KVZone and the Search for a Write-Optimized Key-Value Store

Salil Gokhale, Nitin Agrawal, Sean Noonan, Cristian Ungureanu
NEC Laboratories America, Princeton, NJ
{salil, nitin, sean, cristian}@nec-labs.com

Abstract

Key-value stores are becoming a popular choice for persistent data storage for a wide variety of applications, and multiple implementations are currently available. Deciding which one to use for a specific application requires comparing performance, a daunting task due to the lack of benchmarking tools for such purpose. We present KV-Zone*, a tool specifically designed to evaluate key-value store performance. We used KVZone to search for a key-value store suitable for implementing a low-latency content-addressable store that supports write-intensive workloads. We present a comparative evaluation of three popular key-value stores: Berkeley DB, Tokyo Cabinet, and SQLite, and find that none is capable of approaching the IO rate of our persistent device (a high-throughput SSD). Finally, we present the* Alphard *key-value store which is optimized for such workloads and devices.*

1 Introduction

Applications frequently need to store data persistently, a task traditionally achieved through filesystems or relational databases; in recent years they are increasingly being replaced by simpler storage systems that are easier to build at scale while maintaining reliability and availability. Examples of such systems are columnar stores (*e.g.*, BigTable [4]), document stores (*e.g.*, MongoDB [10]), and most popularly, key-value stores (*e.g.*, SimpleDB [12], Berkeley DB [3], Tokyo Cabinet [14], SQLite [13]). The simplicity and scalability of key-value stores has made them the infrastructure of choice for a wide range of usage scenarios. Not surprisingly, a large number of key-value stores have been developed, both commercial and open-source, each with distinctive performance, availability, and reliability characteristics [11].

A system designer considering the use of a key-value store for persistent data storage needs to be able to accurately compare the performance of different key-value stores for the same set of relevant workloads. Given the large number of key-value stores to choose from, this can be a daunting task. While subjective comparisons and critiques can be found on several websites and bulletin boards [5, 9], we are interested in quantitative performance differences under controlled workloads.

Recently, we developed HydraFS [15], a file system for HYDRAstor [6] content-addressable storage system. Originally designed for streaming applications, such as backup, we are currently working on improving its performance for metadata intensive applications through the addition of a low-latency content-addressable cache, built using a key-value store on top of a high-performance SSD. Our plan was to select an open-source key-value store that performs the best for our target workload. Since the key-value store was used for a local cache, we limited our search to the evaluation of *local key-value stores*, which considerably reduced the number of choices. However, these results are useful even for comparing distributed key-value stores since many of them are implemented as a distributed layer on top of a local key-value store (*e.g.*, Voldemort uses Berkeley DB, ErlangDB uses Mnesia).

Comparing the performance of even the smaller set of local key-value stores was harder than we thought. First, none of the existing storage benchmarks are designed to generate load in terms of key-value pairs, or easy to modify for that. Second, each key-value store we wished to test had a different interface. We believe benchmarks for key-value stores have to be adept at handling the interface diversity. For example, the three key-value stores we considered have different interfaces.

In order to perform an accurate comparison of the different key-value stores, we have taken the first steps in building a new benchmarking tool called KVZone, analogous to the IOZone benchmark for file systems [8]. KVZone is useful for conducting a broad analysis of key-value stores, having multiple configuration options to specify properties of keys, values, and operations on them; it solves the problem of different interfaces by providing a set of adapter modules that present the same interface to the load generator.

The current implementation of KVZone suffices for our needs in comparing key-value stores under write-intensive workloads. Other requirements, such as handling distributed key-value stores and various aspects of locality (*e.g.*, for key-value stores supporting sequential scan of the key-space), are likely to be important for the increasingly varied use of key-value stores. In this paper we seek to draw the attention of the storage community to some of the aspects that we consider important in benchmarking key-value stores, and point out the need for a more methodical approach to understanding the requirements for key-value store benchmarking in general.

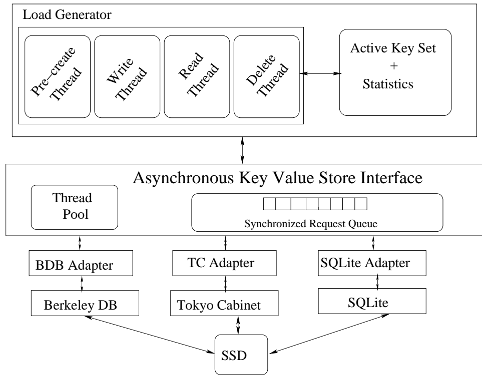

Figure 1: Overview of KVZone.

Key-Value Store	Throughput (MB/s)	Latency (ms) Avg.	Latency (ms) Stdev
SQLite	17.91	817.8	478.7
Berkeley DB	130.14	87.8	51.5
Tokyo Cabinet	178.32	38.4	36.1
Alphard	486.80	0.8	0.5

Table 1: Comparison of existing key-value stores write performance. *Throughput of raw device is approximately 500 MB/s.*

Using KVZone we compared three popular local key-value stores: Berkeley DB, Tokyo Cabinet, and SQLite. Tokyo Cabinet performed by far the best, followed by Berkeley DB; SQLite performed an order of magnitude worse in most tests.

