Technical Report for Cosine

ACM Reference Format:
. 2019. Technical Report for Cosine. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 DEFINING INTERMEDIATE PARAMETERS OF I/O COST MODEL

In this section, we describe the distribution-aware I/O cost model that Cosine uses. We begin by stating the expressions used in the model, and in the next section, we prove the desired expressions capture the fundamental terms of the model.

We determine the number of entries living in a run at a given level i and denote it by R^i. The number of entries residing in buffer is expressed as R^0 as we consider the in-memory buffer to be level 0 of the data structure. R^0 is the same as E_B that indicates the number of entries in buffer. For any hot level i with K runs, where $1 \leq i \leq L-Y-1$, the number of entries in a run can be obtained as the total number of entries in that level divided uniformly across each run. For a size ratio of T, level i contains a maximum of $E_B \cdot T^i$ entries. Therefore, we have, $R^i = \frac{E_B \cdot T^i}{K}$. We indicate the number of runs in cold levels by Z and a portion of each disk-resident blocks (each block can hold upto B entries) is set aside for holding the cascading fence pointers pointing to runs at the next level. Thus, for cold levels, we have, $R^i = \frac{E_B \cdot T^i}{Z} \cdot \frac{B-T}{B}$ and for the last level, we have $R^i = \frac{E_B \cdot T^i}{Z}$ as the last level does not need any reservation for holding cascading pointers to the next level. We summarize the expression in Equation 1.

$$R^i = \begin{cases} E_B, & \text{if } i = 0. \\ \frac{E_B \cdot T^i}{K}, & \text{if } 1 \leq i \leq L-Y-1. \\ \frac{E_B \cdot T^i}{Z} \cdot \frac{B-T}{B}, & \text{if } L-Y \leq i \leq L-1. \\ \frac{E_B \cdot T^i}{Z}, & \text{if } i = L. \end{cases} \tag{1}$$

We begin by defining the necessary parameters for the lookup cost. For uniform distribution, we define C_0 and $C_{r,i}$. The building blocks of the parameters are the α^i values, which captures the probability that a key k appears in a run (or as data in a node) at level i, and α^{BC}, which captures the probability k is in the block cache. We show in the next section that $\alpha^i = \min(1, \frac{R^i}{|U|})$ and $\alpha^{BC} = \min(1, \frac{E_{BC}}{|U|})$. We define the next four probabilities $q, c, c_{r,i}, d_{r,i}$ in terms of these building blocks. These building blocks will be explained in the next section.

$$q = 1 - (1-\alpha^0)\left(\prod_{i=1}^{L-Y-1}(1-\alpha^i)^K\right)\left(\prod_{i=L-Y}^{L}(1-\alpha^i)^Z\right)$$

$$c = (1-\alpha^{BC})(1-\alpha^0)\left[1 - \left(\prod_{i=1}^{L-Y-1}(1-\alpha^i)^K\right)\left(\prod_{i=L-Y}^{L}(1-\alpha^i)^Z\right)\right]$$

$$c_{r,i} = (1-\alpha^0)(1-\alpha^{BC})\left(\prod_{h=1}^{i-1}(1-\alpha^h)^K\right)(1-\alpha^i)^r \left(1-(1-\alpha^i)^{K-r}\left(\prod_{h=i+1}^{L-Y-1}(1-\alpha^h)^K\right)\left(\prod_{h=L-Y}^{L}(1-\alpha^h)^Z\right)\right)$$

$$d_{r,i} = (1-\alpha^{BC})(1-\alpha^0)\left(\prod_{h=1}^{L-Y-1}(1-\alpha^h)^K\right)\left(\prod_{h=L-Y}^{i-1}(1-\alpha^h)^Z\right) (1-\alpha^i)^r\left(1-(1-\alpha^i)^{Z-r}\left(\prod_{h=i+1}^{L}(1-\alpha^h)^Z\right)\right).$$

These building blocks fit together to form $C_0 = c/q$, $C_{r,i} = c_{r,i}/q$ for $1 \leq i \leq L-Y-1$, and $C_{r,i} = d_{r,i}/q$ for $i > L-Y-1$.

Next, we define the analogous distribution-dependent parameters C_0^l and $C_{r,i}^l$ for the skew distribution, where $l = 1$ represents a special key and $l = 2$ represents a regular key. The building blocks are similarly $\alpha^{i,1} = 1 - \left(1 - \frac{p_{put}}{|U_1|}\right)^{R^i}$, $\alpha^{i,2} = \min(1, \frac{R^i(1-p_{put})}{|U_2|})$, $\alpha^{BC,1} = 1 - \left(1 - \frac{p_{get}}{|U_1|}\right)^{E_{BC}/B}$, and $\alpha^{BC,2} = \min(1, \frac{E_{BC}(1-p_{get})}{|U_2|})$. We analogously define the four probabilities $q^l, c^l, c_{r,i}^l, d_{r,i}^l$ in terms of these building blocks:

$$q^l = 1 - (1-\alpha^{0,l})\left(\prod_{i=1}^{L-Y-1}(1-\alpha^{i,l})^K\right)\left(\prod_{i=L-Y}^{L}(1-\alpha^{i,l})^Z\right)$$

$$c^l = (1-\alpha^{BC,l})(1-\alpha^{0,l})\left[1 - \left(\prod_{i=1}^{L-Y-1}(1-\alpha^{i,l})^K\right)\left(\prod_{i=L-Y}^{L}(1-\alpha^i(l))^Z\right)\right]$$

$$c_{r,i}^l = (1-\alpha^{0,l})(1-\alpha^{BC,l})\left(\prod_{h=1}^{i-1}(1-\alpha^{h,l})^K\right)(1-\alpha^{i,l})^r \left(1-(1-\alpha^{i,l})^{K-r}\left(\prod_{h=i+1}^{L-Y-1}(1-\alpha^h(l))^K\right)\left(\prod_{h=L-Y}^{L}(1-\alpha^{h,l})^Z\right)\right)$$

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$d^l_{r,i} = (1-\alpha^{0,l})\left(\prod_{h=1}^{L-Y-1}(1-\alpha^{h}(l))^K\right)\left(\prod_{h=L-Y}^{i-1}(1-\alpha^{h,l})^Z\right)$$
$$(1-\alpha^{BC,l}(1-\alpha^{i,l})^r\left(1-(1-\alpha^{i,l})^{Z-r}\left(\prod_{h=i+1}^{L}(1-\alpha^{h,l})^Z\right)\right)$$

These building blocks fit together to form $C^l_0 = c^l/q^l, C^l_{r,i} = c^l_{r,i}/q^l$ for $1 \le i \le L-Y-1$, and $C^l_{r,i} = d^l_{r,i}/q^l$ for $i > L-Y-1$.

Next, we define the distribution-dependent parameters Q_i for the update cost. The building block of these parameters is the function family $f(V,M) = V \cdot \log\left(\frac{V}{V-M+1}\right)$, which we explain in the next section. This is closely related to Q_i, the number of queries needed to fill up a level, as follows. For a uniform distribution, we let Q_0 be $\max(R^0, f(U,R^0))$ and Q_i be $\max(R^i \cdot K, K \cdot f(U,R^i))$ for $i \neq 0$. When the distribution is skew, the bound is a bit more complex. We define $g_1(0) = f(U_2+U_1, R^0)$ and $g_1(i) = K \cdot f(U_2+U_1, R^i)$ for $i \neq 0$. We define $g_2(i)$ to be $-\infty$ if $R^i \le U_1$. If $R^i > U_1$, then we define $g_2(0) = f(U_2, R^0 - U_1)/(1-p_{put})$ and $g_2(i) = K \cdot f(U_2, R^i - U_1)/(1-p_{put})$. Now, we define Q_0 to be $\max(R^0, g_1(0), g_2(0))$ and for $i \neq 0$, Q_i is defined to be $\max(R^i \cdot K, g_1(i), g_2(i))$ for skew distributions.

2 DERIVATIONS OF THE I/O COST OF QUERIES

In this section, we present the detailed mathematical derivation of the I/O cost for every query type – single-result lookups, writes, no-result lookups, and range queries.

2.1 Single-Result Lookups

For single-result lookups, let $\mathscr{C}$ be the random variable for the cost of a single-result lookup of a given key k. Observe the support of $\mathscr{C}$ is in $\mathbb{Z}^{\ge 0}$. This cost can be decomposed into $\mathscr{C}_0 + \mathscr{C}_{other}$. Here, $\mathscr{C}_0$ is the binary random variable (support in $\{0,1\}$) for the I/O cost of accessing the actual record (which we know exists since this is a single-result lookup). The random variable $\mathscr{C}_{other}$ is a random variable with support in $\mathbb{N}$ for the number of other blocks that are touched (from cascading fence pointers or bloom filter false positives). By the setup of the in-memory helpers and data structure, we know that at most one block in each run at each level is touched. Thus, we can decompose $\mathscr{C}_{other}$ into $\sum_{i=1}^{L-Y-1}\sum_{r=1}^{K}\mathscr{C}_{r,i} + \sum_{i=L-Y}^{L}\sum_{r=1}^{Z}\mathscr{C}_{r,i}$, where $\mathscr{C}_{r,i}$ is a binary random variable that keeps whether a block at run r and level i is accessed for anything except for the actual record. Thus, we can write,

$$\mathscr{C} = \mathscr{C}_0 + \sum_{i=1}^{L-Y-1}\sum_{r=1}^{K}\mathscr{C}_{r,i} + \sum_{i=L-Y}^{L}\sum_{r=1}^{Z}\mathscr{C}_{r,i} \tag{2}$$

