LSM-Based Key-Value Store

CS-265 Big Data Systems

David Kanda
davidkanda@gmail.com

ABSTRACT

This paper presents the design and experiments of the LSM-Based storage system for the CS-265 class. The paper presents multiple design, implementations and experiments that drove the final product, in order to achieve fast writes.

ACM Reference format:
David Kanda. 2019. LSM-Based Key-Value Store. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to the rise in Big Data, multiple storage systems have been design and used in the industry, to handle the influx of data in the amount and speed that the data is being received. This paper looks into the LSM-Based system, a write-optimized data structure in order to analyze and consider the trade-offs of multiple algorithms and data structures that help enable fast writes. This paper explores some of the design decisions taken in order to try to achieve the expected performance. In this paper, we introduce an LSM Tree that implements a skiplist, a further optimization that enable fast in-memory lookup of keys.

As data scales with the advent of Big Data, updates and reads become more costly. Traditional systems do not differentiate between the frequency of access of data, foregoing significant optimization opportunities since, in many applications, users need to access the most recent data the fastest. The LSM tree, provides a mechanism for quick updates, deletes, and writes by collecting them in a pool of active keys before pushing them to secondary storage when the pool is full. By arranging secondary storage in tiers, the cost of merging the buffer to disk is amortized, allowing for efficient writes. Keeping more frequently used keys in memory allows for performant lookups over recent data. The tiered structure of data also provides a natural opportunity for indexing. A variety of indexing structures, including fence pointers and Bloomfilters are commonly used to minimize unnecessary disk accesses.

2 DESIGN

The skiplist-based LSM has two main components to store its keys and values: an in-memory buffer and a disk-based store. The in-memory section is composed of a set of data structures that are optimized for quick insert and lookup on the data based on the assumption the most recently written data is often the most accessed. The disk-based store is composed of a tiered layer storage that scales by a constant factor or growth rate with each tier.

2.1 Skiplist Node

There are multiple implementations of a key-value pair depending on the level of the data store. I chose to implement a skiplist main memory structure in order to achieve the best performance for the data structure due to the skiplists ability to quickly scan the available keys in memory. My implementation of the skiplist includes an array of pointers to other skiplist nodes. This array can be thought of as a vertical column of level pointers, where pointers above the nodeâĂŹs level are null and each pointer below points to the next node on that particular level. In this way, skipping down a level is a matter of reading a value that was already loaded into the cache, rather than a performance hit of chasing a pointer through memory.

2.2 Bloom Filters

Bloom filters are space-efficient probabilistic data structures that are used to test whether an element is in a set. Using a series of hash functions and a bitset, the filter can provide a strong probabilistic guarantee: an element will never induce a false-negative result under a test for membership, and an element will only induce a false-positive result up to some error probability. This false-positive probability value is chosen by the user and is traded off against the space occupied by the filter. Bloom filters find important use in the sLSM when paired one-to-one with runs in memory and on disk. Rather than incur a high cost by searching for a key in every run, the bloomfiler is checked first. If it returns negative, we can safely skip that run, because of the filterâĂŹs no-false-negative guarantee. In this way, weâĂŹll only search a proportion of the runs that have a returned a positive result from the bloomfilter, which could result in a significant time saving for lookups. In our implementation, Bloomfilters are leveraged by pairing each run with a filter test; if the check fails, we simply skip that run. We use the Murmur hash as opposed to slower cryptographic hash functions allowing us to quickly generate the hash values. We also keep track of the maximal and minimal key in each run for low-cost, high-granularity filtering by run.

2.2.1 Disk Levels. Each level has a concept of a block. A block represents a sorted array on disk. A block has information like number of elements per block, min and max keys and indices into the keys within the block. This metadata is stored in disk whenever the sorted array is stored. When a retrieval of a pair is requested the range is checked to ensure that it exists in the block. Once the range is verified, a binary search is performed on the nodes contained in the block to retrieve the corresponding value for the desired key.

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

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

In order to ease building filenames, all names are based on a prefix, sst.

