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

ABSTRACT

To Do

CCS Concepts

•**Information systems → Data management systems;** *Database design and models;*

Keywords

ACM proceedings; LaTeX; text tagging

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 [3], a popular and powerful key-value store, freely admit [4] 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 [2] 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 [7], 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, both in RocksDB and in Python simulations. For the simpler ones, we will also attempt to predict performance and derive optimal parameters analyitically.

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 [2]) 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 [6] 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 [1] 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

CS265 Midway Checkin '97 El Paso, Texas USA
© 2017 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

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 [5], which simulates how an LSM tree with a variably sized cache, memtable, disk layers, and bloom filters performs for an aribtrary 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 1), and that different LSM tree architectures perform very differently under the *same* query distribution (Figure 2). We are working on an optimization procedure to find the best architecture for a given set of queries.

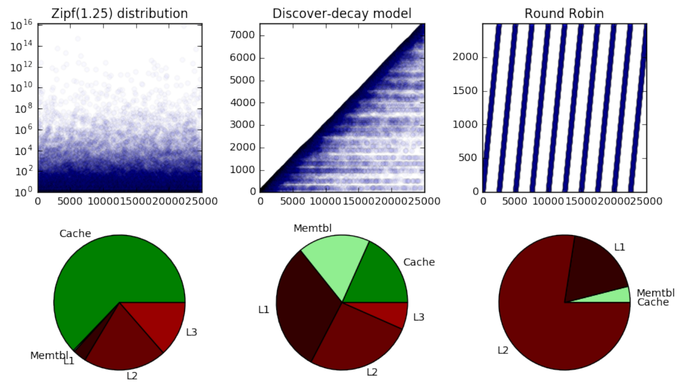

Figure 1: The same LSM tree architecture (a 25-element cache, 100-element memtable, 5x layer ratio, and 10-bit bloom filters with 5 hash functions) performs very differently for different query distributions. Memtable and cache hits (in green) are fast, whereas accesses to the layers (in red) are slow. 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.

4. MODELING

4.1 Uniform

Assume
n_t items in total DB
n_c items that fit in cache
n_m items that fit in memtable
R ratio between layers of LSM tree such that
$L1 = R * n_m$
$L2 = R^2 * n_m \dots$

We can solve for j the total number of layers required to store all the data:

$$n_m * \frac{1-R^j}{1-R} = n_t$$

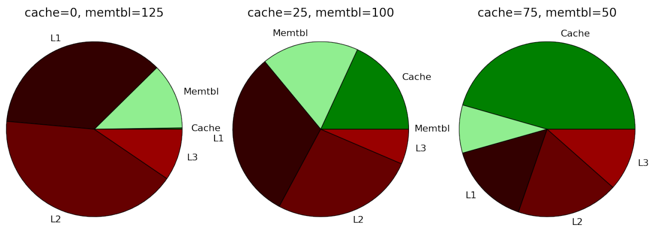

Figure 2: Performance results for different LSM tree architectures on the discover-decay workload. Note that for this time-dependent workload, both the cache and the memtable are useful, but finding the best ratio may require optimization.

$$j = \frac{log(1 - n_t * \frac{1-R}{n_m})}{logR}$$

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

$$\log_R \frac{n_t}{n_m}$$

The average cost of a read, we consider probabilistically over all possible locations of the read item, assuming a random distribution of reads:
Probability that read is in memtable:

$$\frac{n_m}{n_t} = p(mt)$$

Probability that read is in cache:

$$\frac{n_c}{n_t} = p(cache)$$

Probability that read is in L1 but not in cache

$$\frac{n_m * R - \frac{n_m * R}{n_m * \frac{1-(R^j-1)}{1-R}} * n_c}{n_t} = p(L1)$$

Where the numerator is the number of items that are in the first layer

$$n_m * R$$

minus the proportion of items from that layer that are probabilistically in the cache already

$$\frac{n_m * R}{n_m * \frac{1-(R^j-1)}{1-R}} * n_c$$

Probability that read is in L_i generally but not in cache

$$\frac{n_m * R^i - \frac{n_m * R^i}{n_m * \frac{1-(R^j-1)}{1-R}} * n_c}{n_t} = p(L_i)$$

Where here the $R^j - 1$ comes from the fact that items already in memtable (L0) are not allowed to occupy the cache.

