CS265 Midway Checkin

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 [?], a popular and powerful 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" [?]. Configurations must be optimized with respect to a given *workload*, which is rarely known in advance, although it is sometimes roughly characterizable. There has been recent work [?] in determining the optimal memory allocation for bloom filters 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 [?], exhibit enormous complexity with respect to time, skewness, and key repeatability.

Our goal is somewhat ambitious – we seek to optimize not just bloom filter memory allocation but memory allocation across the entire key-value store (to cache, memtable, bloom filters, and possibly even fence pointers), and to do it with respect to workloads we model as stochastic processes.

2. WORKLOADS

Here are a number of basic workloads we will use to generate queries to benchmark our key-value store in Python simulations. For the simpler ones, we will also attempt to predict performance and derive optimal parameters analytically.

Uniform queries will be drawn uniformly from keys $k \in \{0, 1, ..., K\}$, where K is a maximum key (that we explore varying). When we draw a particular key k_i for the first time, we will insert it into the database as a write, and subsequently we will treat it as a lookup. Later we will explore making a certain fraction of these queries into updates. The case of uniformly distributed queries is often one in which the cache is unhelpful, but in practice is highly 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 [?]) 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 [?] as another simple proxy for realistically skewed queries.

Discover-Decay queries are distributed according to the following stochastic process, inspired by the Chinese Restaurant process [?] but with time decay: with every passing time increment $\Delta t \sim \text{Expo}(\lambda_t)$, $n \sim \text{Pois}(\lambda_n)$ new keys are written to the key-value store, which each have an inherent popularity $\theta_i \sim \text{Beta}(a_\theta, b_\theta)$ with a random decay rate $\gamma_i \sim \text{Beta}(a_\gamma, b_\gamma)$ that determines the exponential rate at which they become less popular. The probability of drawing each k_i is given by $p(k_i, t) \propto \theta_i \gamma_i^{t-t_i}$, where t is the current time and t_i is when the key was inserted. At each time step we sample N keys from $\text{Mult}(\{p(k_i, t)\})$. This stochastic process is somewhat arbitrary and we hope to make it more realistic, but it does capture many of the essential behaviors we know characterize key-value stores: new keys are constantly inserted, some keys are much more popular than others, and the popularity of most keys decays over time. The nonparametric nature of this stochastic process may make inference difficult, and we also hope to enhance it to make it more well-suited to realistic workloads (e.g. that exhibit daily periodicity), but it seems much more able to simulate the richness of realistic queries than many of the other models.

3. SIMULATIONS

We implemented a basic simulator of an LSM tree in Python [?], which simulates how an LSM tree with a variably sized cache, memtable, disk layers, and bloom filters performs for an arbitrary sequence of queries. In particular, we are able to simulate how often we are forced to access data on disk (the main performance bottleneck for an LSM tree) and how often we can respond with data in memory. So far, results suggest that the same LSM tree architecture performs very differently under different query distributions (Figure **??**), and that different LSM tree architectures perform very differently under the *same* query distribution (Figure **??**).

To find the distribution between memtable, bloom filters, and cache that minimizes memory accesses for a given workload, we create a simulation which iterates through the possible allocations of memory given a total memory size, using both Monkey and RocksDB default bloom filter layer allo-

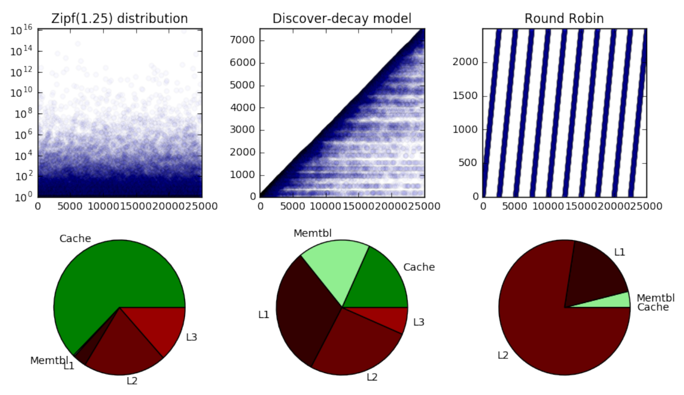

Figure 1: The same LSM tree architecture (a 25-element cache, 100-element memtable, 5x layer ratio, and 10-bit bloom filters with 5 hash functions) performs very differently for different query distributions. Memtable and cache hits (in green) are fast, whereas accesses to the layers (in red) are slow. For the Zipf workload, the cache is much more useful than the memtable, while the situations are reversed for the Round-Robin workload. Both are useful in the discover-decay case.

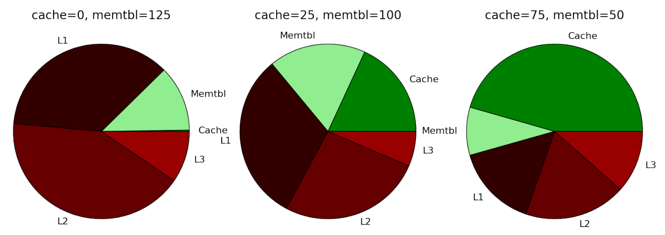

Figure 2: Performance results for different LSM tree architectures on the discover-decay workload. Note that for this time-dependent workload, both the cache and the memtable are useful, but finding the best ratio may require optimization.

cations for a given bloom filter memory size.

The resulting surfaces suggest that for some workloads, local optimization (that is, 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
- P size of the buffer in pages
- B 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 Expected Cost with Bloom Filters - Generalized Distribution

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

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

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.

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.

To estimate the additional value of any marginal memory in the buffer with respect to reads, we must make a number of simplifications, as P, the number of pages in the buffer, factors into every term in this equation. Further, the interaction between P 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 P, 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. Further, we can provide a simple underestimate of this by assuming that any items we add to the memtable will be removed from otherwise occurring in L1, without considering the added value resulting cascade through all other values in all other layers of the table.

Then, an appropriate estimate of how useful any additional space of memory in the memtable is is simply the resulting change in $p(L_i)$ * 1 (as any element at L1 is accessed with a single I/O and no consideration of bloom filter false positive rates). 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.

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 $\log_T \frac{BP}{BP+1}$

And so over the course of the window of interest, we expect to save $\log_T \frac{BP}{BP+1}$ * the number of update queries issued I/Os. The only statistic we need to compute this is the empirical number of update queries over the window.

4.5 Future Modeling Work

What we have done so far is to analytically compute basic quantities (like the expected cost of individual reads) as a function of LSM tree attributes for basic query distributions. Once we extend this analysis to allow the number of layers to vary with the memtable size, we should be able to directly compare these analytic results to simulation and experimental benchmarks, and then finally attempt to derive optimal LSM tree parameters.

5. BRINGING IT ALL TOGETHER

Once we have analytic and simulated predictions of optimal RocksDB configurations that match benchmarking experiments, the final step will be to consider whether we can implement adaptive updating in RocksDB to optimize performance on the fly. We are considering several options, including on-the-fly inference and maintaining a small number of fixed parameter settings that we can dynamically decide between to simplify the problem.

Overall, however, we believe even being able to determine the optimal LSM tree architecture in theory (based on a deep and accurate characterization of the workload) will be a useful contribution.