2.3 Sorting and merging

When the buffer becomes full, a fraction m of the runs is flagged and their elements are collected, sorted then written to the shallowest disk levelâĂŹs next available run. In this way, adjacent levels share the following relationship: the size of a levels runs is identical to the total size of its shallower neighbor multiplied by the fraction of runs merged m. A disk level might be full, requiring a cascade of merges down to lower disk levels. This complex operation is quite nuanced: though the runs being merged are individually sorted, the resulting run needs to be sorted. Because runs at lower levels do not get in memory, some optimizations are necessary to save both time and space.

In our implementation, we also multithreaded merging to decrease our latency. When an insertion triggers a merge, a dedicated merge thread takes ownership of the runs to merge and executes the merge in parallel, allowing the main thread to rebuild the buffer and continue to answer queries. If a lookup request comes while the merge thread is executing the merge, the main thread searches the memory buffer for the requested key, and if unsuccessful, waits for the merge to complete before querying the disk levels.

2.4 Range Queries

Range queries involve looking up, for each run, all the elements in the range. For skiplists, this is as simple as locating the node corresponding to the smallest key greater than or equal to the first key in the range. then, simply following the skiplists pointers until the current node is greater than or equal to the second key in the range, or else the end of the list has been reached. For disk-based runs, we first filter by key, then do only the 1 or 2 lookups we need to and get the indexes in the run that frame the range. From there, we construct a hash table as follows. Starting with the newest run and working backwards towards the oldest, and all elements in the range, and for each, 1) insert the key and value into the hash table and 2) if the element is not a delete or already in the table, write it to the result set. The hash table guarantees that only the newest non-deleted values will remain in the result set. For our hash table, we again use the Murmur3 hash function. We use linear probing rather than chaining to optimize for small key-value pairs, and keep true key-value objects in the table, rather than pointers, in order to remain cache-optimal. When the hash table is more than half full, we double its size and rehash each element, leading to amortized O(1) insertion and true constant-time probing. Because collecting elements in the range in both skiplists and disk runs is a linear-time operation, and because hashing is an amortized constant-time operation, a range query over n keys is expected to take O(n) time.

3 EXPERIMENTS

The development and experiments were run on a MacBook Pro 13-inc, 2012 with 2.5 GHz Intel Core i5, with 1 processor with 4 cores. L2 Cache per core of size 256 KB and L3 Cache of 6 MB and memory is 8 GB 1600 MHz DDR3. In order to evaluate the system at the component level as well as overall, a custom benchmark application harness was written to make the queries. More than 1, 000, 000 elements can be inserted in one second with a buffer size of 800 elements.

3.1 Insertion and Lookup Baseline

Table 1: Experiment inserting a large dataset

Number of Inserts	Inserts/Second(s)	Total Insert(s)
1,000,000	1.33E+06	0.754368
5,000,000	1.02E+06	4.91643
10,000,000	1.00E+06	9.95114
50,000,000	8.33E+05	60.0281

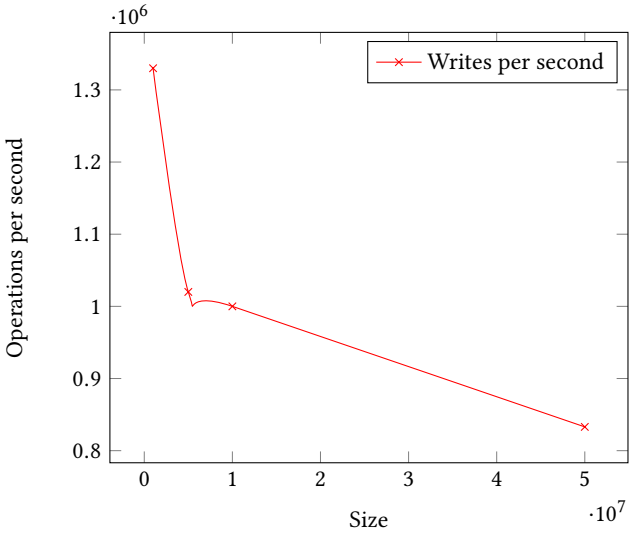


Figure 1: Number of insertions per second vs size of dataset

Table 1, shows insertion times for large datasets ranging from 1 million to 50 million records. The data structure is able to accommodate inserting between 100,000 and 1M values within 1 second as desired. The data displays a 37% speed decrease with respect to insertions per second with a 50% increase in workload size.

