CS265 Final Project: LSM-tree gradient descent

Optimizing Memory Allocation Between Memtable, Cache, and Bloom Filters

Mali Akmanalp
Harvard University
mea590@g.harvard.edu

A. Sophie Hilgard
Harvard University
ash798@g.harvard.edu

Andrew Ross
Harvard University
andrew_ross@g.harvard.edu

1. INTRODUCTION

Tuning data systems is hard. Even for systems like key-value stores that only support the most minimal API (`put` and `get`), the possibilities are often overwhelming. The developers of RocksDB [4], a popular and powerful LSM tree-based key-value store, freely admit that "configuring RocksDB optimally is not trivial," and that "even [they] as RocksDB developers don't fully understand the effect of each configuration change" [5]. Additionally, the optimality of a database configuration depends on that database's *workload*, which is rarely known in advance. There has been recent work [3] in determining the optimal memory allocation for bloom filters in LSM trees in terms of worst-case analysis and with respect to a number of basic workloads, but realistic key-value store workloads, which have been analyzed e.g. for Facebook [11], exhibit enormous complexities with respect to time, skewness, and key repeatability which have not been factored in.

Our goal is somewhat ambitious – building on Monkey [3], we seek to optimize not just bloom filter memory allocation but memory allocation across the entire LSM tree; that is, given a total amount of memory M, we want to choose an amount of cache memory M_{cache}, buffer/memtable memory M_{table}, and bloom filter memory M_{bloom} such that $M = M_{cache} + M_{table} + M_{bloom}$ and disk accesses across the LSM tree are minimized. Furthmore, we want to perform this optimization with respect to a diverse set of workloads that we model as stochastic processes.

2. STOCHASTIC WORKLOADS

To benchmark our results, rather than generating workloads which are fixed sets of queries, we define a diverse set of random workload generating classes. Each class is highly configurable and often hierarchical in their probabilistic models. From them, we randomly regenerate workloads with similar high-level characteristics but different queries. This strategy does not necessarily guarantee realism but helps us avoid "overfitting" to a particular set of queries. The workloads we define contain many simple distributions that are standard in the literature, along with more complex, time-varying workloads (inspired by [11] and [2]) that attempt to mimic more realistic settings.

2.1 Simple workloads

Uniform queries will be drawn uniformly from keys $k \in \{0, 1, ..., K\}$, where K is a maximum key (that we explore varying). The case of uniformly distributed queries is often one in which the cache is unhelpful (unless $M_{cache} > K$), but in practice may be unrealistic. Nevertheless, this is the scenario that many analyses assume for calculations of big-O complexity.

Round-Robin queries are drawn deterministically using $k_i = (i \mod K)$, i.e. we iteratively draw each key in sequence, then repeat. This is also a bad case for our key-value store in its default configuration; the fact that a key has been recently written or read is actually a contraindication we will access it again.

80-20 queries (which are considered in [3]) are drawn such that 20% of the most recently inserted keys constitute 80% of the lookups. This is a simple model we will be able to analyze analytically that exhibits more realistic skew.

Zipf queries are distributed according to a Zipf or zeta distribution, where the probability of a given key k is $\propto \frac{1}{k^s}$, where $s \in (1, \infty)$ describes the skewness of the distribution; in the limit $s = 1$, it is uniform with $K = \infty$. Zipf-distributed queries are considered in [7] as another simple proxy for realistically skewed queries.

For all the above queries, when we draw a particular key for the first time, we will insert it into the database as a write, and subsequently we will either look it up or update it with probability w.

Examples of these workloads can be seen in the first row of Figure 1.

2.2 Complex workloads

Discover-Decay queries are distributed according to the following stochastic process, inspired by the Chinese Restaurant process [1] but with time decay: with every passing time step, we draw a number of reads n_r, writes n_w, and updates n_u assuming queries arrive according to Poisson processes with configurable rates:

$$n_r \sim \text{Pois}(\lambda_r)$$
$$n_w \sim \text{Pois}(\lambda_w)$$
$$n_u \sim \text{Pois}(\lambda_u)$$

Poisson processes are a reasonable choice for modeling the arrivals of database queries or in general the number of events that occur in a continuous time interval, and have been shown to be appropriate for database queries [10, 8, 9]; they are the model that emerges if we make no special assumptions (i.e. maximum entropy) about the length of time between queries. A more realistic model might increase and lower the rate to mimic the diurnal and weekly cycles seen in [11], but especially on short timescales, Poisson processes should be much more realistic than assuming a uniform arrival rate (which we did above).