Expected cost of read:

$$E[C] = p(mt) * 0 + p(cache) * 0 + \sum_{i=1}^{j} p(L_i) * i$$

$$= \sum_{i=1}^{j} \frac{n_m * R^i - \frac{n_m * R^i}{n_m * \frac{1-(R^j-1)}{1-R}} * n_c}{n_t} * i$$

4.2 Skewed Reads (80-20)

Now consider the case for skewed reads, where we say d_{hf} (d_{lf}) percent of the data receives r_{hf} (r_{lf}) percent of the reads (where $d_{hf} + d_{lf} = 1$ and $r_{hf} + r_{lf} = 1$). On average, we can assume that the cache contains $r_{hf} * n_c$ items from $d_{hf} * n_t$ and $r_{lf} * n_c$ items from $d_{lf} * n$. Then the expected cost of a read is dependent on whether the data item being read is in $d_{hf} * n_t$ or $d_{lf} * n$ as the probability of a cache hit varies.

Concretely, consider where we have 3 levels and 800 total items with a cache of size 10 and a ratio of 2 (for L0=100, L1 = 200, L2 = 400 items), with $d_{hf} = .2$ and $d_{lf} = .8$ and $r_{hf} = .8$ and $r_{lf} = .2$. Then the cache on average contains 8 items from $d_{hf} * n$ and 2 items from $d_{lf} * n$. If we execute a read on one of the 200 items in d_{hf}, then, there is a $\frac{8}{200}$ chance that that item is in the cache. If we execute a read on one of the $200 * \frac{1}{4} = 50$ items of $d_{hf} * n_t$ in L1, we expect that $\frac{2}{6} * 8$ of those items would have actually been found already in cache, as this level contains $\frac{2}{6}$ of all of the items not in the memtable. Then the probability that a read is found in L1 is the proportion of the $d_{hf} * n_t = 160$ items that will reside in L1 but not in the cache, which is $\frac{40 - \frac{2}{6} * 8}{160}$.

For data in $d_{hf} * n$,

Probability that read is in memtable:

$$\frac{n_m * d_{hf}}{d_{hf} * n_t} = p(mt_{hf})$$

Probability that read is in cache:

$$\frac{r_{hf} * n_c}{d_{hf} * n_t} = p(cache_{hf})$$

Probability that read is in L_i but not in cache:

$$\frac{n_m * R^i * d_{hf} - \frac{n_m * R^i}{n_m * \frac{1-(R^j-1)}{1-R}} * r_{hf} * n_c}{d_{hf} * n_t} = p(Li_{hf})$$

Expected cost of read on item in d_{hf}:

$$E[C_{hf}] = p(mt_{hf}) * 0 + p(cache_{hf}) * 0 + \sum_{i=1}^{j} p(Li_{hf}) * i$$

$$= \sum_{i=1}^{j} \frac{n_m * R^i * d_{hf} - \frac{n_m * R^i}{n_m * \frac{1-(R^j-1)}{1-R}} * r_{hf} * n_c}{d_{hf} * n_t} * i$$

The expected cost of a read on an item in d_{lf} can be enumerated analogously, and we combine the expectation of reads in d_{hf} and d_{lf} as:
Expected cost of read $= r_{hf} * E[C_{hf}] + r_{lf} * E[C_{lf}]$

4.3 Bloom Filters

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.

4.4 Variable Cache Size, Constant Number of Layers

To analyze a variable cache/memtable allocation with a given memory size n_v, we first consider the simplification of assuming a constant layer structure. Of course, in general a larger memtable will allow for larger layers, affecting both read and write performance. Let n_l be the size of the first layer when $n_v = n_m$ (all memtable) and $n_c = n_v - n_m$.