Table 2: Experiment reading a large dataset

Number of Reads	Lookups/Second(s)	Total Lookups(s)
1,000,000	51240	19.516
5,000,000	58463.2	106.684
10,000,000	54283.9	184.217
50,000,000	51473	971.383

Table 2, shows read times for large datasets ranging from 1 million to 50 million records. As expected, there are fewer lookups per second than writes per second since the LSM tree has been optimized for writes rather than reads.

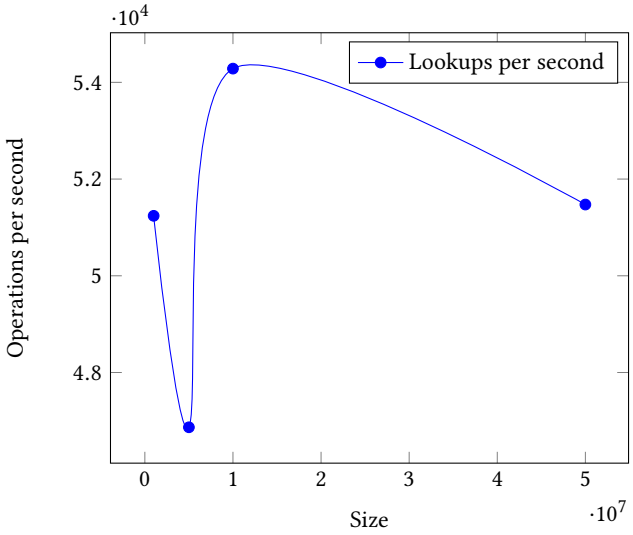


Figure 2: Impact of a large dataset on performance

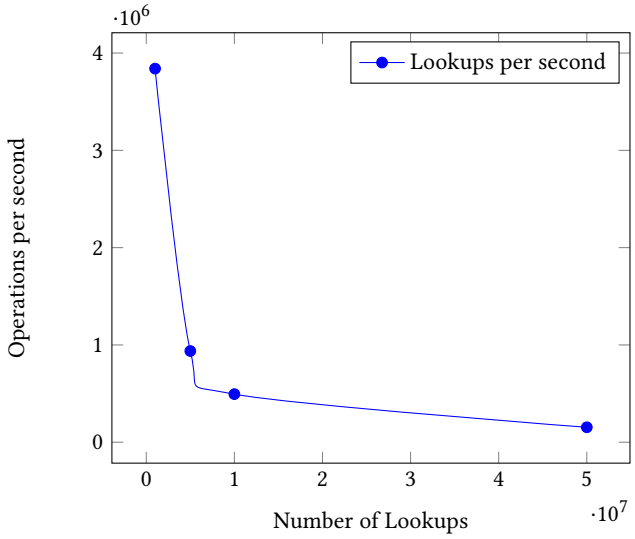


Figure 3: Number of operations per second vs Number of Lookups

3.2 Fractional Run Merging

Table 3 presents an analysis of the performance gained by setting the merge fraction knob. Rather than going through the entire skiplist in main memory and merge it down to the first level on disk, we instead present a fraction of the skiplist to be merged, thus saving the overhead in merging list into the disk run.

Table 3: Experiment with scaled fraction of run merged to disk

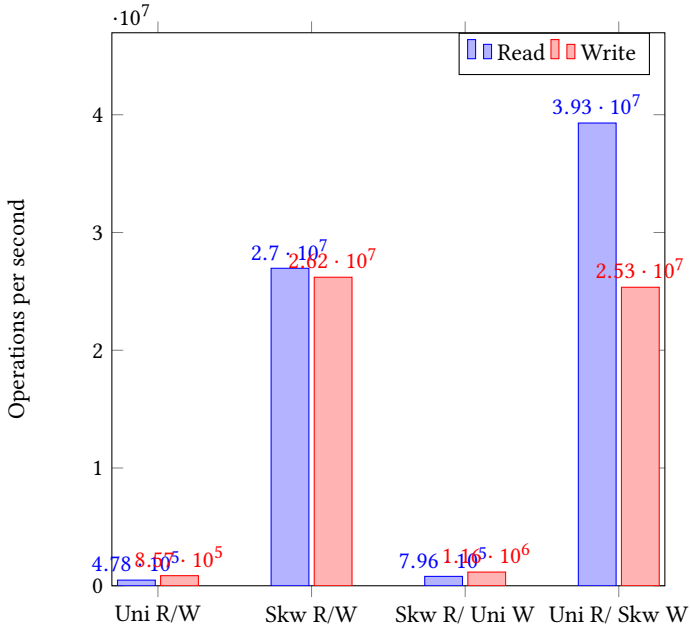
Num of Inserts	Merge Fraction	Inserts/s	Reads/s
1000000	0.25	1.01E+06	79008.9
1000000	.5	1.18E+06	31362.3
1000000	.75	1.15E+06	99358.7
1000000	1	1.30E+06	56733.1