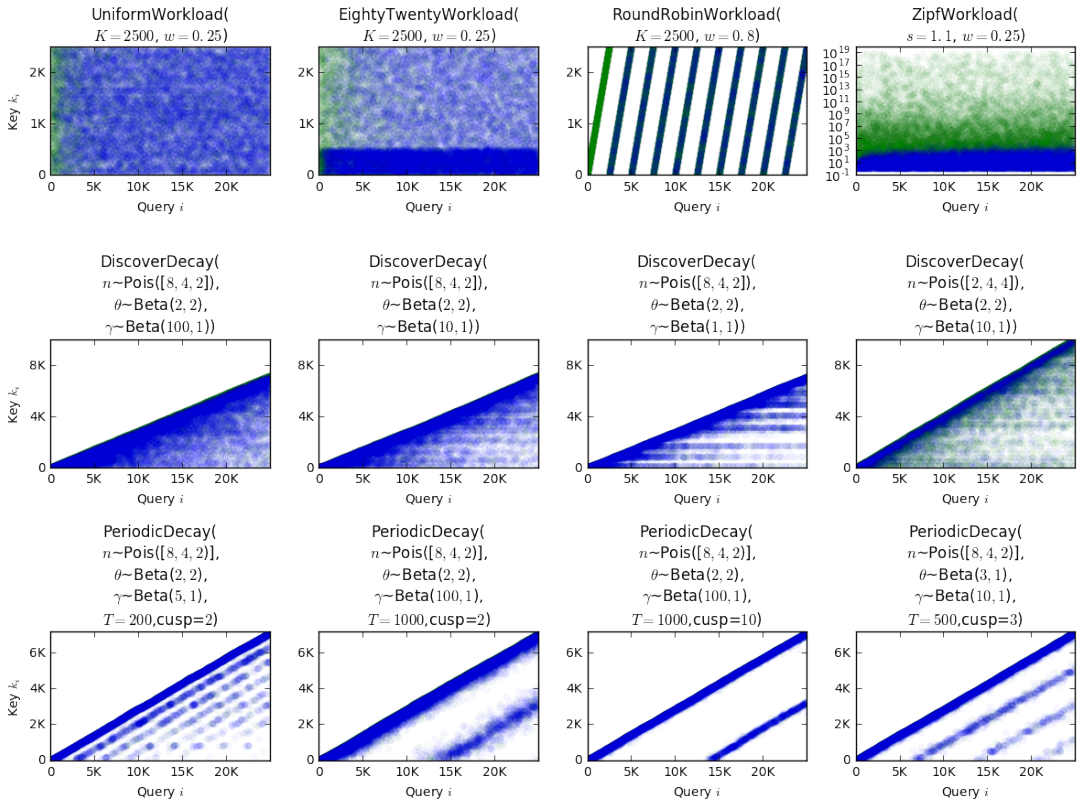

Figure 1: Example workloads we generated for benchmarking. The first row contains simple workloads where the distribution of key popularities does not change over time, and where the read/write ratio is a uniform probability. The second row contains Discover-Decay workloads, which add/read/update keys according to Poisson processes and simulate popularity decays over time. The third row is a modified version of Discover-Decay that adds a periodic signal to the decaying popularity with a configurable period and cusp sharpness. Blue dots represent reads and green dots represent writes or updates.

Once we've drawn our n_w new keys k_i, we assign them an initial popularity

$$\theta_i \sim \text{Beta}(a_\theta, b_\theta)$$

with a random decay rate

$$\gamma_i \sim \text{Beta}(a_\gamma, b_\gamma),$$

which is the factor by which they exponentially decay each subsequent time step. At any time t, the popularity of each key is given by $p(k_i, t) \propto \theta_i \gamma_i^{t-t_i}$, where t_i is when the key was inserted. We use these time-dependent popularities to draw each of our n_r reads and n_u updates from Mult($\{p(k_i, t)\}$). Examples can be seen in Figure 1.

Periodic Decay workloads are a simple modification of the Discover-Decay model where $p(k_i, t)$ now depends not only on the decay rate γ_i but also on a periodic function of the key's age $t - t_i$. To mimic the combination of exponential decay and sharp periodic peaking we see in [11] (from which we reproduce the relevant plot in Figure 2), we multiply $\theta_i \gamma_i^{t-t_i}$ by an inverse cycloid function with period T, clamped from 0 to 1, and taken to a configurable power (to make the cusps sharper or duller) that we call the cycloid's `cuspity`. Examples can be seen in row 3 of Figure 1.

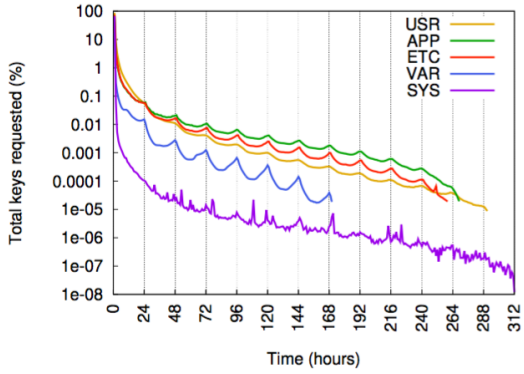

Figure 2: Key reuse histogram from [11] showing the popularity of keys requested multiple times as a function of the number of hours since the previous access for a number of real key-value store workloads from Facebook. Note the overall trend of exponential decay with a sharply periodic subsignal on a 24-hour scale (for most workloads).

3. THE WORKLOAD MATTERS

The first important takeaway is that how much memory we should allocate to the cache, buffer, and bloom filters is highly dependent on the workload.

To investigate this, we wrote Python code to simulate how an LSM tree with a variably sized cache, memtable, disk layers, and bloom filters performs for an arbitrary sequence of queries [6]. In particular, we count disk accesses (the main performance bottleneck for an LSM tree) as well as statistics about the utility of each component. For the experiments below, we started from an empty LSM tree, but future experiments should pick a sensible intermediate state.

Figure 3 shows how the number of disk accesses (on a log color scale) changes as we run a full simulation along different allocations of 8000 bytes (corresponding to the amount of memory needed to store 1000 64-bit entries, and assuming a page size of 2048 bytes), with about 50000 queries for each simulation. We allocate these 8000 bytes using a 400-byte grid spacing along the simplex of possible splits, and test using both baseline (equal bits-per-entry) and Monkey bloom filter allocations [3].

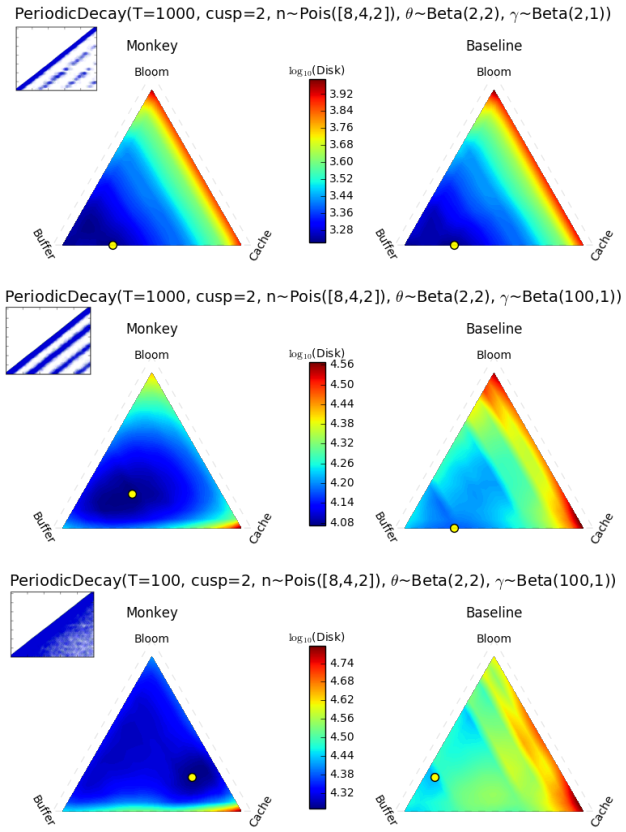

Figure 3: Sets of simulation results for three variants of the Periodic Decay workload (shown in the top-left corner of each set of plots). The minimum, plotted as a yellow dot, is determined by exhaustive search over the simplex of possible allocations. Note how the best allocation is highly dependent on the workload, and how Monkey always outperforms the baseline allocation.