Using linearity of expectation (which applies to dependent random variables), we know that,

$$\mathbb{E}[\mathscr{C}] = \mathbb{E}[\mathscr{C}_0] + \sum_{i=1}^{L-Y-1}\sum_{r=1}^{K}\mathbb{E}[\mathscr{C}_{r,i}] + \sum_{i=L-Y}^{L}\sum_{r=1}^{Z}\mathbb{E}[\mathscr{C}_{r,i}] \tag{3}$$

There are two sources of randomness: (a) the bloom filters, and (b) the data structure layout, i.e., which keys reside in which run. We use the fact that the bloom filters have independent randomness from the data structural layout randomness (which is purely determined the randomness over put queries, $\mathcal{D}_{put}$). Thus, $\mathbb{E}[\mathscr{C}_{r,i}] = p_i \cdot C_{r,i}$ where $C_{r,i} := \mathbb{E}[\mathscr{C}'_{r,i}]$ is the expectation of the binary random variable $C'_{r,i}$ which is 1 if a bloom filter for run r and level i is accessed *or* if a block in run r and level i is accessed, and is 0 otherwise. If no bloom filter exists, then $C_{r,i} := \mathbb{E}[\mathscr{C}_{r,i}]$. Let $C_0 := \mathbb{E}[\mathscr{C}_0]$. We see that this means that:

$$\mathbb{E}[\mathscr{C}] = C_0 + \sum_{i=1}^{L-Y-1} p_i \sum_{r=1}^{K} C_{r,i} + p_i \sum_{r=1}^{Z} C_{r,L-Y} + \sum_{i=L-Y+1}^{L}\sum_{r=1}^{Z} C_{r,i} \tag{4}$$

Now, we can simply take an expectation over the randomness in the get distribution to arrive at the expressions in the main paper: when $\mathcal{D}_{get}$ is uniform, we know that $\mathbb{E}[\mathscr{C}]$ is independent of the key k, so we obtain $C_0 + \sum_{i=1}^{L-Y-1} p_i \sum_{r=1}^{K} C_{r,i} + p_i \sum_{r=1}^{Z} C_{r,L-Y} + \sum_{i=L-Y}^{L}\sum_{r=1}^{Z} C_{r,i}$. When $\mathcal{D}_{get}$ is skew, the cost is p_{get} times the expected cost of a special key access plus $1-p_{get}$ times the cost of a normal key access, as shown in the Table 2 of the main paper.

Now, we need to compute C_0 and each $C_{r,i}$ value for a given key k. Note that $C_0 = \mathbb{E}[\mathscr{C}_0]$, for $i \le L-Y$, $C_{r,i} = \mathbb{E}[\mathscr{C}'_{r,i}]$, and for $i > L-Y$, $C_{r,i} = \mathbb{E}[\mathscr{C}_{r,i}]$. It is to be noted that by considering a single-result lookup, we are implicitly *conditioning* on the event that key k being in the data structure. So, when evaluating these expectations, we are really considering random variables that live on the conditional probability space where the sample space is the event that k is in the data structure. To evaluate the conditional probabilities, we use the fact that for a random variable X and an event E, $\mathbb{P}[X=1 \mid E] = \frac{\mathbb{P}[X=1 \text{ and } E]}{\mathbb{P}[E]}$. Now, we formally define the building blocks $q, c, c_{r,i}, d_{r,i}$ from the previous section. We assume q be the probability that key k is in the data structure. We let c be the probability that k is not in memory and k is actually on disk. (Note: in this paper, we are making the assumptions that if $Y > 0$, then $Z = 1$). We denote $c_{r,i}$ to be the probability that the bloom filter at run r at level i (for $i \le L-Y-1$) is accessed for something other than key k and that key k lives later in the data structure. We denote $d_{r,i}$ as the analogous probability for $i > L-Y-1$. Using these building blocks, we express $C_0 = c/q$, $C_{r,i} = c_{r,i}/q$ for $1 \le i \le L-Y-1$, and $C_{r,i} = d_{r,i}/q$ for $i > L-Y-1$.

At this stage in the computation, everything becomes approximate as the true expressions are intractable to compute exactly. We make several simplifying assumptions throughout the analysis in order to obtain closed-formed estimates of these quantities.

We consider α^i as the probability that a key k appears in a run (or as data in a node) at level i, and α^{BC} as the probability k is in the block cache. Therefore, for uniform distributions, we obtain,

$$\alpha^i = \min(1, \frac{R^i}{|U|}), \alpha^{BC} = \min(1, \frac{E_{BC}}{|U|}) \tag{5}$$

For skew distributions, we obtain,

$$\alpha^{i,1} = 1 - \left(1 - \frac{p_{put}}{|U_1|}\right)^{R^i}, \alpha^{BC,1} = 1 - \left(1 - \frac{p_{get}}{|U_1|}\right)^{E_{BC}/B} \tag{6}$$

$$\alpha^{i,2} = \min(1, \frac{R^i(1-p_{put})}{|U_2|}), \alpha^{BC,2} = \min(1, \frac{E_{BC}(1-p_{get})}{|U_2|}) \tag{7}$$

where the superscripts 1 and 2 indicate k as a special key and a regular key, respectively. For the special keys, this is actually an approximation – we ignore that each run / data in a node consists of distinct elements. $1 - \frac{p_{put}}{|U_1|}$ denote the probability that a given

record does not contain key k, and we take this to the power R^i to compute an approximate probability that k appears in none of the R^i data entries. We then subtract this quantity from 1 to obtain the probability that key k appears. For normal keys, the calculation looks like a lot like the regular uniform calculations with the approximation that only a $(1 - p_{put})$ fraction of the R^i entries will be normal keys (i.e. the rest will be special keys). Note that all of the α^{BC} estimates are slightly inaccurate in that they do not take into account that keys in the block cache are repeated elsewhere in the data structure, but we believe the resulting error is minimal. Moreover, these estimates assume that the data structure is full (i.e. every run is filled to capacity). It is straightforward to add weights w if we instead want to hypothesize that runs are partially filled, and we often use a weight of $1/2$ in our predictions.

Now, we aggregate the α values into estimates of q, c, $c_{r,i}$, and $d_{r,i}$. The quantity $(1 - \alpha^{BC})$ gives us an approximate probability that k is not in the block cache, while the quantity $(1 - \alpha^i)$ gives us an approximate probability that k is not in run r / not data in a node at level i. We use the fact that $c_{r,i}$ and $d_{r,i}$ can be computed through considering the probability key k is not at the given run / node or anywhere earlier in the tree and that key k appears later in the tree. This exact expression is intractable so we make several simplifying assumptions. First, we condition on the event that the data structure is full (i.e. every level is filled to capacity), which enables us to use the α estimates to predict the probability that a key appears at a given level. Second, we assume that the merging strategy is such that all of a given level is merged at once, so that the data at runs is (conditionally) "independent", which enables us to treat the probability that key k appears at two different levels as independent. Now, to compute q, we can simply multiply together the probabilities that k is not in each run / node in a level and then subtract this quantity from 1. Similar logic yields c, $c_{r,i}$ and $d_{r,i}$.

2.2 Writes

For update cost, we emphasize the cost estimates are amortized, since any given merge is quite expensive, and merges happen infrequently. We also assume that merges are all-to-one in the sense that all runs in a given run are merged at once into a single run in the next level, and that all of the keys in level $L - Y - 1$ are merged through the cold levels in a batch. For cold levels, we consider how data is propagated from buffers at internal nodes to the final leaf node, but do not consider balancing costs or the costs of the tree shape changing drastically. This is representative of a "small" number of put queries that take place after bulk loading. It suffices to compute the total cost of a key propagating eventually to the bottom level of the data structure, in the following sense: if a put query of key k participates in a merge with keys arising from P other put queries that costs M I/Os, then we say that the cost incurred by a single put query is $\frac{P}{M}$. To compute the cost incurred by a put query, the relevant quantity is $\sum_{i=1}^{L} \mathscr{P}_i$, where $\mathscr{P}$ is the random variable cost incurred by a single put query in the merge from level $i-1$ to level i. For $i \leq L - Y - 1$, this depends on the number of put queries needed to fill level $i-1$. We let this quantity be $\mathscr{Q}_i$. Then it is clear that $\mathscr{P}_i = \frac{R^{i-1} \cdot K + R^i}{B \mathscr{Q}_i}$, since $\frac{R^{i-1} \cdot K + R^i}{B}$ is the read cost. We make the (crude) approximation that $\mathbb{E}[\mathscr{P}_i]$ is approximately equal to $\frac{R^{i-1} \cdot K + R^i}{B \mathbb{E}[\mathscr{Q}_i]}$. For the merges involving cold levels, first let's consider the case of $T = B$. we see that each key in level $L - Y - 1$ results in at most one block access at each future level. Another bound is the total number of blocks in that level. Thus, we obtain $\frac{\min(E_B \cdot T^{L-Y-1}, E_B \cdot T^i / B)}{\mathbb{E}[\mathscr{Q}_i]}$. For $T < B$, there is some gain for $i \geq L - Y + 1$ obtained by buffers batching writes at each internal node. In a given batch of $B - T$ entries, $\min(B - T, T)$ nodes at the next level are touched. So, each write costs at most $\frac{\min(B-T,T)}{B-T}$. This gives a $\frac{\min(B-T,T)}{B-T}$ multiplier on the above expression for levels $i \geq L - Y + 1$.