Uniform Case:
Consider the formula used earlier:

$$p(mt) * 0 + p(cache) * 0 + \sum_{i=1}^{j} p(L_i) * i$$

In the extreme case where $n_m = 0$ (**no memtable**), the formula in the numerator of the sum simplifies to be over n the total number of items, as there is no memtable layer and the probability of the first layer now has a cost of 1. However, we now have to add a number of items to each level of the tree that sum to the amount that were in L0. We add them as a geometric series per layer to maintain the structure

$$\sum_{i=i}^{j} \frac{(n_l) * R^i + \frac{(n_l)*R^i}{n_t} * n_v - \frac{(n_l)*R^i}{n_t} * n_v}{n_t} * (i)$$

The two latter terms cancel and we are left with

$$\sum_{i=1}^{j} \frac{(n_l) * R^i}{n_t} * i$$

In the extreme case of **no cache**,

$$\sum_{i=1}^{j} \frac{(n_l) * R^i}{n_t} * i$$

And we can see that this is in fact the same result as the no memtable case.

In general, for any selection of n_m and n_c we have

$$\sum_{i=1}^{j} \frac{(n_l) * R^i - \frac{(n_l)*R^i}{(n_l)*\frac{1-(R^j-1)}{1-R}} * (n_v - n_m)}{n_t} + \frac{\frac{(n_l)*R^i}{(n_l)*\frac{1-(R^j-1)}{1-R}} * (n_v - n_m)}{n_t} * i \quad (1)$$

and so with a random workload, the choice is irrelevant *assuming a constant number of layers*. In actuality, with a larger memtable we will be able to decrease the total number of layers needed.

Skewed Reads (80-20) Case:
Now consider the case for skewed reads, where we say d_{hf} (d_{lf}) percent of the data receives r_{hf} (r_{lf}) percent of the reads (where $d_{hf} + d_{lf} = 1$ and $r_{hf} + r_{lf} = 1$). On average, we can assume that the cache contains $r_{hf} * n_c$ items from $d_{hf} * n_t$ and $r_{lf} * n_c$ items from $d_{lf} * n$.

In the extreme case of **no cache**, the result will be as above.

In the extreme case of **no memtable**, the expected cost of a read is dependent on whether the data item being read is in $d_{hf} * n_t$ or $d_{lf} * n$ as the probability of a cache hit varies. For data in $d_{hf} * n$,

Probability that read is in cache:

$$\frac{r_{hf} * n_c}{d_{hf} * n_t} = p(cache_{hf})$$

Probability that read is in L_i but not in cache:

$$\frac{n_l * R^i * d_{hf} - \frac{n_l * R^i}{n_t} * r_{hf} * (n_v)}{d_{hf} * n_t} + \frac{\frac{n_l * R^i}{n_t} * d_{hf} * (n_v)}{d_{hf} * n_t}$$
$$= \frac{n_l * R^i * d_{hf} + \frac{n_l * R^i}{n_t} * (d_{hf} - r_{hf}) * n_v}{d_{hf} * n_t} = p(Li_{hf}) \quad (2)$$

Expected cost of read on item in d_{hf}:

$$E[C_{hf}] = p(cache_{hf}) * 0 + \sum_{i=1}^{j} p(Li_{hf}) * i$$

$$= \sum_{i=1}^{j} \frac{n_l * R^i * d_{hf} + \frac{n_l * R^i}{n_t} * (d_{hf} - r_{hf}) * n_v}{d_{hf} * n_t} * i$$

From this equation we can see that the probability (and thus the expected cost) is increasing in d_{hf} and decreasing in r_{hf}. This makes sense, as for many reads (high r_{hf}) on a small amount of data (low d_{hf}) we would expect frequent cache hits.

For general memtable/cache allocation, probability that read is in L_i but not in cache:

$$\frac{n_l * R^i * d_{hf} - \frac{n_v * R^i}{n_l * \frac{1-(R^j-1)}{1-R}} * r_{hf} * (n_v - n_m)}{d_{hf} * n_t} + \frac{\frac{n_v * R^i}{n_l * \frac{1-(R^j-1)}{1-R}} * d_{hf} * (n_v - n_m)}{d_{hf} * n_t} = p(Li_{hf}) \quad (3)$$

4.5 Allow Number of Layers to Vary with Memtable

4.6 Bloom Filter Allocation

5. BENCHMARKING

So far, we've succeeded in getting basic benchmarks for RocksDB running and identifying which configuration parameters to vary.

6. CONCLUSIONS

7. ACKNOWLEDGMENTS

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

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

[4] R. T. Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide#final-thoughts, 2017.

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

[6] 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.

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