We will include results for many more workloads than just the Periodic Decay variants shown in Figure 3, but even just those three are sufficient to illustrate a variety of takeaways. For the top plot, whose workload consists primarily of reads of very recently added keys but with a few old popular keys recurring at intervals, we obtain the best results by allocating most of our memory to the buffer (for reading recently added keys), but saving a little bit for the cache (for the old but popular recurring keys). The bloom filters are actually useless to us, and as such, it does not matter whether we use Monkey or not to allocate our bloom filters.

The middle plot shows a case where old keys recur in greater numbers (because the decay rates are lower), so the cache alone does not suffice, although it still helps for the most popular old keys (since there is still a spread). Now we transition into a regime where the bloom filter also helps, but only when we use Monkey to allocate memory optimally across the filters. The baseline allocation is not quite good enough to move the optimum away from the pure cache and buffer configuration (and the optimization landscape is starting to look non-convex).

The bottom plot, which has a much higher periodic frequency along with slow decay, shows non-convexity or multimodality in the Monkey results; one local minimum involves allocating a significant amount of memory to the cache and less to the buffer, while another local minimum allocates memory to the buffer (plus slightly more bloom) and much less to the cache. The multimodality makes sense, because our LSM tree affords us multiple strategies for avoiding disk accesses on popular keys; one is to focus on recent keys, which are more likely to be popular, and another is to focus on keys which remain popular for a long time. Interestingly, in the baseline case, the cache strategy no longer works well, perhaps because it implicitly relied on the bloom filters in some way. Note also the discontinuities that occur along the buffer direction, which are a result of the number of layers sharply changing with the buffer size.

These results demonstrate (1) there are many qualitatively distinct regimes of optimality in an LSM tree, (2) all components can be important, (3) Monkey consistently outperforms the baseline bloom filter allocation, even on fairly exotic workalods, and (4) due to the possibility of non-convexity, globally optimizing memory allocation might be challenging.

However, the resulting surfaces suggest that at least for some workloads, local optimization (similar to gradient descent) can be effective for moving toward an optimal memory allocation without requiring that we iterate through all possibilities. Additionally, we note that to compute iterative local optimizations, we do not require that the entire workload be known a priori. Rather, we can collect minimal statistics throughout the query execution process and use these to estimate the current value in saved I/Os any marginal byte of the cache, bloom filters, or memtable is providing. To formulate the useful statistics, we turn to modeling.

4. MODELING

We first consider the case of a uniform query distribution and then show how the formulation can be generalized to any distribution with an empirical trace.

4.1 Uniform query distribution

Assuming we have

- N items in total DB
- E size of an entry in bits
- M total memory
- M_c memory allocated to cache
- M_{buffer} memory allocated to buffer
- B size of the buffer in pages
- P entries that fit in a disk page
- T ratio between layers of LSM tree such that
- $L1 = T * B * P$, $L2 = T^2 * B * P$, and so on,

then we can solve for L the total number of layers required to store all the data:

$$B * P * \frac{1 - T^L}{1 - T} = N$$

$$L = \lceil \log_T \left(\frac{N(T-1)}{PB} + 1 \right) \rceil$$

The average cost of a write remains the same as for the basic LSM tree case:

$$\text{write cost} = \log_T \frac{N}{BP}$$

The average cost of a read must be considered probabilistically over all possible locations of the read item, in this case assuming a uniformly random distribution of reads:

- Probability that read is in memtable = $p(\text{MT}) = \frac{B*P}{N}$
- Probability that read is in cache = $p(\text{cache}) = \frac{M_c/E}{N}$
- Probability that read is in L1 but not in cache = $p(L1)$

$$= \frac{B * P * T - \frac{B*P*T}{N-B*P} * M_c/E}{N}$$

where the numerator is the number of items $B*P*T$ that are in the first layer minus the proportion of items from that layer that are probabilistically in the cache already:

$$\frac{B * P * T}{N - B * P} * M_c/E$$

and finally where the $N - B * P$ comes from the fact that items already in memtable (L0) are not allowed to occupy the cache.

Therefore, given a uniform query distribution, the full expected cost in disk reads of a read is

$$E[C_{\text{uniform}}] = p(\text{MT}) * 0 + p(\text{cache}) * 0 + \sum_{i=1}^{L} p(L_i) * i$$

$$= \sum_{i=1}^{L} \frac{B * P * T^i - \frac{B*P*T^i}{N-B*P} * M_c/E}{N} * i$$

4.2 Bloom Filters

The previous analysis hasn't yet accounted for the presence of Bloom filters, which reduce the likelihood we will unnecessarily access a lower layer. For a Bloom filter of k bits with h independent hash functions $h_1, h_2, ...h_h$, the probability that a given bit is still set to 0 after inserting n keys is

$$(1 - \frac{1}{k})^{n*h}$$

Then the probability of a false positive is

$$(1 - (1 - \frac{1}{k})^{n*h})^h \approx (1 - e^{-hn/k})^h$$

We can minimize this over h to find the optimal number of hash functions, which is $h = \ln(2) * \frac{k}{n}$. Assuming that this is the number of hash functions h we will use, the probability of a false positive as a function of the number of bits is then

$$(1 - e^{-\ln(2)*k/n*n/k})^{\ln(2)*\frac{k}{n}} = (\frac{1}{2})^{\ln(2)*\frac{k}{n}} \approx (.6185)^{\frac{k}{n}}$$

For an item in any any level L_i of the LSM tree with $i \geq 2$ we can reduce the expected cost of accessing that item from i by the number of Bloom filter negatives at any level $j < i$.

Then the expected cost of accessing an item at L_i is

$$\sum_{j=1}^{i-1} p(fp_j) * 1 + 1$$

Where $p(fp_j)$ is the probability of a false positive for that key at level j and 1 is the cost of actually accessing the item at level i assuming fence pointers that lead us to the correct page.