Thus, it suffices to lower bound $Q_i = \mathbb{E}[\mathscr{Q}_i]$. The building block of these parameters is the function family $f(V, M) = V \cdot \log\left(\frac{V}{V-M+1}\right)$, which is a coupon collector estimate of the number of queries to obtain M distinct elements in a uniform distribution with width V. The coupon collector problem studies the number of distinct values obtained from a sequence of i.i.d draws from a uniform distribution. The closed form estimate $f(V, M)$ is well-known, and is an approximation of the quantity $\frac{V}{V} + \frac{V}{V-1} + \ldots + \frac{V}{V-M+1}$ which captures the exact expected number of queries. This is closely related to Q_i, the number of queries needed to fill up a level, as follows. For a uniform distribution, we can thus compute Q_0 as approximately $f(U, R^0)$ (with some tweaking to account for error in estimation). We can compute Q_i similarly by approximately $K \cdot f(U, R^i)$ for $i \neq 0$. When the distribution is skew, the bound is a bit more complex. One bound arises from the fact we can upper bound by the case of uniform distributions with universe size $U_1 + U_2$ to obtain $g_1(0) = f(U_2 + U_1, R^0)$ and $g_1(i) = K \cdot f(U_2 + U_1, R^i)$ for $i \neq 0$. The other bound arises from the fact that if $R^i > U_1$, then $R^i - U_1$ are necessarily filled up by normal keys, which only show up in a fraction of put queries: we obtain $g_2(0) = f(U_2, R^0 - U_1)/(1 - p_{put})$ and $g_2(i) = K \cdot f(U_2, R^i - U_1)/(1 - p_{put})$. Combining these two estimates with a trivial estimate of M yields the desired expressions.

2.3 No-Result Lookups

Early stopping does not occur for no-resulting lookups, so every bloom filter must be touched. The number of I/Os now only depends on the randomness of bloom filters.

2.4 Range Queries

Every run in a hot level is necessarily touched. The number of blocks touched depends on the selectivity s, and it becomes critical that the runs are sorted. For the cold levels, if the data structure has $T = B$, then we can follow the path down to the leaf nodes and then use the fact that the leaf nodes are connected in a linked list to obtain the desired estimate. If $T < B$, then we need to check internal nodes as well as the leaf nodes for data. The fraction of internal nodes that are touched also depends on the selectivity.

3 ANALYSIS OF DISTRIBUTION-AWARE COST MODELS

We illustrate a comparative analysis of the proposed distribution-aware cost model with the worst-case model using an example. We present a case study of a database with 10 billion (10^{10}) entries experiencing a workload comprising of 50 billion operations. The workload is read-intensive with 80% reads and 20% writes. The workload possesses a skewed access pattern in which, a small set

of 10000 keys are 1000 times more likely to be chosen compared to the other keys for writes. This workload is inspired by real-life industrial examples of workload concerning twitter feeds of celebrities. A example scenario is of handling tweets from 10000 celebrities who are more likely, say by a 1000×, to tweet than a non-celebrity user on a particular day, and whose posts on that day are *significantly* (e.g. 8 million times) more likely to be read. For the sake of simplicity, in this example, we assume that the database is stored on a single 16 GB VM, bought by $8000 using AWS, although these observations generalize to more realistic multi-node systems as we will demonstrate later in the paper.

Precise Estimation of I/Os. In this scenario, running the optimization engine with the worst-case cost model results in a configuration $\Omega_{\text{worst-case}}$ as $T = 8$, $K = 7$, $Z = 1$, $L = 3$, $Y = 0$, $M = 16$ GB, $M_B = 3.2$ GB, $M_F = 12.8$ GB, $M_{FP} = 0.29$ GB, $M_{BF} = 12.5$ GB, where the FPR of the design is 0.006. The estimated number of I/Os incurred by $\Omega_{\text{worst-case}}$ according to the worst-case cost model is 40×10^9. However, when the same design is analyzed using the proposed distribution-aware cost model, the resulting I/O cost turns out to be 33×10^9, which is significantly lower. The main reason for the difference in cost is that, the proposed cost model perceives a design very differently from the worst-case model. Given a design, unlike the worst-case model, the proposed model probabilistically estimates and the number of levels and the number of the runs within each level of the design which every query within a workload may touch. The model exploits the characteristic of *early stopping* and takes into account that different queries can touch data residing at different levels of the tree and that result of any query may be returned much earlier than the last level. This leads to the first property of the proposed model as follows:

Property 1: *Due to early stopping, the estimated I/O cost of a particular design is significantly lower in the distribution-aware cost model as compared to the worst-case model.*

The implication of property 1 is that the performance of a design is significantly underestimated by the worst-case cost model. Therefore, if a user aims to achieve a certain performance for an application, worst-case models are likely to suggest designs that deliver much higher than the targeted performance, but as a result, the cost to be payed (for purchasing storage and compute resources) for supporting such designs also increases, thus significantly overcharging the application. On the other end, if a user has a budget fixed for an application, the worst-case model will select designs with performance much lower than what can be afforded with the cost.

Improving the Search Process Over the Design Space. It evidently follows from property 1 that, owing to the inaccuracies in the model design, searching for designs using worst-case models leads to design suggestions that are highly sub-optimal. Therefore, we incorporated the distribution-aware cost model within the search process to examine the difference in the suggested designs. We observe that the optimal design suggested by the proposed cost model, $\Omega_{\text{distribution-aware}}$, turns out to be $T = 32$, $K = 13$, $Z = 1$, $L = 2$, $Y = 0$, $M = 16$ GB, $M_B = 12.8$ GB, $M_F = 3.2$ GB, $M_{FP} = 0.29$ GB, $M_{BF} = 2.91$ GB, with the FPR being is 0.31. With this design, the I/O estimate according to the distribution-aware model is about 19×10^9 I/Os which shows that the search algorithm selects better designs compared to those of the worst-case models. We pinpoint to the fact that with the same financial budget of $8000 and the same cloud provider, distribution-aware cost models enables users to select better designs that offer higher performance. This leads to the second property of the model as follows:

Property 2: *The distribution-aware cost model can select better designs that offer lower performance, compared to worst-case models, with the same budget and the same cloud provider.*

The reason behind property 2 is that the distribution-aware lookup cost, accounting for early stopping, seeks to increase the size of the buffer in order to maximize the number of buffer hits of the *special* keys that are accessed more frequently. Because the special keys are likely to live in the buffer due to the workload access pattern, the cost model trades off a larger buffer size with a reduced FPR. On the contrary, the worst-case model does not account for the benefits of increasing the buffer size. Due to this, we observe that the worst-case I/O cost is much higher on the first design than the second design: the worst case I/O cost on the second design is 54×10^9 I/Os.

Generation of Fundamentally Different Designs. We perform a generalized analysis by extending the above analysis to include multiple cloud providers and a wide range of budget from $500/month to $50000/month. We include different workloads with varied read percentages in the above case study. For each budget, we generate the best performance using both the worst-case and the distribution-aware model and compute the average of the factor by which the the number of I/Os were reduced with the proposed model over the worst-case model. In Figure 1, the x-axis represents the different workloads and the y-axis shows the times of improvement in performance. Each line in the figure represents a different query distribution imposed on the above example. For example, uniform distribution means all keys from the workload are read and updated with equal probability. A 60% skew means that the point lookups targeted the special keys with a probability of 0.6% and the residual keys with a probability of 0.4%. Similarly, the definition holds true for 70% and 80% skews. We observe that, regardless of the distribution of the workload, the proposed cost model significantly improves the performance of read-intensive workloads as the model reasons about the cost of point-lookups differently from that of the worst-case model. We observe that for uniform distribution the distribution-aware cost model performs about 1.5× better but is particularly compelling for skewed workloads in which the performance increases as the skew percentage increases. This is mainly because the proposed model takes into account the possibilities of answering point lookups of frequently accessed keys that need disk I/Os less often. From this observation, we deduce the third property as follows:

Property 3: *The difference in performance created by the distribution-aware cost model with respect to the the worst-case model at the level of designs within a single VM, translates to generating fundamentally different configurations in the continuum that differ in the choices of hardware, combinations of VM instances, or the cloud provider for the same budget.*

With the increase in the growth of workloads and data, the error in the estimation of I/Os for worst-case models significantly increases. In fact, as the workload grows beyond a a certain point,

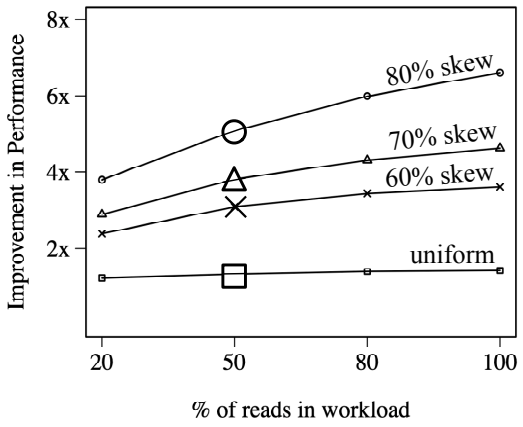

Figure 1: The figure shows the holistic implications on performance that can be achieved within the exhaustive continuum comprising of multiple designs, multiple cloud providers, and different budgets.

worst-case models turn out to be counterproductive within cloud computing environments. At such stages, I/O estimation becomes all the more crucial as it regulates either the cost payed for a targeted performance or the performance obtained for a fixed budget. To this end, the proposed distribution-aware cost model focuses to closely capture the cost-performance relationship that results in accurate estimation of the I/O cost.