We can see that with smaller merge fractions, the number of insertions per second goes down. This is because that, while there are fewer records being merged, there are more merges occurring when the records are being written. This system allows for a mixture of old an new data to be available in the memory levels which can allow for better lookup time for random lookups. The number of lookups per second tell a different story. Since the memory levels are a mixture of old and new values, the lookup performance is more arbitrary. The reads per second fluctuate greater between the different ratios implying that lookup time can be quite random.

3.3 Insertion vs Lookup Ratio

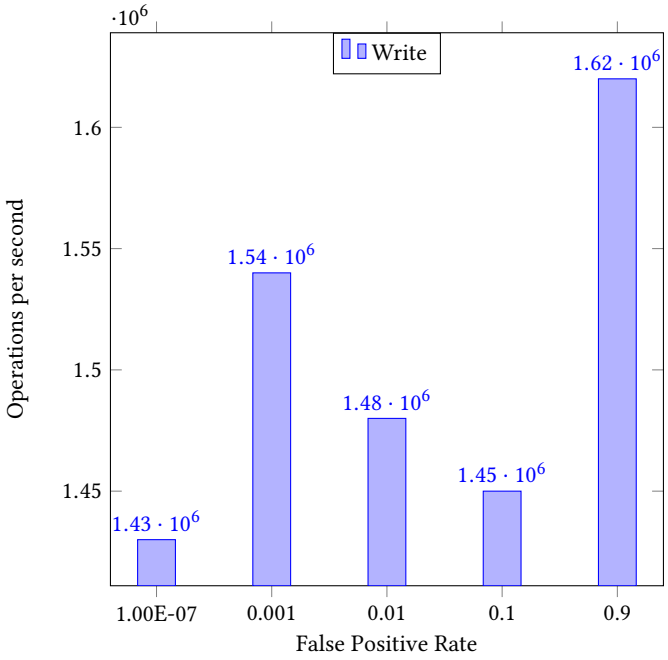
Figure 3. Demonstrates how the number of operations per second decreases as the number of lookups increases given a LSM tree containing 1,000,000 different keys. This decrease makes sense since there might be more keys that need to access the disk to find their corresponding values than simply looking within main memory.

3.4 Uniform vs Skewed Insertion vs Update



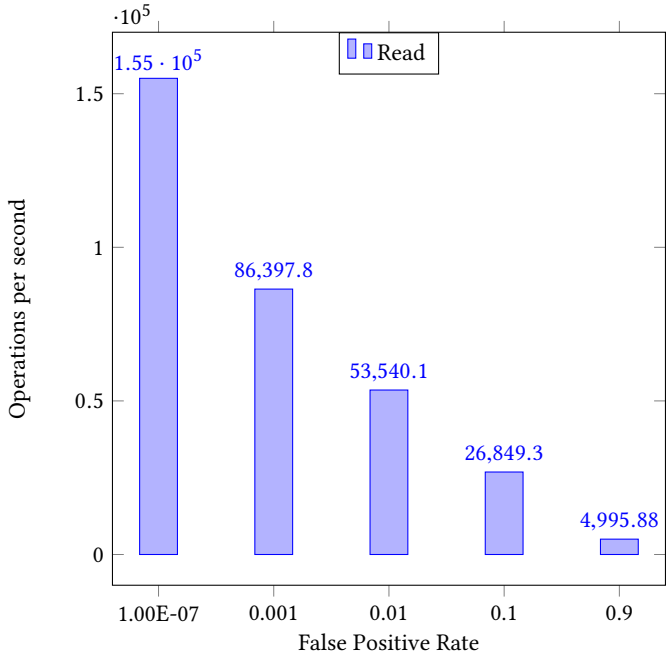
The figure above demonstrates how the distribution of queries against the data structure affects the number of read or write operation effeciencies. Skiplists do not blindly insert elements; rather, if a key is already in the active skiplist, its value is simply updated. This means that for data with low variance of keys, insertion can be incredibly fast. For this experiment, I generate keys via a normal distribution to serve as the uniformly distributed datasets. For skewed data, I created an exponential distribution to generate a non-uniform, exponential distribution around the integer 10,000,000 with which to insert to or write from LSM tree. The

