Technical Report for Cosine

ACM Reference Format:
. 2019. Technical Report for Cosine. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 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,

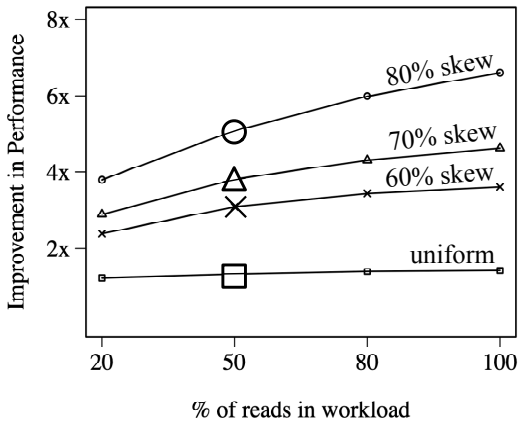


Figure 1: The figure shows the holistic implications on performance that can be achieved within the exhaustive continuum comprising of multiple designs, multiple cloud providers, and different budgets.

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 [15, 31] 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

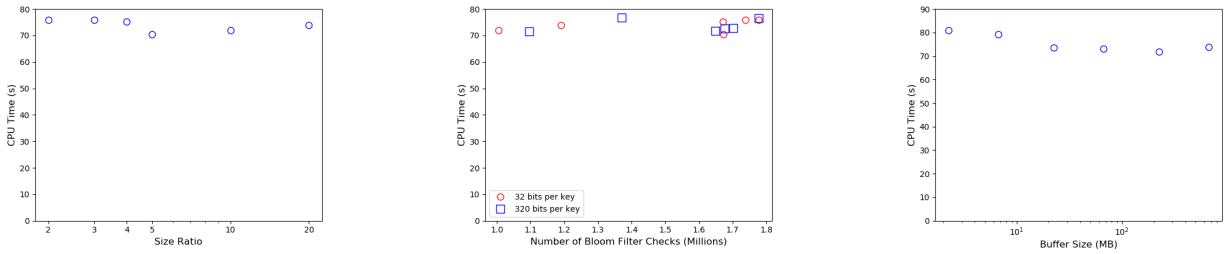
Table 1: A compression behavior lookup table for CPU and storage, given hardware and entropy context.

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. This is shown in Table 1, where we display the *additive* CPU overhead and storage space reduction associated with three workload entropy contexts and three different compression algorithms for one hardware configuration. We use machines with Core i5 processors, 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.

Compression for trading CPU against I/O footprint. Lossless compression algorithms [1, 16, 17, 22] 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 in order to realize significant I/O savings.

In an otherwise I/O bound context, utilizing a compression algorithm adds pre-I/O steps of CPU-intensive compression and decompression steps on the data. The nature of this overhead creates an *additive* penalty to latency due to 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 CPU estimation which is more precise than 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 thatvarying T the growth factor (size ratio between levels), and memory for the bloom filters M_{BF} and size of the



(a) (b) (c)

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

writes buffer M_B both create small differences in overall CPU usage, therefore suggesting that the design features of the I/O model do not need CPU motivated performance penalties for CPU bound latency reasons that are external of the I/O model.

5 ADDITIONAL DETAILS ON EXPERIMENTAL SETUP

We provide the experimental parameters used for cloud pricing policies in Table 2. These providers, pricings and specifications create a set of hardware choices when determining the best cloud configuration.

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 seven different settings with different workload distributions, compression schemes, and multiple threads. For each setting, Cosine estimates the latency by consolidating the results of the I/O and CPU models. We compare the same with the actual workload execution latency when used RocksDB and WiredTiger. We observe that, Cosine's models correctly predict the performance at diverse settings, whereas, the worst-case I/O model is significantly inaccurate.

6 ROBUSTNESS OF COSINE

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.

7 ADAPTABILITY OF COSINE

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 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

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

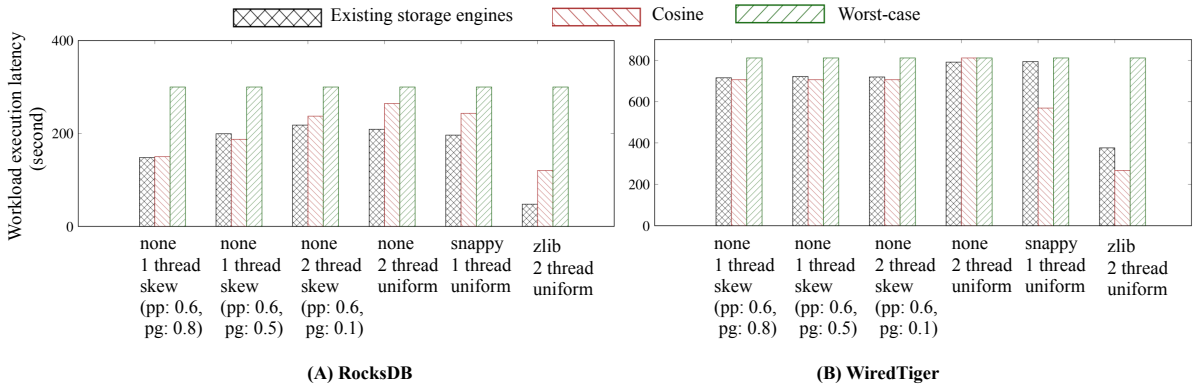


Figure 3: Accurately capturing the cost of complex designs through consolidation of I/O models and CPU models.

supporting dynamic adaptation to configurations is out of the scope of the paper.

8 EXTENSIBILITY OF COSINE

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 [8, 29], LSM-Bush [12]), but also in-memory data structures (e.g., LSM-trie [32], Masstree [23]).

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 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, 13, 18, 25–27] 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 [13, 21, 26, 27], simpler hash-based sharding schemes [4, 25], or sharding by key ranges [18, 25]. 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, 6, 10, 14, 19], 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) [7, 20, 30]. 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 [24].

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.

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) [9, 28]. 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, 11]. 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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*.

[13] 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.

[14] 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.

[15] Facebook. [n. d.]. RocksDB. *https://github.com/facebook/rocksdb* ([n. d.]).

[16] Facebook. 2019. Zstandard - Fast real-time compression algorithm.

[17] Google. 2019. Snappy - A Fast Compressor/Decompressor.

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

[19] 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.

[20] 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.

[21] 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.

[22] Jean loup Gailly and Mark Adler. 2017. Zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library.

[23] 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.

[24] 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.

[25] MongoDB. [n. d.]. Online reference. *http://www.mongodb.com/* ([n. d.]).

[26] Oracle. 2018. Introducing Oracle Database 18c. *White Paper* (2018).

[27] OrientDB. 2019. OrientDB Documentation: Auto Sharding Index Algorithm.

[28] 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.

[29] Justin Sheehy and David Smith. 2010. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper* (2010).

[30] 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.

[31] WiredTiger. [n. d.]. Source Code. *https://github.com/wiredtiger/wiredtiger* ([n. d.]).

[32] 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.