Algorithm 1 Construction of Hardware Space

```
1:  Output: H = {H_{s_i}}, ∀s_i ∈ S
2:  procedure getAllVMCombinations(S)
3:      H_{s_i} = NULL
4:      for all s_i ∈ S do
5:          set Λ =< λ_{i,1}, λ_{i,2}, ··· , λ_{i,k_i} > to < 0, 0, ··· , 0 >
6:          for machines ≤ MAX_CLUSTER_SIZE do
7:              getCombination(s_i, 1, k_i, machines, Λ)
8:  procedure getCombinations(s_i, start, end, machines, Λ)
9:      if machines == 1 then
10:         for i ∈ [start .. end] do
11:             Λ[i]+ = 1; Add configuration Λ to H_{s_i};Λ[i]− = 1
12:     else
13:         Λ[start]+ = 1
14:         getCombination(s_i, start, end, machines−1, Λ)
15:         Λ[start]− = 1
16:         if start != end then
17:             getCombination(s_i, start+1, end, machines, Λ)
```

4 BENCHMARKING CPU UTILIZATION AND EVALUATING EFFECT ON PERFORMANCE

We created an extensive benchmark on RocksDB and WiredTiger to capture CPU utilization overhead and scaling of modern storage engines [33, 60] on varying hardware and workloads to understand the effects of the design features used in the I/O models, as well as what additional features we can use to navigate this tradeoff. In particular, the benchmark varies inputs (dataset size, workload) and I/O design features ($T, K, Z, M_{BF}, M_{FP}, M_B, concurrentthreads$) to isolate effects on the CPU utilization, as well as introducing additional features such as compression. We discovered that most I/O design features have a negligible effect on CPU utilization with the exception of two: a) multi-threading and b) compression.

Workload Entropy	Compression Type	Get Overhead (%)	Put Overhead (%)	Space Reduction (%)
0.1	Snappy	0.7	11.06	82.87
	ZLIB	10.25	21.75	91.22
0.25	Snappy	0.53	8.21	68.2
	ZLIB	25.45	31.26	83.18
0.5	Snappy	0.53	12.5	46.86
	ZLIB	26.77	12.1	70.28
0.75	Snappy	0.95	5.17	27.64
	ZLIB	28.65	22.29	59.04

Table 1: A compression behavior lookup table for CPU and storage, given hardware and entropy context.

Benchmark Setup. To capture CPU utilization overhead we compose variations on different workload conditions by running a benchmark consisting of 5 million writes and 50 thousand point queries run at different entropies and with varied compression schemes, including when no compression scheme is defined. This is shown in Table 1, where we display the *additive* CPU overhead and storage space reduction associated with four workload entropy contexts and two different compression algorithms for one hardware configuration. We use machines with Core i5 processors with 4 hyperthreads, 16GB DDR4 RAM, and dual drives with 256 GB of 2.5" SATA SSD and 256GB of m.2 SSD (Samsung 960 EVO) with Ubuntu 16.04LTS.

For a given compression scheme, its CPU overhead is the additional net change in CPU utilization compared to no compression scheme, where 100 percent utilization represents full utilization of all available CPUs. We measure CPU utilization over 1 second windows from the iostat tool, and average them over the course of each workload phase to reach a CPU utilization measurement. To then produce a compression scheme's CPU overhead, we subtract the respective measurements from an equivalent context with no compression scheme applied. We differentiate CPU overhead associated between that of I/Os of read query work and I/Os of write query work. Similarly, for the compression scheme, we determine the compression ratio, or space savings rate, by comparing the persistent data size on disk with that of the equivalent context with no compression scheme applied.

Compression for trading CPU against I/O footprint. Lossless compression algorithms [1, 34, 41, 49] can be deployed before storage block serialization to reduce the I/O cost of any given physical read or write to storage device. These compression algorithms make a trade, utilizing free available CPU to exploit redundancy and similarity between data entries in order to realize significant savings for I/O and storage needs.

In an otherwise I/O bound context, utilizing a compression algorithm adds pre-I/O steps of CPU-intensive compression and decompression steps during query processing. We model this nature of overhead as an *additive* penalty to latency due to the addition of pure CPU-bound steps.

On a CPU-overloaded machine, this lightens the overall load on the physical I/O device at the expense of potentially blocking query

processing with more intensive CPU load. This means, by choosing more heavyweight compression algorithms, we can further increase the realized storage and I/O savings as long as the bare metal physical machine can handle the additive CPU load. Cosine's CPU model, allows estimation of CPU effects due to compression, which allows an additional layer of precision beyond pure Disk Access Models which are only concerned with I/O access costs from block or page movements across the memory and storage hierarchy.

Design Features of I/O Model vs CPU Usage. Outside of the larger effects on performance (latency) for multi-threading and compression, our benchmarks indicate that most I/O design features have a negligible effect on CPU usage. Overall CPU time remains relatively stable for extended read-only workloads, where we run 1 million point read queries on 5 million keys in a RocksDB instance, for keys and values of 128 and 896 bytes, respectively. In Figure 2 we observe that varying T the growth factor (size ratio between levels), and memory for the bloom filters M_{BF} and size of the writes buffer M_B both create small differences in overall CPU usage, therefore suggesting that the design features of the I/O model may not need CPU motivated performance penalties for CPU bound latency reasons that are external of the I/O model. However, this can be a source of future modeling for further refining cost contributions if further data structure design features more aggressively use CPU compared to memory and storage.

5 ADDITIONAL DETAILS ON EXPERIMENTS

VM Parameters Used Within Experiments. We provide the experimental parameters used for cloud pricing policies in Table 2. For each provider, we present the configurations of the VMs used in terms of the memory capacity and the number of vCPUs within each VM.

Consolidation of CPU and I/O Models Towards Precise Latency Estimation. Figure 3 is the result of an experiment to compare the estimation of Cosine with that of the actual workload execution latency incurred within real-life storage engines. As shown in the figure, we use six different settings with different workload distributions, compression schemes, and query threads. For each setting, Cosine estimates the latency by consolidating the outputs of the distribution-aware I/O and CPU models. We compare the same with the actual workload execution latency when using RocksDB and WiredTiger. We observe that, Cosine's models approach the actual performance across diverse settings, whereas, the worst-case I/O model is significantly inaccurate.

With the help of the aforementioned experiment, we highlight the two-fold contributions of Cosine's cost models. Firstly, we identify that certain configurations require modeling of the CPU cost in addition to the I/O cost in order to be useful for predicting real latency. In such settings, although the I/O cost is dominant, the estimation of the CPU cost adds to the precision, which in turn, increases the overall accuracy of the holistic cost-performance optimization. Secondly, in addition to the distribution-aware I/O models (Section 3 of the main paper), we take a first step in precisely modeling the CPU cost emanating from the factors of compression and multiple query threads within VMs (Section 4 of the main paper). For settings with compression schemes, we observe that Cosine can isolate the latency contributions of CPU and I/O as shown using stacked representations for the fifth and sixth settings of Figures 3(A) and (B). In the absence of a compression scheme, the cost models of Cosine can capture a precise latency estimate of a workload but cannot isolate the contributions of CPU and I/O. In this case, we do not try to attribute latency causes to sources of I/O and CPU, and model the resulting latency as dominated by the influence of the I/O model. This choice stems from our observation that beyond features like compression and multi-threaded query processing, data structure design features have relatively smaller effects on CPU usage. However we recognize that further breakdown of how latency results, across all configurations, can be a useful point for future analysis and refined modeling.

Variation of Search Time With More VM Types. Figure 4 demonstrates how the search time of Cosine is affected as more VM types are added within the hardware space. As we vary the number of VM types in the x axis, we observe that the time taken to complete the search process increases as the cardinality of the hardware space increases. Nevertheless, for a significantly large number of VM types (about 35000), the search time takes about 6 hours. We highlight that this is significantly faster than the conventional benchmark where it takes upto month to determine a configuration, at the risk of it being sub-optimal.

6 ROBUSTNESS

The robustness of Cosine significantly depends on the confidence of the workload sample as changes in the workload composition affects the configuration optimizing the cost-performance trade-off. To this end, Cosine takes into account the uncertainties of workload through three distinct ways. Firstly, Cosine uses a slightly different variant of memory hopping for workload samples with low confidence. It terminates memory hopping without performing binary search on the hop regions thereby preventing the fine-grained allocation of the buffer (M_B) for a design. This results in assigning memory to the buffer at a course-granularity which in turn, leaves room to absorb uncertainties of workload upto a level. This allows Cosine to handle all workload fluctuations that demand M_B to be between $M_B \pm M'_B$ where M'_B would be the fine-grained allocation of memory to the buffer. However, for workload samples that have high confidence or are not likely to change over time, Cosine uses the full memory hopping routine to ensure precise allocation memory across buffers, bloom filters, and fence pointers.

Secondly, for each given configuration, after allocating memory to buffers, bloom filters, and fence pointers, there may be residual memory that can be assigned to either of the in-memory structures. Based on the workload composition, Cosine selectively decides to assign it to (a) buffers (for read-intensive or mixed workloads) or (b) fence pointers (for write-intensive workloads as they may grow with time leading to increase in the number of cold levels).

Thirdly, Cosine over-estimates some of the attributes of the workload sample during the construction of the workload feature vector. For example, while constructing the feature vector for uniform workloads, Cosine estimates U to be 100 times of the highest key in the workload sample, thereby allowing keys larger than that in the sample to be added to the workload over time.