4.3 Expected Cost with Bloom Filters - Base Case

Assuming a random distribution of reads, we now consider also the probability that a bloom filter allows us to ignore a level:
Expected cost of read for an item in the tree =

$$p(mt) * 0 + p(cache) + 0 + \sum_{i=1}^{L} p(Li) * \sum_{j=1}^{i-1} p(fp_j)$$

Expected cost for a null result read = $\sum_{j=1}^{L} p(fp_j)$

Given a total memory allocation M, the total number of bits we can allocate to bloom filters is $M - M_c = \sum_{i=1}^{L} m_i$
Then the total formula for the expected cost of a read in the tree is:

$$E[c] = \sum_{i=1}^{L} \frac{B*P*T^i - \frac{B*P*T^i}{N-B*P} * M_c/E}{N} \cdot \left[\left(\sum_{j=1}^{i-1} (.6185)^{\frac{m_j}{B*P*T^j}} \right) + 1 \right] \quad (1)$$

Whereas with a given percentage of null reads in the workload p_{null}:

$$E[c] = (1 - p_{null}) \sum_{i=1}^{L} \frac{B*P*T^i - \frac{B*P*T^i}{N-B*P} * M_c/E}{N} \cdot \left[\left(\sum_{j=1}^{i-1} (.6185)^{\frac{m_j}{B*P*T^j}} \right) + 1 \right] + p_{null} \sum_{j=1}^{L} p(fp_j) \quad (2)$$

$$E[c] = \sum_{i=1}^{L} (1 - p_{null}) \frac{B*P*T^i - \frac{B*P*T^i}{N-B*P} * M_c/E}{N} \cdot \left[\left(\sum_{j=1}^{i-1} (.6185)^{\frac{m_j}{B*P*T^j}} \right) + 1 \right] + p_{null} \cdot p(fp_i) \quad (3)$$

4.4 Gradients of Cost with Bloom Filters - Generalized Distribution

4.4.1 Cache Gradient

Note that in the above, the workload specific factors are the probability that a read is at any given level and the related probability that any given item from a level is already in the cache. To compute an empirical estimation of the probability that any given item is in a layer but not already in the cache, we can simply keep statistics on the total number of times a key was found in that layer divided by the total number of (non-null) read queries executed. Then we can consider the following simplification:

$$E[c] = \sum_{i=1}^{L} (1 - p_{null}) \left[p(L_i) - \frac{p(L_i)}{(N-BP)} * M_c/E \right] \cdot \left[\left(\sum_{j=1}^{i-1} (.6185)^{\frac{m_j}{B*P*T^j}} \right) + 1 \right] + p_{null} \cdot p(fp_i) \quad (4)$$

Taking the derivative with respect to the number of entries in the cache, M_c/E, we get

$$\sum_{i=1}^{L} -(1-p_{null}) p(L_i)/(N-BP) \cdot \left[\left(\sum_{j=1}^{i-1} (.6185)^{\frac{m_j}{B*P*T^j}} \right) + 1 \right]$$

Which is just the average cost of a read throughout the tree. Then, to keep statistics on how valuable we expect the cache to be, we maintain statistics on the average cost of every read performed in the window of interest.

4.4.2 Bloom Filter Gradients

Because the memory allocation problem is discrete anyway, we consider the value of the bloom filters as a finite difference, that is the approximate value of any marginal bloom filter bit at layer k will be $E[c|m_k + 1] - E[c|m_k]$. In this computation, all terms in the sums drop out except for those concerning m_j, and we are left with:

$$\sum_{i=k}^{L} (1 - p_{null}) \left[p(L_i) - \frac{p(L_i)}{(N-BP)} * M_c/E \right] \cdot \left\{ \left[\left((.6185)^{\frac{m_k+1}{B*P*T^j}} \right) + 1 \right] - \left[\left((.6185)^{\frac{m_k}{B*P*T^j}} \right) + 1 \right] \right\} + p_{null} \left((.6185)^{\frac{m_k+1}{B*P*T^j}} - (.6185)^{\frac{m_k}{B*P*T^j}} \right) \quad (5)$$

Rearranging terms, we get:

$$\sum_{i=k}^{L} \left[(1 - p_{null}) \left[p(L_i) - \frac{p(L_i)}{(N-BP)} * M_c/E \right] + p_{null} \right] \cdot \left((.6185)^{\frac{m_k+1}{B*P*T^j}} - (.6185)^{\frac{m_k}{B*P*T^j}} \right) \quad (6)$$

Where this is exactly the number of times the given bloom filter is accessed times the difference in the theoretical false positive rates given memory allocations m_j and $m_j + 1$. Then, to keep statistics on how valuable we expect any given bloom filter to be, we maintain statistics on the number of times every bloom filter was accessed in the window of interest.

4.4.3 Buffer Gradient: Gets

To estimate the additional value of any marginal memory in the buffer with respect to reads, we must make a number of simplifications, as B, the number of pages in the buffer, factors into every term in this equation. Further, the interaction between B and most of the terms is not available in closed form, in general. Rather, the critical terms $P(L_i)$ we are empirically estimating. Then, for reasonably large values of N and B, we will assume that the bloom filter false positive rate stays approximately the same, as does the value of the cache. Then, we consider only the change in I/Os occurring from the altered probability of any given element occurring in any layer as a result of more elements being in the memtable. We can provide a simple estimate of this by assuming that any items we add to the memtable would have otherwise occurred in L1, and in the resulting cascade, T^i times that number of items will be moved up into each layer L_i from the layer below.

Then, an appropriate estimate of how useful any additional space of memory in the memtable is for reads is simply the resulting change in $p(L_i)$ for each layer (that is, the number of hits we expect to see on the newly added elements) $*fp_i$ for any layer $i \neq 0$, as the original cost of accessing that element was $\sum_{j=1}^{i} fp_j + 1$, and the new cost of accessing is $\sum_{j=1}^{i-1} fp_j$, the difference between which is just fp_i. For $i = 0$, the buffer itself, the expected savings per hits is exactly 1, as the item will be moved from having an access cost of 1 to 0. To estimate how many additional times L1 would be accessed if we instead allocated the final portion of the memtable to L1, we keep statistics on how often the final spots of the memtable were accessed in a read. In practice, these spots are accessed only very infrequently, as the buffer is accessed only a handful of times at this stage before being flushed. This statistic might be more helpful on a system with constant compaction rather than a full layer flush. For the rest of the layers, we simply assume the same hit rate per key as measured over the existing keys on any level and multiply by the number of elements we will be adding to calculate the expected accesses to the new keys on each level. We then multiply by the empirical rate of bloom filter false positives on the level.