uniform read/write affords a very low number of operations per second since there is a large base of values to pull from. The skewed Read/Write is more interesting. Since both inserts and lookup values are skewed around the same exponential values, it greatly increases the number of operations per second since there is little variety. There are a number of duplicate keys in the tree so it is rare that a value will exist on disk. Instead most values are stored within main memory since, while there are 1 million inserts within the data structure. Only a small number of those million are distinct. The skewed reads and uniform writes demonstrate that the writes dictate number of operations.



The figure shows how a smaller false positive rate has a negative effect on the performance of write operations.

3.5 Bloomfilter False Positive Rate



The figure above represents the operations per second for reads on a LSM tree containing 100,000 entries. As expected, the lower the desired false positive rate, the greater the operations per second achieved. This is because there is a greater key space to store the hashes of incoming keys allowing for a more accurate Bloomfilter existence indication.

3.6 Concurrency

In this experiment, lookup skew was varied along with the number of threads performing concurrent lookups, demonstrating that with highly clustered lookups, the LSMs scaling factor is higher with each thread than with evenly-distributed lookups. This is due to the fact that the disk is able to optimize its seeking to service the lookup requests better when there is a small locality of keys. Perhaps multiple threads could even be serviced by the same disk pages, cutting down on disk operations even further. The closeness of the keys allows the access pattern to act like sequential requests rather than random requests.

The concurrent experiment above was run on a dataset containing 3,000,000 entries with normal key distribution. With the simulation of sequential requests, the number of threads converges at around 3 threads.

3.7 Number of Runs

Skiplists operate by having a set of runs that contain sparse keys and more less sparse keys as the pointers are followed downward. The last run in any skiplist contains all the possible keys in the skiplist. This architecture allows for fast scanning of the contents of the skiplist at the higher levels and slower, more fine-grained scanning at the lower levels. A decrease in the number of runs increases the lookup performance while the insertion performance suffers.

Table 5 shows the effect of scaling the number of runs on an LSM with 1,000,000 insertions. The larger the amount of runs, the less time is spent performing large merges. Traversing a large amount of runs incurs a large lookup cost however. This could be due to