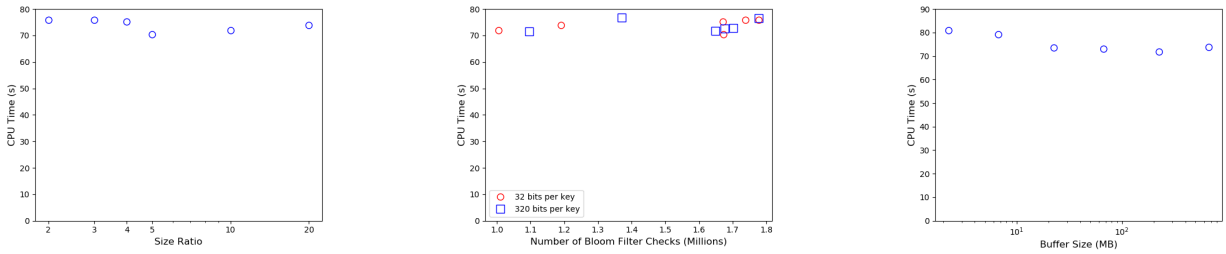

Figure 2: Absolute CPU usage is less responsive to I/O design features like Growth Factor, Bloom Filter Size and Buffer Size.

Table 2: Experimental parameters used for cloud pricing policies across different cloud providers

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

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

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

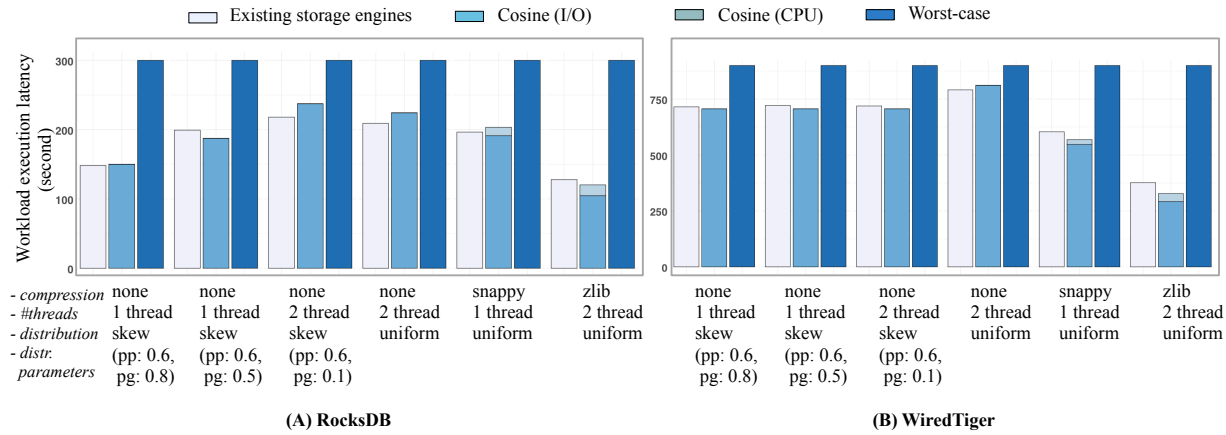

Figure 3: Accurately capturing the cost of complex designs through consolidation of I/O models and CPU models.

7 ADAPTABILITY

One of the crucial properties of a key-value storage engine is dynamic adaptation to changes in the application context, including changes in workload, hardware, cloud provider, or design. As mentioned in Section 6, Cosine is able to tailor the optimal configuration to handle different contexts up to some uncertainty. However, in order to be fully adaptable, Cosine should support dynamic switching of configurations as the context changes over time, however this is a very challenging problem. A change in one or more of the attributes within the application context requires the search algorithm be rerun, but once a change is identified, switching to the new optimal design has some cost emanating from the overhead of switching. Modeling this cost appropriately requires a robust analysis of the overhead due to switching between (a) designs, (b) machines, (c) pricing schemes, and (d) cloud providers. Furthermore, it is important to differentiate between transient and steady-state workload changes. For example, a 'burst' of 10% more queries for a few minutes would not necessitate an expensive migration, but

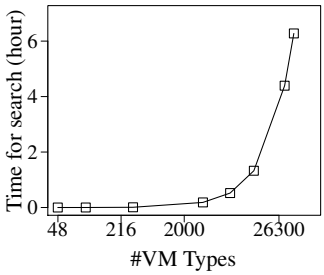

Figure 4: The time complexity of the search process within Cosine as more VM instances are added

if the change lasts for days the cost-benefit calculation changes and it might be necessary for optimal performance. Detecting these chracteristics of changes requires the deployment of a monitoring module. This would consume additional resources which must be included in the cost calculations, and the form of this monitoring may represent another design feature where more lightweight monitoring consumes fewer resources but reacts less effectively to changes. Solving these challenges represents exciting future possibilities, but currently the functionality of Cosine is restricted to determining optimal configurations before the initiation of the workload, and supporting dynamic adaptation to configurations is out of the scope of the paper.

8 EXTENSIBILITY

With growing data and diverse applications, it is crucial to be able to extend Cosine to support addition of auxiliary elements within each design knobs such as, other compression schemes, advanced workload features, cloud providers, and diverse data structure designs. Below, we discuss how Cosine can be extended to incorporate new elements which, in turn, enriches the configuration space and leads to selection of enhanced optimal-configurations with diverse possibilities of the design knobs.

Adding New Workload Features. The interleaving of queries within a workload affects the read-write I/Os as the data access patterns are further regulated by existence of data within in-memory data cache used within modern key-value storage engines. The internal cost models of Cosine are designed through reasoning of data access patterns and probabilistic existence of data entries within in-memory and disk-resident data structures. Given the workload distribution, the cost models can be further extended to include admittance and replacement policies within this cache to probabilistically determine existence of data blocks conditioned on the interleaving sequence of queries. This do not necessitate changing the existing I/O or CPU models but can be added as an extension on top of the core components.

Adding New Data Structures. The search space of Cosine can be extended to accommodate data structures, other than the B-trees, LSM-trees, and the hybrid variants, that are being used within state-of-the-art key-value storage engines. This not only includes data structure variants that persist data on-disk (e.g., LSH table [24, 57], LSM-Bush [30]), but also in-memory data structures (e.g., LSM-trie [62], Masstree [50]).

One of the first steps towards including a data structure design is to be able to decompose the design into fine-grained data layout primitives and integrate the primitives into the consolidated design space of existing designs. Every time, we add a new design to the design space, multiple design variants for different domain values of the primitives corresponding to the new design also get included to the space. Decomposing designs to its layout primitives is a manual process based on reasoning, observation, inference, and expertise.

As Cosine maintains unified cost models for all data structure designs, addition of new designs necessitate revisiting the cost-models to include diverse design parameters. The revised cost-models can be verified by comparing the estimations of the model with that of the actual performance incurred within the corresponding storage engines.

Incorporating New VM Instance Types and Cloud Providers. Currently, Cosine supports specific VM instance types – `r5` type for AWS, `n1-highmem` type for GCP, and `E2-64` type for Azure. Cosine maintains a library of the different cloud providers it supports and within the library of each provider, it maintains a list of VM specification and attributes related to the pricing scheme. For any cloud provider, Cosine can be effortlessly extended to include more VM types by adding the specification (concerning the number of vCPU cores, memory, and I/O limit) and hourly rates of the VM type to the library of definitions that Cosine uses. Similarly, diverse cloud providers can be added to Cosine's library to enrich the search space for more possibilities.

Adding New Compression Techniques. Currently, Cosine supports two compression schemes – Snappy and Zlib. For adding a particular compression scheme to Cosine, it is crucial to analyze the impact of the scheme on both space reduction as well as the CPU utilization. We empirically determine these effects in terms of constants that affects read and write-I/Os, as shown in Table 3 of the main paper. We apply the particular compression scheme on workloads with different entropy and build the lookup table for the behaviour of the compression scheme. This process can be repeated to extend Cosine to include more compression schemes.

9 ADDITIONAL RELATED WORK

We discuss related areas of multi-node cloud cluster management, cloud resource pricing, utilization and guarantees, and other data management work in cost optimization and their relation to Cosine's focus on optimality in systems and data structure design in the cloud.

Multi-Node Considerations: Sharding and Data Partitioning. Distributed data systems [4, 31, 42, 52, 54, 55] have grown in popularity due to the ease of horizontal scaling with cloud deployments to accommodate workload and data growth. In order to scale, these systems shard their data, distributing it among compute nodes and dynamically rebalancing as the total dataset size changes to maintain scaling properties without a priori guarantees on the data's growth rate. Current state-of-the-art systems use workload unaware methods, such as consistent hashing [31, 47, 54, 55], simpler hash-based sharding schemes [4, 52], or sharding by key ranges [42, 52]. These systems intend only to balance overall data size across all compute shards, usually with respect to a weighting determined by workload demands. Workload-aware partitioning has

been advocated in a number of systems and schemes for sharding and data placement, in order to improve distributed data layout [2, 5, 19, 27, 32, 44], but is not commonly used in production systems.

Our work can be used A) orthogonally to the sharding scheme, or B) as part of a co-optimization. Cosine can be used to optimize per-node data layout after either workload-unaware or workload-aware sharding, providing strictly additive benefits by finding the best data structure design for each node. In addition, our optimization problem formulation integrates with sharding: we design the distributed partitions of the data layout in a weighted manner that minimizes the criterion function for total cost (often measured as data movement, or over-the-network I/O). Cosine can relax its assumption of proportional sharding by straightforwardly incorporating this cost-model and co-optimizing the sharding scheme with the per-node data layout to minimize the total cost over any number of heterogeneous-sized partitions, allowing Cosine to be workload aware. This approach can be extended to consider heterogenous per-node hardware choices across the cluster at the expense of more search time, tapping into a larger space of cluster designs which captures state-of-the-art sharding.