4.4.4 Buffer Gradient: Puts

For the buffer, we must additionally consider the saved update/insert I/Os.

$$\text{write cost} = \log_T \frac{N}{BP}$$

Taking the derivative with respect to BP, the number of

items in the buffer, we get $\frac{1}{BP}$ In discrete terms, this evaluates to $\log_T \frac{BP}{BP+1}$.

Unfortunately, this simplification only works if we can assume that memory is being allocated in page-size chunks and that the workload has no duplicates. In practice, the number of I/Os associated with reading and writing throughout the merging process is a stepwise function that depends on page size, as reading or writing one element from or to a page has the same I/O cost as reading or writing a full page. To simplify our analysis of the page size write savings, we consider only a ratio of $T = 2$, and we begin by addressing the case wth no duplicates.

With no duplicates, the final number of elements at any level of the tree is a deterministic function of the number of elements inserted as well as the layer sizes. Then considering the empirical number of items inserted into the buffer as well as the size of the original buffer, we can solve for the theoretical final structure of an alternate LSM tree that had a buffer of size $BP + 1$.

Additionally, given the number of elements on any given layer, no duplicates, and an original buffer size $BP + 1$, we know the number of times each $T^i * (BP + 1)$-size chunk on each level will have been read and written given the current fullness of the layer. We can then multiply these numbers of known chunk reads and writes by the ceiling of the size of those possible chunks (which, with ratio $T = 2$ will be $T^i * (BP + 1)$ and $T^i * (BP + 1) * 2)$ divided by pagesize, P. This gives us a more realistic number in which additions of less than a pagesize of memory are not helpful in I/O savings.

Comparing the read and write costs of this theoretical tree to the empirical reads and writes accesses of the existing tree gives us an expected I/O savings related to updates for the larger tree.

We consider additionally the fact that I/O savings are in general lessened by the number of duplicates inserted, as duplicates will not be merged the full length of the tree. To take this into account we also keep a statistic for the total number of duplicates merged over the window of interest per layer and use this to calculate the percentage of duplicates removed relative to total keys at each level. This factors in in several places. First, when computing the theoretical allocation of keys in the final tree, we consider the total number of items that would have come in to the memtable from the empirical count and adjust this at each layer by the percentage that are expected to have been removed as duplicates. Further, when computing read and write I/Os during merging, we expect that number of items written when the layer is already half full should be decreased by the expected number of duplicates removed among the two sets of keys. Again, the resulting I/O savings will be stepwise in pagesize. In particular, if the original size of the array would have only been slightly into the final page, it will take very few duplicates to reduce the I/O count by 1, whereas if all pages would have been full, it will take a full page's worth of duplicate removals to improve I/Os. The same savings will be experienced again when these items are read to be merged into the lower layer.

The correct way to handle the duplicates requires somewhat more consideration, but the only statistics we are currently using The are the empirical number of update queries and the empirical number of duplicates found and removed on each layer over the window of interest.

4.5 Estimating Statistics with $O(1)$ Memory

Cache: to estimate the number of disk accesses we will save by adding dM extra bits of memory to the cache, we let consider dM as a number of extra entries in the cache. That is, we calculate the savings from having dM/E extra cache entries available. As mentioned above, the relevant statistic here is the average cost of a read in the database. To calculate this, we collect statistics on the total number of disk accesses and total number of queries. The expected cost per query is then the number of disk accesses over the window divided by the total number of queries. To approximate the probability of the item being in the cache times the number of queries, we maintain a statistic for the number of times the last cache slot was accessed during the window of interest and make the assumption that the number of hits on the next marginal slot(s) would be approximately the same. Then we can calculate the final expected I/O savings as

$$dM/E * E[hits] * E[cost/query]$$

Bloom Filters: To estimate the number of disk accesses we will save by adding dM extra bits of memory to the bloom filters, we first decide how to allocate that $M'_{bloom} = M_{bloom} + dM$ bits using Monkey or the baseline allocation, giving us m_i and m'_i bits per bloom filter on each layer. At each layer i, for both m_i and m'_i, we update rolling averages of the theoretical false positive rate $\hat{fp}_i = \mathbb{E}\left[0.6185^{\frac{m_i}{n_i}}\right]$ and $\hat{fp}_i' = \mathbb{E}\left[0.6185^{\frac{m'_i}{n_i}}\right]$ every time the bloom filter is queried (where n_i is constantly changing based on insertions and flushes of the filter). These statistics (individual floats) give us an estimate of the aggregate false positive rate at m_i and m'_i robust to changing layer fullness. Finally, we keep a counter $n_{i,\text{bloom false}}$ of the number of times requested items are *not* in bloom filter i. This counter is incremented either when the bloom filter returns false (which we know immediately) or returns a false positive (which we can record after fruitlessly searching the layer). This counter allows us to estimate disk accesses resulting from our current or altered false positive rates. The final savings is therefore

$$\text{Savings}(M'_{bloom}) = \sum_i (\hat{fp}_i' - \hat{fp}_i) * n_{i,\text{bloom false}},$$

and only requires keeping two floats and one integer. Note that in our simulation, for flexibility, we keep a histogram of n_i values at each bloom filter request to avoid needing to predetermine m'_i, but in a practical implementation this is unnecessary.

Note that because we can obtain these estimates on a layer-by-layer basis, we can investigate whether reallocating memory from one bloom filter to another, empirically, should reduce I/Os. Validating the results of Monkey [3], in Figure 4 we find that for the baseline allocation, moving bits does improve performance, but for Monkey, it does not, regardless of workload.