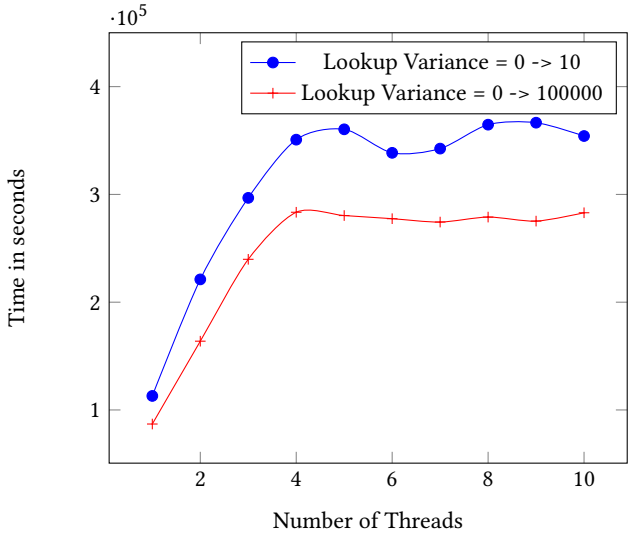


Figure 4: Performance with concurrency converges at 6 threads

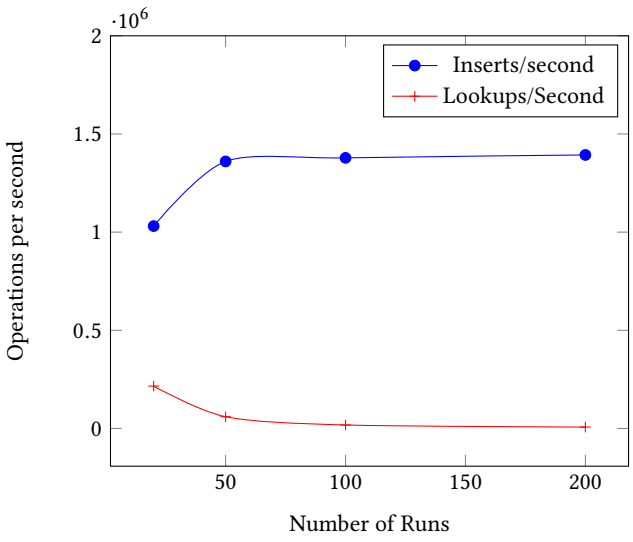


Figure 5: Scaling number of Runs

smaller runs being more succinct and thus quicker to traverse than levels with more runs.

4 CONCLUSIONS

4.1 Future Work

- *Starting for Disk:* Currently, the system does not support restoring from disk. After the program executes and the LSM destructor is called, the sst files are deleted. When to program is started, the sst file cache is cleared, making way for a new database. The logic was to enable quick experimentation without having to delete the old sst files. Restoration from disk will allow the system to be more extensible and using for real-world scenarios.
- *Caching:* A smart caching strategy, specially for Binary Tree nodes loaded from retrieval queries. In the same way, unused nodes should be freed from memory. This is hard case to consider, specially when concurrently there is a query that might potentially cause the node to be loaded into memory again.
- *Buffer on Disk:* In order allow concurrent reads and writes on disk, while performing a flush to the next level on disk, we can keep increasing the number of blocks per level. This would allow to continuously keep receiving flush from top levels, and still check for elements that are being flushed.
- *In-Memory Range queries:* Range queries are not handled very well in memory. One approach was the implementation of a B+Tree in memory, but compared to the disk version, this would have to be immutable, causing latency on writes. One approach is to concurrently add elements as possible, without disturbing writes. Similar to the consumer-producer approach, the buffer produces an index, indicating how many elements have been inserted, and a separate thread has its own index of the elements inserted into the B+Tree and compares against the index in the buffer. If the index from the buffer is bigger, the value is just retrieve given that it does not affect state. Once flushing to the next level is in place, the B+Tree operations should be stopped and the whole B+Tree should be discarded.
- *Disk Range queries:* After certain range query operation, the list of pairs have multiple missing ranges, this causes multiple loads from disk at the next block or level. If the missing ranges are close to each other, a more general range should be used, and then just ignore old pairs in memory, instead of ignoring them from disk.
- *HashMap on Disk:* Based on certain information, we could design a way to store elements in a way that disk retrieval is $O(1)$. The Problem with this approach would be range queries.
- *Flush Threshold:* If the number of elements after deleting duplicates and deletions is less than a threshold, there might not be a need to flush, specially for in-memory levels. There could be a check if the number is beneath this threshold.
- *Distinct elements:* Currently there is no way to count how many distinct elements are stored. Specially when deletions are happening. Pruning the values associated with keys would help to optimize the blocks on disk.
- *In-Memory sorting:* HeapMerge proved to be a very effecient algorithm to merge the runs together. It would be beneficial to explore other merging algorithms and compare their effeciencies.
- *Stats about cache misses:* There is no indication as to how many in memeory cache misses occured. In order to make further decisions about optimizations and explore other data structures, first we need to know how many run and block misses we are experiencing.
- *Smaller Block Sizes on Disk:* Currently when merges happen, the complete block is loaded from disk. Given that the number of blocks increases per level, it might be beneficial to

use a smaller size node for bottom levels, in order to support more effecient merging. In the same way, we could have temporarily files in order to support large, cumbersome merges.

4.2 Take away

- Range queries are costly. We must suffer a disk query in order to load the metadata to find the indicies. Although a scan through the range query is a binary search, traversing each block, and then each level is costly.
- As multiple levels are created, single threads can lock for an extended amount of time.
- Multiple experiments were run in order to find the most ideal parameters, however, there is always room for more improvement.
- The performance of the data structure behaved within the requirements of the spec. For SSD storage we expect your system to be on the order of 100K-1M updates per second, while for HDD storage we expect 1K-10K of updates per second. For SSD storage we expect your system to be on the order of 1K-5K reads per second, while for HDD storage we expect 20-100 reads per second).