Cloud Resource Pricing, Utilization and Guarantees. Cloud tenants and providers are increasingly sensitive to the price charged for resources (VM compute, network bandwidth, etc) [20, 45, 59]. Cloud providers focus on increasing utilization and revenue by balancing the tension between ensuring the availability of extra resources to guarantee performance thresholds, and selling these resources (e.g. bandwidth, or CPU credits available) for additional revenue [51].

While the monetary dimension is a concern in this subarea, we are unaware of attempts to forecast and derive the tradeoff between end economic cost and performance of individual application instances in relation to their starting configurations. Cosine is orthogonal to vendors' designs of pricing schemes and can integrate with additional pricing schemes as they arise. While pricing and resource markets can change for monetary costs dynamically, Cosine allows the usage of arbitrary pricing schemes on each cloud provider. Cosine navigates between multiple pricing schemes (or markets) across different providers to satisfy arbitrary balances of cost and performance and provides the cloud tenant the combination of hardware choice and data layout that gives the best performing configuration for their given workoad and preferences.

Workload Tailored Data Layouts. In the relational context, a recent system Chestnut proposes automatic data layout design for object oriented database applications [63], using traditional constrained optimization techniques (integer linear programming). Chestnut first determines the space of data layouts to search, via analysis of the application code to determine the application's workload needs. To enumerate the possible search space for a single best layout, Chestnut considers a handful of data layout design factors such as row vs column vs grouped formats, and auxiliary index design such as B-tree and Hash index. However, Chestnut fundamentally deals with OODA in-memory workloads layered on top of a persistent relational database and is focused on finding an optimal performing application-tailored in-memory data layout.

In contrast, Cosine enumerates over a possible key-value store configuration space in a way with four distinguishing properties. First, unlike Chestnut, Cosine searches a constrained key-value store *configuration space*, that considers the cloud hardware context for I/O, storage, compute and memory costs, not just pure performance in terms of latency or throughput. Second, Cosine is able to describe designs that cover not only LSM-trees, B-trees, B+-trees, and B^ϵ-trees, but also all intermediate variants that are unexplored till date. Third, Cosine considers the performance-cost implications of the persistent data store, not just the performance of the layer which operates in-memory. We however, recognize that for future work, Cosine's performance modeling of mostly in-memory workloads can be further refined, as our current models focus on quantifying I/O's to persistent block storage as that is the largest latency bottleneck. It could be possible, as suggested in [46], to extend other data models above the key-value model. For instance, further refining I/O cost modeling to account for further operation support for quries across or within values that can be serialized or interpreted as objects or documents with further queryable sub data. This would go beyond the scope of our current Cosine search space and modeling. Fourth, Cosine considers the possibility of utilizing sufficient CPU resources to assign more than one query thread to speed up the performance of read-heavy workloads, which Chestnut does not address.

Data Management as Interpretable Optimization. Beyond the NoSQL context, in data warehousing, the topics of optimal index selection, query planning, and algorithms design are domains with established classic meta problems of search, ranking, selection, and synthesis of efficient configurations, designs and plans (e.g. physical storage, indexes, query planing) [25, 56]. Recently, work using black box machine learning (ML) driven approaches such as neural networks have grown in popularity. The learned approach tries to capture the mappings of each operational context to its best configuration for optimal performance. However, this requires accurate modeling of the holistic system behavior and understanding the interaction of multiple factors to achieve robust performance inference. Any predicted end performance needs to be learned from features and patterns that the ML model extracts from the observed performance traces of the data system [3, 28]. As such, generalizing performance prediction across differing hardwares, workloads, and other contexts is challenging.

Cosine's recommendations are more robust than pure ML approaches. It uses explicit I/O and CPU models which break down end performance prediction into clearly delineated parts, which black box approaches like neural networks have no guarantee of when predicting and choosing configurations. This also allows us to gain insights into the configurations and improve generalizability to new hybrid designs. Cosine is responsive - it does not need a runway of training time for trying configurations or tuning of ML model parameters, which allows rapid deployment and cost reduction by reducing the need for full runs on test environments that learned models need to accumulate traces for training. Additionally, in what-if mode, Cosine has the potential to reason about "neighboring" configurations and suggest an alternate configuration alongside the optimal, where a relatively small increase in cost

beyond the user's maximum cost threshold could still result in a favorable performance gain.

10 WORKLOAD CHARACTERIZATION

As mentioned in Section 3 of the main paper, the workload characterization considers composition of the workload and the distributions (uniform and skew) of both the keys and queries. In this section, we justify the different design choices in the workload feature vector and the modeling of the distribution.

Distribution-Based Methodology. The distribution-based methodology used within Cosine is general, and more distributions could be added in the future versions. We define four operation types, as displayed in Table 1 of the paper (writes, point read lookups - single result and no result, and scans). We begin first with the uniform and skew distribution as they are the building blocks of approximating more complex distributions. Fundamentally reads and writes can take on different declared distributions. It is of note, that our skew distribution formulation further builds off the uniform, as a composition of two uniform distributions. Employing further known statistical transformations on top of uniform distributions will yield enriched distribution representations for characterization extension. For instance, the Box Muller transformation shows how to yield normal random variables (RVs) from uniform RVs [22]. Our approach can express zipfian distribution approximately as a skew distribution as other recent studies of data structures and algorithm have done [23].

Mapping to YCSB. Our workload characterization maps to YCSB's benchmark generator inputs [26], and in certain cases allows more workload expression.

(1) We provide four query type proportions as input as well. Note in our key-value model we treat updates and inserts similarly as PUTs, while we distinguish between single result and zero result point lookups.
(2) YCSB's query distribution default is uniform, as is ours.
(3) Of YCSB's six pre-defined workloads (A-F), five are i.i.d with respect to the distribution of their queries, compared to the base data [26].
(4) We are able to capture the aspect that within queries, reads and writes can be from two i.i.d distributions. This is not explicitly covered within YCSB's benchmark.
(5) Our characterization expresses skews which can be used to approximate YCSB's zipfian. Further composition of more complex random variables using the uniform and skew can get closer to describing more types of skew.
(6) YCSB Workload D has a *latest* temporal distribution of which 95% lookups and 5% inserts, and are drawn from a swath of latest keys. While we cannot explicitly describe this workload at the small scale of a few queries, in aggregate our skew distributions can be configured, such that incoming read queries pull from a skew distribution S1, while incoming writes pull from a separate skew distribution S2, covering the same support space of keys, where S1 and S2's respective head of the distributions are located at different ranges of the key space. Approximating a distribution where a key subrange is much hotter than the rest of the keys. At scale for aggregate performance, assigning a different skew to base data vs read vs write queries can approximate workloads with temporal skew.

Workload Simplicity. As the cardinality of the design space dominates the size of the search space, it is crucial that we strike a balance between simplicity (which is related to runtime of computing optimal solution) and accuracy while evaluating designs during the search process. Therefore, we make the simplifying assumption that the query keys are drawn i.i.d from our data distributions. We further verify the correctness and the precision of the I/O models when applied of the input workload feature vector in Figure 4(A) of the main paper.

Interleaving of Queries. In a real workload, the ordering of queries may exhibit different access patterns including correlated data access. If Cosine is designed to explicitly take into account correlations and orderings, we would be able to retrieve the exact cost for every sequence of queries, but at the cost of an extremely high dimensional workload feature vector. This, in turn, would lead to intractability, thereby impeding the reasoning and analysis of the I/O models. As the search space of Cosine is massive, to facilitate effective and efficient evaluation of the designs, it is crucial to prioritize precise closed-form estimate of the amortized I/O cost of the workload for each design, rather than any individual query. Cosine is still able to capture and approximate access patterns through estimation of the probabilistic existence of keys both in the memory, cache, and different disk-resident levels of the data structure design after each batch of queries. The I/O cost models of Cosine particularly work well for large workloads where the effects due to correlated access and interleaving patterns are amortized over a large number of queries.

11 COST FOR RESOURCES DOMINATES OTHER COST FACTORS

As Cosine is a templated storage-engine deployed with the best cost-performance tradeoff within cloud environments, one natural question that arises is – how does Cosine take into account the numerous other cost factors concerning operational stability or ecosystem support that affect the choice of the cloud provider for a given application. In fact even for a single provider, in addition to conventional performance metrics such as throughput or latency, there exists a multitude of Service Level Agreement (SLA) factors contributing to performance: 1) availability of VMs, 2) fault-tolerance of the cluster, 3) security guarantees, 4) tools and API support, 5) durability, and 6) recovery. The current version of Cosine does not take these cost or SLA factors into consideration. In this section, we present a discussion on how the contributions of Cosine is relevant and useful regardless of the inclusion of these factors as the cost for resources dominates over that emanating due to other factors. In the next section, we illustrate how Cosine can be extended to include all these factors into its architecture.

Cost Emanating From Resources Dominates. We observe that in a cost breakdown for large-scale applications, the cost due to storage and compute resources dominate over the other factors as mentioned above. We demonstrate this with a couple of examples below:

Example 1: Let us consider a 1 TB database migrated once a month. We use the corresponding pricing models of database migration for AWS [7], GCP [39], and Azure [12] and assume the underlying storage type to be General Purpose (SSD) Storage. The rate for a monthly transfer offered by AWS, GCP, and Azure are \$0.115/GB, \$0.17/GB, and \$0.37/GB, respectively. At the end of the month, it incurs about \$115, \$170, and \$370 for AWS, GCP, and Azure, respectively. However, as we observe from our experiments (Figure 5 in the main paper), the competitive performance of a workload (that also has about 1 TB of base data) starts at a budget of about \$15000/month. Therefore, we believe that the data migration services in this example contributes by about 2% of the overall budget.