Buffer: To estimate the number of disk accesses we will save in reads by adding dM extra bits of memory to the buffer, we use statistics maintained on the total bloom filter accesses per layer, bloom filter false positives per layer, and hits per layer. We estimate the expected additional number of hits on any given layer as the original hits times the new theoretical size divided by the actual original size. That is,

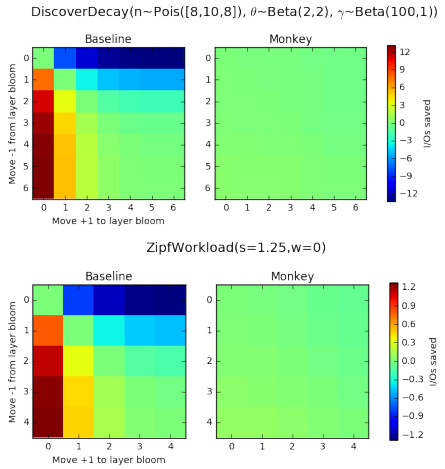

Figure 4: Estimated change in I/Os when moving bits from one bloom filter to another (keeping total bloom filter memory constant). Regardless of workload, changes in I/Os for Monkey are all less than 1, indicating its optimality.

the number of extra hits is equal to

$$new\_hits_i = hits_i * \frac{size_i + dM * T^i}{size_i}$$

For each expected hit, we have an I/O savings equal to the false positive rate on the bloom filter of that layer, as described in the previous section. To calculate this for a layer i, we use

$$E[savings/hit]_I = \frac{false\_positives_i}{bloom\_accesses_i}$$

Then the total number of I/Os saved should be

$$\sum_{i=0}^{L} new\_hits_i * E[savings/hit]_i$$

where for layer 0, the buffer, the $E[savings/hit] = 1$, as the access cost at L1 is always exactly 1 and the access cost at the buffer is always 0.

To estimate the number of disk accesses we will save in writes/updates by adding dM extra bits of memory to the buffer, we maintain statistics on total number of entries that passed through any given layer, number of duplicates removed at any given layer, and number of entries in any given layer at the end of the period. For a workload without duplicates, we can simply use these statistics to deterministically calculate the final allocation and number of read and write I/Os that would have occurred throughout the process for a second tree with buffer size + dM, calculating every batch of read and write merges and summing over the number of pages that would have been involved. For the original tree we can either use statistics on empirical I/Os during the merging process or use the same deterministic formula to calculate what they would have been. The expected saved I/Os then is simply

$$cost_{tree} - cost_{tree+dM}$$

When we consider duplicates, the estimate becomes much more noisy. To consider the effect of duplicates on reducing the total number of pages read and written during the merging process, we reduce the number of entries that pass through each layer of our theoretical larger tree by the percentage of duplicates removed at each layer, calculated as

$$\frac{duplicates\_removed_i}{total\_entries_i}$$

This then changes the final layer structure of the estimated tree. We also consider that duplicates should reduce the total number of entries written and then read after two segments are merged together. Then for those read and write components that occur on an already half-filled layer, we reduce the number of elements by multiplying by

$$1 - \frac{duplicates\_removed_i}{total\_entries_i}$$

This will reduce the total I/Os by number of page reads it makes unnecessary. With this adjusted cost for the larger tree, we again calculate the expected saved I/Os as the estimated I/Os of the hypothetical larger tree subtracted from the empirical or theoretical I/Os of the existing tree.

4.6 Testing Accuracy and Variance of Statistics

To confirm that our estimates are reasonable, we ran 250 simulations for three separate workloads and compared our estimates of each gradient to the actual savings for a separate tree with 8 bytes of extra memory in the corresponding LSM component (against which we ran the same workload). Results can be seen in Figure 5.

There is a large amount of variance in the simulated results, both because of randomness in the separate instantiations of the workload and randomness in the execution of its queries, but for the most part, our estimates of the average savings are both precise and accurate. There is a slight deviation for the uniform memtable savings calculation, but the variance is so high that it does not appear to be significant.

The fact that our estimates of the expected I/O savings are so precise across workloads gives us confidence first that our simulation and modeling are correct, and second that they will generalize to more complex, real-world workloads with more queries and keys.

5. DATABASE GRADIENT DESCENT

Now that we have validated the accuracy of our basic gradient estimates, we evaluate all three of them at every grid point along the simplex of simulated LSM trees with constant total memory. We then overlay an arrow on top of the disk access contour plot pointing from the lowest gradient component to the highest gradient component (signifying that our method suggests moving 8 bytes from one component to the other). Finally, for each grid location, we follow the arrows until we either reach the edge of the simplex or a point we have already reached. We then plot an orange dot. This process simulates what would occur in a discrete, stochastic form of gradient descent. To evaluate how well our adaptive optimization procedure would work, we can visually inspect the orange dots, the arrows, and the yellow dot (signifying the experimental global minimum) and see how well they agree.

Results for basic workloads can be seen in Figure 6.

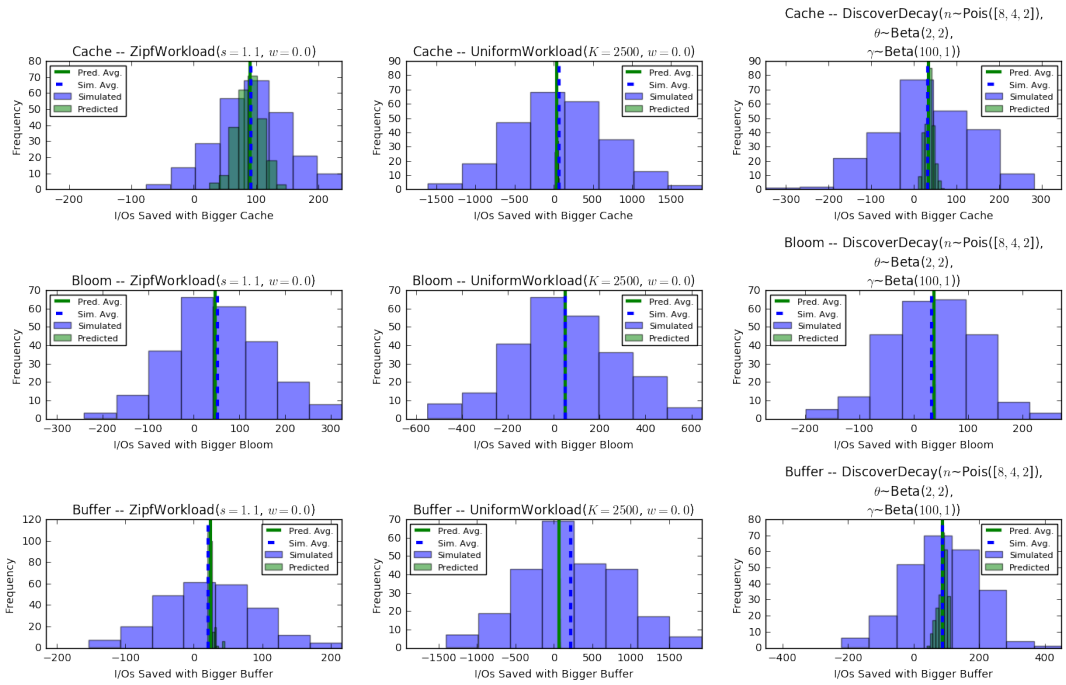

Figure 5: Light-footprint statistical estimations of the gradient vs. simulated results for cache, bloom filters, and the buffer on three distinct workloads.

