicBackup. 2018. 10 Mistakes Companies Make When Choosing Cloud Computing Providers. https://nordic-backup.com/blog/10-mistakes-choosing-cloud-computing-providers/.

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

[98] R. Padilha, E. Fynn, R. Soule, and F. Pedone. 2016. Callinicos: Robust Transactional Storage for Distributed Data Structures. In *In Proceedings of the USENIX Annual Technical Conference (USENIX ATC '16)*.

[99] C. Parlette. 2018. 7 Ways Cloud Services Pricing is Confusing. (2018).

[100] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[101] Google Cloud Platform. 2019. .

[102] D. D. Preez. 2014. Viber migrates from MongoDB to Couchbase halves number of AWS servers. https://diginomica.com/viber-migrates-mongodb-couchbase-halves-number-aws-servers.

[103] J. Ralph. 2019. Which Cloud is Best – AWS vs GCP. https://www.wirehive.com/thoughts/which-cloud-is-best-aws-vs-gcp/.

[104] RapidValue. 2018. How to Choose between Azure/AWS/GCP for Cloud Web Development? https://www.rapidvaluesolutions.com/comparison-criteria-choose-azure-aws-gcp-cloud-web-development/.

[105] 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.

[106] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. In *Proceedings of the VLDB Endowment*.

[107] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 23–34.

[108] SQLite4. 2020. Online reference. *https://sqlite.org/src4/* (2020).

[109] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, MichaelStonebraker, David J. DeWitt, Marco Serafini, and Ashraf Aboulnaga andTim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. In *PVLDB*, Vol. 12.

[110] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. IBTune: Individualized Buffer Tuning for Large-Scale Cloud Databases. *Proc. VLDB Endow.* 12, 10, 1221–1234.

[111] Twain Taylor. 2019. Oracle cloud digs in for a long hard battle against AWS. http://techgenix.com/oracle-cloud/.

[112] R. Tkatchuk. 2017. If the cloud is so great, why are so many businesses unsatisfied? https://www.cio.com/article/3163967/if-the-cloud-is-so-great-why-are-so-many-businesses-unsatisfied.html.

[113] Duong Tung Nguyen, Long Bao Le, and Vijay Bhargava. 2018. Price-based Resource Allocation for Edge Computing: A Market Equilibrium Approach. *IEEE Transactions on Cloud Computing* PP (06 2018), 1–1.

[114] Tobias Vinçon, A. Bernhardt, Ilia Petrov, and Andreas Koch. 2020. NKV in Action: Accelerating KV-Stores on NAtive Computational Storage with NEar-Data Processing. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2981–2984.

[115] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Wanzeng Fu, Beng Chin Ooi, and Pingcheng Ruan. 2018. ForkBase: An Efficient Storage Engine for Blockchain and Forkable Applications. arXiv:1802.04949 [cs.DB]

[116] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. 2020. MotherNets: Rapid Deep Ensemble Learning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 199–215.

[117] Abdul Wasay and Stratos Idreos. 2021. More or Less: When and How to Build Convolutional Neural Network Ensembles. .

[118] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 117–135.

[119] 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.

[120] WiredTiger. 2020. Source Code. *https://github.com/wiredtiger/wiredtiger* (2020).

[121] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling Tiered Cloud Storage in Anna. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 624–638.

[122] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. 2020. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 603–615.

[123] Fan Yang, Youmin Chen, Youyou Lu, Qing Wang, and Jiwu Shu. 2021. Aria: Tolerating Skewed Workloads in Secure In-memory Key-value Stores. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1020–1031.

[124] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *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, 1601–1615.