Example 2: For the same database used in Example 1, let us consider reliability and disaster recovery plans offered by the cloud providers. We consider the corresponding pricing models for AWS [9], GCP [36], and Azure [13]. The corresponding rates for a monthly subscription in this case would cost \$99 for AWS, \$92 for GCP, and \$41 for Azure. These plans come as one time subscriptions independent of workload. We observe that, the subscription cost for reliability and disaster recovery with this example database is a maximum of 0.6% the original cost (\$15000).

Example 3: For this case, we demonstrate the scenario with the feature of availability of VMs for each cloud provider, which is a crucial SLA requirement for end-users. The availability, as also explained in Section 12, is a feature that is offered free of cost (i.e., it does not have a pricing model for charging end-users) with any service that an end-user purchases. It is measured in terms of monthly uptime percentage denoting the expected proportion of time the servers are expected to be up and running. However, if a provider fails to achieve the promised uptime (99.9% for AWS [10], 99.5% for GCP [37], or 99.5% for Azure [17]), it credits the end-user with 10% of the total price payed for a given billing cycle. The cost considerations for this feature effectively adds to the budget decided by the end-user for each month (or billing cycle) with a significantly low probability of about 0.001, thereby having neglibly impact on the overall cost.

All SLA Metrics Are Computable. We highlight that no matter how complex the underlying processing of a cloud service is, we are able to compute these costs from each of these factors because they are offered as services with simple cost models. From these examples, we observe that even if we take into account all these cost factors, the cumulative contribution to cost would not overpower the impact of purchasing resources. This can be further backed up by studies that show resources consume more than 60% of the total cost [43] [18] in cloud.

Data Structure Designs Control Optimization. The dominance of resources in the overall cost is driven by the co-design of data structure and hardware at the core of Cosine. Each design dictates the amount of resource footprint (CPU and memory) it uses, which in turn, affects the costs of configurations. Therefore, we claim that, the core of the cost-performance optimization used within Cosine is fundamentally driven by the data structure design space. Although inclusion of additional cost factors add to the robustness of Cosine, they do not significantly affect the outcome of the optimization.

Top Cloud Providers Do Not Exhibit Noticeable Difference in Non-Resource Cost Factors. In the small space of budget where the SLA and other cost factors contribute, there can be noticeable differences due to pricing models of different cloud providers. Currently, as we restrict ourselves to the comparison among the established cloud providers, we do not expect significant differences in terms of the SLA. This is because, as these providers are in close competition, the prices and the service values they offer are very close. For example, if we observe the factors used in the aforementioned examples, we would notice only minor differences in the pricing models across the three providers. However, as more and more cloud providers are added to Cosine, inclusion of these cost factors would be crucial and would impact correctness of the overall optimization.

12 EXTENSION OF COSINE WITH SLA

At the cloud provider end, each SLA feature and other cost-factors have their own internal mathematical model to support the pricing. However, the end-users perceive these factors through simple, computable pricing models as exposed by the providers. Therefore, all of these features can be readily computed and integrated within Cosine. In this section, we show some examples of these features. For each feature, we include an overview of the pricing model that the cloud providers follow and a brief implementation plan of how we plan to integrate the corresponding model within Cosine. We present a detailed description of the first feature and summarized descriptions for the subsequent features.

(1) **Database Migration**: Database migration is offered as a cloud service that end-users can purchase on top of the core storage and computing resources. For each geographic region, a cloud provider provides a pricing model for migrating databases to one or more data-centers of that region. Using this pricing model, one can compute the cost of migration. For example, for each instance type, the pricing model of AWS [7] advertises a cost in \$/hr for migration. Migrating the content of a `t2.micro` VM costs \$0.018/hour, while for migrating from SSD, the cost is \$0.115/GB. Therefore, if one knows the data volumes within persistent and non-persistent data storage, it is possible to compute the costs due to migration. Similarly, one can obtain the costs incurred with GCP [39] and Azure [12].
Implementation Plan: For every configuration, Cosine knows the storage and the VM instance type. Using the storage type, Cosine can look up the price of migrating for each provider for that storage type. With the size of the base data, Cosine can compute the total cost of migrating the entire persistent data. A similar process can be used to determine the cost to transfer data across VMs. The cumulative cost will be added to increase the cost of a configuration. At the time of navigating within the cost-performance continuum, if the cost of a configuration exceeds the input budget, Cosine will anyway ignore it. Further, Cosine has the knowledge of the network bandwidth for each provider in the search space. Based on the input frequency of migration (as part of the SLA), Cosine can determine the migration time for a database to different geographic regions.

(2) **Availability or Monthly Uptime Percentage**: This refers to the promised percentage of time during which the the VMs of an application are promised to be up and running. AWS and Azure offer 99.99% as the monthly uptime percentage [10] whereas Google offers 99.5% [37]. In case of AWS, any deviation from the promised value, results in 10% (< 99.99%, > 90%) or 30% (< 90%) as service credit percentage which means the corresponding percentage of the total charges paid by the end-user is returned at the end of each billing cycle. The models used by GCP and Azure are similar with the conditional penalties being different.
Implementation Plan: Firstly, for a given requirement of uptime by the end-user, Cosine can exclude a service provider from the search space that fails to meet the requirements. Once the optimal configuration is determined, Cosine will share the service credit percentage of the cloud provider with the end-user for future reference and verification purposes.

(3) **Security**: Cloud data security services are offered in AWS through identity and access management (IAM) framework that protects user confidential data and offers role-based access control. Currently AWS provides this service free of cost as of 2019 [8]. GCP offers Security Command Center [38], a security management and data risk platform that prevents threats and meets compliance requirements for customers. The price charged for Security Command Center depends on how much data is transferred securedly using the secured API ($0.3/GB). Similarly, Azure provides Security Center that protects user data and resources at prices dependent of the number of servers used over time ($0.021/server/hour) [16].
Implementation Plan: We observe that this metric is measured differently across different providers. For inclusion of this metric, as we enter the search space of a cloud provider say GCP, we plan to determine the maximum cost (before constructing the hardware space) to be able to transfer data through the secured API as a threshold percentage of the base data. We deduct the corresponding cost from the overall input budget. Similarly, for Azure, during logging the performance (latency) of a particular configuration, we include the security cost (in $/server/hour) emanating from the cluster size of the configuration.

(4) **Reliability and Disaster Recovery**: This refers to ensuring uninterrupted flow of workload operations during natural disasters, mechanical failure or human error. AWS provides support for reliability using its CloudEndure Disaster Recovery service that charge customers yearly or monthly based on their cluster size [9]. Similar pricing models exist for GCP [36] and Azure [13].
Implementation Plan: This metric within Cosine can be added by deducting the price of the disaster recovery scheme (monthly/yearly/3 yearly) from the budget. The pricing model of this metric is designed as a one time investment for the cloud providers.

(5) **Operational and Tool Support**: A DevOps model supports the facilitation and integration of automated processes, advanced tooling and use of technology stack, faster deployment, and application evolution. DevOps is offered as SaaS with its own pricing models by the top cloud providers [11, 15, 35].
Implementation Plan: Similar to the point above, support to DevOps can be integrated into Cosine as a fixed deducted from the input budget. There are multiple DevOps schemes that the end-user can subscribe to and the preferred scheme can be input to the SLA requirements.

(6) **Durability**: For different storage types, the cloud providers [6, 14, 40] possess durability guarantees measured as the number of 9's after the decimal of 99. For example in Azure, for locally redundant storage, zone redundant storage, geographically redundant storage, and read-access geographically redundant storage, the durability is 99.999999999% (11 9's), 99.9999999999% (12 9's), 99.99999999999999% (16 9's), and 99.99999999999999% (16 9's), respectively.
Implementation Plan: The support for durability can be added during the construction of the hardware space. For all provider-hardware-region combinations that do not meet the input durability requirements (specified in the SLA) we exclude them from the search space of Cosine.

13 EXTENDING COSINE WITH ECONOMIC MODELING

The previous section considers costs that fall within the cloud model. However, there can also be additional costs that fall outside of it and are specific to a particular business or industry. By adding economic model based costing functions as additional inputs, Cosine can account for these factors and add another level of cost modeling. This is done on top of storage engine pricing, and would allow the development of a wider holistic costing strategy.

(1) **Avaliability, Reliability, and Durability**: Downtime and reliability have a real economic impact on businesses. For example, one hour of downtime for Amazon on Prime Day in 2018 cost it up to an estimated $99 million dollars in sales [61]. Similarly, a lost transaction due to a durability issue has a quantified cost. Cyber risk management firms have created detailed models for the impact of these effect, mostly targeted at the insurance industry, which can output cost ranges for these catastrophic events [48, 53]. For cloud avaliability and data loss, we can directly determine expected costs based on the SLAs or complied reports of previous issues. For storage engine reliability, we would first need to create a benchmark to determine realistic bounds on these effects, and can then apply the models.