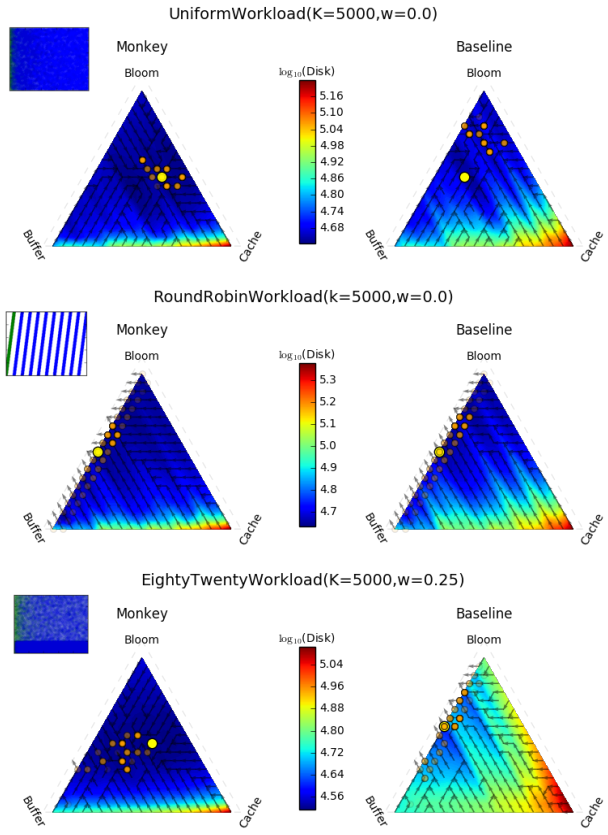

Figure 6: Uniform, Round-Robin, and Eighty-Twenty simulation results overlaid with gradient estimates. Following gradients reliably leads to a good configuration.

Discontinuities in the number of layers as we vary the buffer size makes the optimization non-convex, but Monkey improves both absolute performance and convexity. The arrows track the underlying contours remarkably well, and following gradient estimates leads us to the global minimum in every case except the uniform baseline.

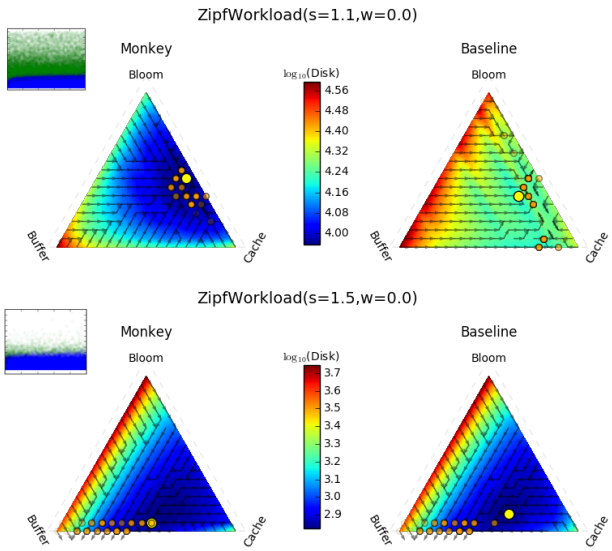

Figure 7: Zipf simulation results overlaid with gradient estimates for $s = 1.1$ (lightly skewed) and $s = 1.5$ (highly skewed).

The results for Zipf workloads in Figure 7 look quite different, but we are also generally able to reach the global minimum, at least in the Monkey case. As we increase the skewness parameter s, we observe a qualitative change in both our simulated results and estimated gradients. At low s, the best configuration is a mixture of mostly Bloom filter and cache memory with a relatively small buffer, while at high s, Bloom filters are less useful and it is better to allocate more memory to the buffer. This effect may be due to the fact for less skewed workloads, we are more likely to request unpopular keys which may be buried deep in the tree (for which we need Bloom filters), but for highly skewed workloads, we obtain better write and read savings by using the buffer as kind of an auxiliary cache.

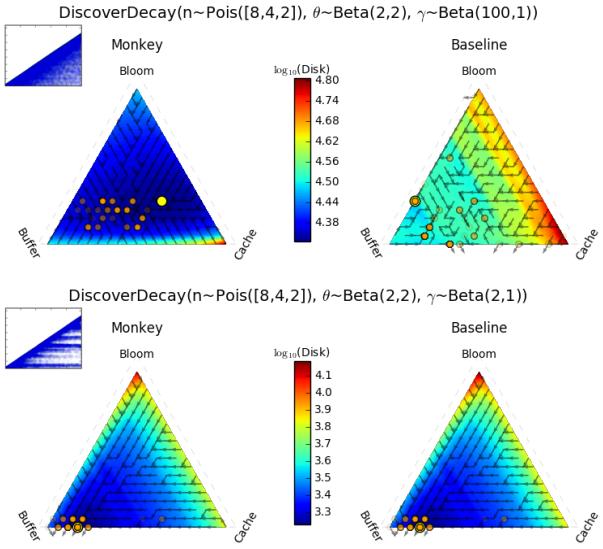

Figure 8: Discover-Decay simulation results overlaid with gradient estimates for $\gamma \sim$ Beta$(100, 1)$ (lightly skewed) and $\gamma \sim$ Beta$(2, 1)$ (highly skewed).

For Discover-Decay workloads (Figure 8), we observe a similar transition (as with Zipfs) as we vary skewness, but with a shifted balance between the cache and the buffer (because we have much more frequent inserts and updates). Again, we reliably find a good local or global minimum following gradient estimates.

Finally, returning to the Periodic Decay workloads, we find that gradients more or less capture the behavior we noted near the beginning of the paper. For lower effective numbers of popular keys (but high temporal locality), we tend to end up allocating most of our memory to the buffer and none to the bloom filters, but as our "working set" expands, we are pushed closer to the center of the graph. In the bottom row, gradient descent is drawn into two distinct modes based on the starting location, suggesting that our gradient estimations are high-resolution enough to capture the nonconvexity. In general (thanks to Niv Dayan for this insight as well as many others in this paper), there are many situations in which increasing either the buffer or the bloom filter will reduce I/Os, so we should expect multiple locally optimal allocation strategies to exist.