(2) **Operational and Tool Support**: The modeling of software development costs is a well developed field of study with a nearly 40 year history [21, 58]. The newest iterations of these models, such as COCOMO III, attempt to take into account the impact of DevOps tools and other parts of the modern software development ecosystem [29]. Coupling Cosine with one of these models to include the impact of different amounts of ecosystem support would allow for a fuller accounting of the costs associated with a cloud provider choice.

Each of these cases requires additional inputs into the model based on the individual business factors of the client, and adds additional costs to the overall model. However, these considerations are on top of the core storage engine costs, which are still a critical

portion of the overall cost and can be described by Cosine in its current form.

Challenges That Still Persist. After these factors are included within Cosine, there would be two challenges that would still persist – (a) It is extremely difficult to compare some of these factors across multiple cloud providers. For example, it is hard to reason about the operational support or reliability guarantees across different providers as it is qualitative in nature and there exists no metrics to quantify or measure these factors in a unified manner across different providers. (b) It is also non-trivial to identify weights that assign relative priorities to each of these factors. For example, from the user preference, it will be non-trivial to prioritize security over fault-tolerance by assigning precise weights that capture user preference.

14 CPU ASSUMPTIONS

Here we clarify two sets of assumptions about modeling CPU resource usage. First we address how our methodology handles external effects due to factors like hardware and system specific effects. Second we further explain our treatment of the threading space.

Isolating Hardware and System Specific Effects. As reflected in Section 4, when we run the CPU benchmarks for the compression lookup table, we seek to measure the *additive* overhead of using different compression schemes by incurring additional CPU cost in exchange for savings in I/O and storage. Our methodology controls for external hardware and system specific effects (such as CPU overheads inherent to hyperthread implementation, OS, Kernel) in the baseline benchmark measurements by running the benchmark on the system of interest. For instance the compression behavior lookup table captures the *additive* performance overhead of compression alternatives compared to the overhead of no compression scheme.

Thread Ceiling and CPU Linearity. A thread consumes cycles issuing/waiting for I/O, user application computational tasks (e.g. compression, hashing), or system tasks (thread switching, etc). Therefore we try to set a conservative ceiling for when a hyperthread (virtual core) would have ample capacity for taking further work. We call the capacity within this the *CPU sufficient regime*. We focus on the CPU sufficient regime, because data structure designs that leverage idle CPU resources (as dictated by cloud hardware specs) can strike better performance vs cost tradeoffs for a given hardware setup.

For instance, on our experimental hardware, the average reading of the RocksDB LSM-tree expends 5-6% of available CPU. In this particular case, it suggests that scaling up to the number of hyperthreads is not CPU bound, and that the linearity assumption is reasonable. However this does not necessarily hold true when CPU intensive compression is utilized, as more hyperthread capacity is expended on compression and decompression during the writing and reading of data. We see for instance, in the experiments for RocksDB under no compression, that latency scaling is approximately linear up to the number of available hyperthreads. We are happy to conduct further experiments to support when and how the linearity assumption breaks down beyond available hyperthreads, for the CPU insufficient regime.

Currently we do not attempt to model beyond this thread ceiling. When the number of threads exceeds the hyperthread limit, the CPU - latency scaling is no longer guaranteed to be linear, as the ceiling of available hyperthread capacity can be violated by thread contention due to simply A) sheer number of threads or B) user computational tasks like compression.

REFERENCES

[1] 2019. LZ4 - Extremely Fast Compression algorithm.
[2] 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 (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 2–2.
[3] 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.
[4] Apache. [n. d.]. Cassandra. *http://cassandra.apache.org* ([n. d.]).
[5] Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. 2019. Cache-aware Load Balancing of Data Center Applications. *Proc. VLDB Endow.* 12, 6 (Feb. 2019), 709–723.
[6] AWS. 2019. Amazon S3 Storage Classes. https://aws.amazon.com/s3/storage-classes/.
[7] AWS. 2019. AWS Database Migration Service pricing. https://aws.amazon.com/dms/pricing/.
[8] AWS. 2019. AWS Identity and Access Management (IAM). https://aws.amazon.com/iam/.
[9] AWS. 2019. CloudEndure Disaster Recovery. https://aws.amazon.com/marketplace/pp/Amazon-Web-Services-CloudEndure-Disaster-Recovery/B073V2KBXM.
[10] AWS. 2019. Prior Version(s) of Amazon EC2 Service Level Agreement - Not Currently In Effect. https://aws.amazon.com/ec2/sla/historical/.
[11] AWS. 2019. What is DevOps? https://aws.amazon.com/devops/what-is-devops/.
[12] Azure. 2019. Azure Database Migration Service pricing. https://azure.microsoft.com/en-us/pricing/details/database-migration/.
[13] Azure. 2019. Azure Site Recovery pricing. https://azure.microsoft.com/en-us/pricing/details/site-recovery/.
[14] Azure. 2019. Azure Storage Overview pricing. https://azure.microsoft.com/en-us/pricing/details/storage/.
[15] Azure. 2019. Pricing for Azure DevOps. https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/.
[16] Azure. 2019. Security Center pricing. https://azure.microsoft.com/en-us/pricing/details/security-center/.
[17] Azure. 2019. SLA for Virtual Machines. https://cloud.google.com/functions/sla.
[18] Backblaze. 2017. The Cost of Cloud Storage. https://www.backblaze.com/blog/cost-of-cloud-storage/.
[19] Nicholas Ball and Peter Pietzuch. [n. d.]. Skyler: Dynamic, Workload-Aware Data Sharding across Multiple Data Centres. ([n. d.]).
[20] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. The Price Is Right: Towards Location-Independent Costs in Datacenters. ACM HotNets.
[21] Barry W. Boehm. 1981. *Software Engineering Economics*. Prentice Hall.
[22] G. E. P. Box and Mervin E. Muller. 1958. A Note on the Generation of Random Normal Deviates. *Ann. Math. Statist.* 29, 2 (06 1958), 610–611.
[23] Jehoshua Bruck, Jia Gao, and Anxiao Jiang. 2006. Weighted Bloom Filter. In *Proceedings of the International Symposium on Information Theory (ISIT)*. 2304–2308.
[24] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 275–290.
[25] 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.
[26] 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.
[27] Carlo Curino, Yang Zhang, Evan P C Jones, and Samuel Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment* 3, 1 (2010), 48–57.
[28] 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 (SIGMOD '19). ACM, New York, NY, USA, 666–679.
[29] 2018 COCOMO Users Group David Seaver. 2018. Agile to DEVOPS: Estimate Software and See the World. https://csse.usc.edu/new/wp-content/uploads/2018/10/cocomo-version-final-seaver.pdf.
[30] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
[31] 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.
[32] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 79–94.
[33] Facebook. [n. d.]. RocksDB. *https://github.com/facebook/rocksdb* ([n. d.]).
[34] Facebook. 2019. Zstandard - Fast real-time compression algorithm.
[35] GCP. 2019. DevOps. https://cloud.google.com/devops/.
[36] GCP. 2019. Disaster Recovery Planning Guide. https://cloud.google.com/solutions/dr-scenarios-planning-guide.
[37] GCP. 2019. Google Cloud Functions Service Level Agreement (SLA). https://cloud.google.com/functions/sla.
[38] GCP. 2019. Pricing. https://cloud.google.com/security-command-center/pricing.
[39] GCP. 2019. Pricing for Migrated Workloads. https://cloud.google.com/migrate/compute-engine/pricing.
[40] GCP. 2019. Storage classes. https://cloud.google.com/storage/docs/storage-classes.
[41] Google. 2019. Snappy - A Fast Compressor/Decompressor.
[42] Bram Gruneir. 2017. Scalable SQL Made Easy: How CockroachDB Automates Operations .
[43] J. Hamilton. 2010. Overall Data Center Costs. https://perspectives.mvdirona.com/2010/09/overall-data-center-costs/.
[44] Monika Henzinger, Stefan Neumann, and Stefan Schmid. 2019. Efficient Distributed Workload (Re-)Embedding. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 13 (March 2019), 38 pages.
[45] 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 (WINE 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 73–86.
[46] 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)*.
[47] 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 (STOC '97)*. ACM, New York, NY, USA, 654–663.
[48] Lloyds and Air Worldwide. 2018. Cloud Down: Impacts on the US economy. https://www.lloyds.com/news-and-risk-insight/risk-reports/library/technology/cloud-down.
[49] Jean loup Gailly and Mark Adler. 2017. Zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library.
[50] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 183–196.
[51] 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.
[52] MongoDB. [n. d.]. Online reference. *http://www.mongodb.com/* ([n. d.]).
[53] George Ng. 2019. More Powerful Than Ever: The Fourth-Generation of Cyence for Cyber Risk Management. https://www.guidewire.com/blog/technology/more-powerful-ever-fourth-generation-model-cyence-cyber-risk-management.
[54] Oracle. 2018. Introducing Oracle Database 18c. *White Paper* (2018).
[55] OrientDB. 2019. OrientDB Documentation: Auto Sharding Index Algorithm.
[56] 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.
[57] Justin Sheehy and David Smith. 2010. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper* (2010).
[58] Andrew Stellman and Jennifer Greene. 2006. *Applied Software Project Management*. O'Reilly Media, Inc.
[59] 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.
[60] WiredTiger. [n. d.]. Source Code. *https://github.com/wiredtiger/wiredtiger* ([n. d.]).
[61] Sean Wolfe. 2018. Amazon's one hour of downtime on Prime Day cost it up to 100 million dollars in lost sales. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7.
[62] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 71–82.
[63] C. Yan and A. Cheung. 2019. Generating Application-Specific Data Layouts for In-memory Databases. In *Proceedings of the VLDB Endowment*.