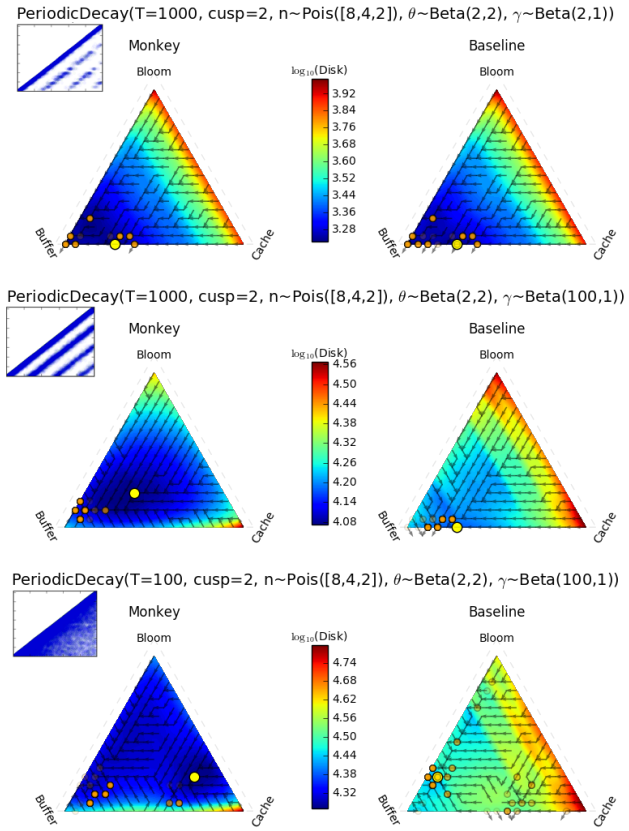

Figure 9: Periodic Decay.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have used both simulation and modeling to analyze how LSM trees perform under a variety of memory allocations and stochastic workloads. In particular, we have shown that the optimal allocation of memory between the LSM tree's cache, buffer, and bloom filters strongly depends on the workload, although we have independently verified the conclusions of Monkey [3] that *given* a total amount of bloom filter memory, there is a unique optimal allocation of it across the bloom filters.

Additionally, we derived formulas for the expected I/O savings if we increase the memory in any component of the LSM tree, along with methods of accurately estimating these savings without using almost any additional memory. We used these estimates to implement a discrete form of stochastic gradient descent for database parameters, and showed that although the optimization problem is sometimes nonconvex, we can usually reach a very good optimum.

As a first next step, we should verify these theoretical and simulated results by testing against a real LSM tree implementation such as RocksDB or LevelDB. We should also see if we can improve the precision of our gradient estimates by storing additional statistics. For example, there are a number of cases where we take rolling averages, but could replace them with histograms instead (in particular, we found it convenient to store a histogram of bloom filter lengths at each access to allow us to calculate the average false positive rate for any number of bits after the fact rather than rolling

averages of expected false positive rates for a few potential memory allocations). Histograms are a much heavier statistic than a simple number but provide a greater flexibility in the reallocation stage. More work needs to be done to calculate how much added benefit could be achieved in practice from these costlier statistics and to find scenarios in which they might be worthwhile.

Additionally, we have not developed a strategy for how to best use our gradient estimates to adaptively change the database in real time. For example, how many queries do we need to see before our gradient estimates become accurate, and what is the optimal time/threshold for reallocating memory? In an environment where we assume full flushing of a given layer and the recomputation of all bloom filters for the layers involved in the flush, it seems like this would be an optimal point to reallocate memory locally among the data structures that are being rewritten anyway. However, this leads to only rare opportunities to recompute deeper layers of the LSM tree, and in practice LSM tree layers are not actually flushed in this 'all-at-once' fashion. A fully fleshed-out policy would need to model the cost of reallocating memory and restructuring the tree alongside our noisy estimates of the gain, ideally using decision-theoretic metrics, to evaluate exactly when and what we should change.

Additionally, just as with gradient descent on continuous loss functions, we need to choose a learning rate, which in this case corresponds to how much memory we should reallocate between components. In this paper, we only considered reallocating the smallest available piece of memory. However, in situations where we have a large amount of conviction in the predictions and they show large potential gains, it might make more sense to reallocate a large block of memory at once, which would help us reach the optimal configuration faster and with fewer restructuring overheads. However, too large a step size might make the optimization procedure less stable and even result in worse performance, at least transiently. More work in modeling, simulation, and experiment is needed to determine a fast but safe descent schedule.

A slightly more far-fetched idea would be to move away from local optimization altogether by determining richer ways of statistically characterizing workloads. For example, we interpreted many of our results, anecdotally, by describing the "temporality" of a workload or the size of its "working set" of keys accessed at any given time. It may be possible that the optimal memory allocation of an LSM tree is a simple, deterministic function of statistically estimable quantities that correspond to these ad-hoc characterizations. If that is the case, then we could do an online analysis of the workload once then jump directly to the optimal configuration.

Finally, we saw in our buffer savings calculation that relative sizes of memory access (i.e. pages) can affect the total I/O savings. In the end, it would be important to consider how the difference in sizes between cache lines, pages on disk, and possibly read chunk sizes from flash would affect our estimates and the optimal allocations. We should also consider fence pointers.

Acknowledgements.

Thanks to Niv Dayan for many helpful suggestions and insights.

7. REFERENCES

[1] D. J. Aldous. Exchangeability and related topics. In *École d'Été de Probabilités de Saint-Flour XIIIâĂŤ1983*, pages 1–198. Springer, 1985.

[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[3] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store, 2017.

[4] Facebook. Rocksdb. https://github.com/facebook/rocksdb, 2017.

[5] Facebook. Rocksdb tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2017.

[6] S. Hilgard, M. Akmanalp, and A. Ross. https://github.com/asross/cs265/blob/master/simulations/lsmulator.py, 2017.

[7] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[8] L. Velcescu. Distribution of time interval between the modifications of result sets cardinalities in random databases. *Analele Universitatii" Ovidius" Constanta-Seria Matematica*, 21(3):295–306, 2013.

[9] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 37–48. ACM, 2002.

[10] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 285–296. VLDB Endowment, 2003.

[11] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 18(2):41–49, 2014.