However, even Tokyo Cabinet was not capable of approaching the IO rate of our SSD. To close the performance gap with the SSD, we built Alphard. The design goal was to execute each key-value store operation with at most one device IO. This was achieved by: employing an asynchronous interface, where requests are first placed in a queue and serviced by multiple threads; eliminating the need for metadata IO by keeping in memory all important metadata and using a layout which allows storing the modified metadata in the same IO with that of the associated data.

Our evaluation shows that for write intensive workloads Alphard can achieve performance comparable with that of the underlying device.

2 KVZone Performance Analysis Tool

2.1 Modes of Operation

Figure 1 shows the overall design and important data structures of KVZone; it generates a workload consisting of lookups, inserts and deletes in configurable proportions, where the sizes of keys and values are drawn from specified distributions. The system currently supports fixed size and uniform distributions; we plan to add support for realistic distributions of key-value sizes, similar to generation of file-system images by Impressions [1].

The rate of requests issued by the benchmark can be specified as either a *throughput* or *latency* target. In the former case, the tool determines the number of inserts per unit of time by taking into account the sizes of the key-value pairs. The tool continues until the *total* number of bytes requested are inserted, looked-up, or deleted. In the latter, a lower (or higher) latency is achieved by decreasing (respectively increasing) the number of operations generated in the next unit of time; there is an option to stop the execution when this number falls below a specified limit, signifying that the target is un-achievable.

One can also specify the *lifetime* of keys, the duration between a key's insertion and deletion; it can be either fixed, or be drawn from a uniform distribution. Note that in order to avoid the store becoming full or empty during a long run, the rate of deletions must equal that of insertions.

Finally, one can measure the performance of lookups or deletes alone by pre-populating the key-value store with a number of key-value pairs, specified prior to the benchmark run.

2.2 Design and Implementation

KVZone consists of three main components: a load generator, an asynchronous interface, and a set of adapter modules allowing the use of different key-value stores; we now describe each of these in more detail.

The load generator has a set of threads, one for each operation type (lookup, insert, delete), and one for pre-populating the store with key-value pairs prior to benchmarking. The insert thread obtains the details of a request (*i.e.*, the key and the size of the value) from a list of keys and a list of values, pre-generated to match the distributions specified by the user. On insert completion, the key is added to an *active key set*, which helps the lookup and delete threads to generate valid requests. Each thread alternates between submitting requests and sleeping, based on the rate requested for that operation by the user.

In order to support key value stores, such as Alphard, that have an asynchronous interface the KVZone interface needed to be asynchronous as well; each operation (insert, lookup, delete) specifies a continuation that is executed on operation's completion. The asynchronous interface consists of a set of queues holding requests submitted by the load generator, and a set of worker threads that issue synchronous calls to the appropriate adapter, passing through one of the queues; this interface allows a key-value store implementation to process more than one request with just one IO operation. On completion, the same thread invokes the continuations that were passed by the load generator.

In designing KVZone, we wanted to be able to run the exact same workload on different key-value stores to enable a fair performance comparison. Since different key-value stores can have different interfaces, we needed to translate the generic request from the load generator to one that is understood by the implementation-specific interface of a particular key-value store; an adapter module

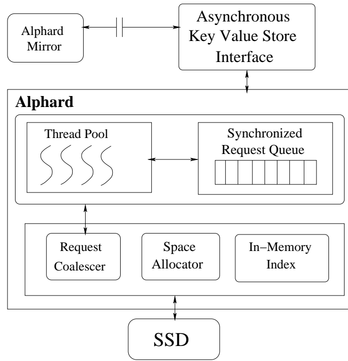

Figure 2: Overview of Alphard.

(one per key-value store) does precisely that.

For key-value stores that have a synchronous interface, the adapter module performs a *dequeue* operation to obtain the request to be issued. The adapters maintain performance counters necessary to measure the native latency of a key-value store (without queuing delays). The callbacks passed by the load generator are responsible for recording operation latencies that include queuing delays.

In the future, KVZone can be improved along several dimensions including configurability and scalability, along with a set of canned configurations representing some canonical workloads. We intend to further investigate key-value store workloads in our production environment and by running real applications; one possibility is to run a database query engine and observe the corresponding set of key-value operations.

3 The Alphard Key-Value Store

Using KVZone to compare three popular key-value stores we observed that none achieves a throughput close to that of the underlying device, as summarized in Table 1. To close the large gap between the device and the highest-performing key-value store for our workload (more than 60%!), we implemented Alphard. Our discussion of Alphard is quite terse, emphasizing its impact on the requirements for KVZone and the benefits of the comparative evaluation on Alphard's design.

3.1 Design Goals

The requirements for Hydra's key-value store reflect the needs of typical high-performance primary storage: low latency, high throughput, scalable concurrent performance, and no single point of failure. The key-value store contains optimizations, such as using direct I/O, that boost its performance when used with flash-based media. (FusionIO recommends using O_DIRECT for all I/O to their devices [7].)

3.2 Design and Implementation

Figure 2 shows the design of Alphard and its components which we discuss next; mirroring is relatively straightforward and we omit its discussion due to lack of space.

3.2.1 Space Allocation and On-disk Layout

Alphard contains an allocator to directly manage space, and does not require a file system to access a block device; having a dedicated allocator was useful for two reasons. First, usage of O_DIRECT requires that in-memory buffers, disk addresses, and request sizes be aligned to the device block size. Second, Alphard batches inserts, and this requires special handling during allocation.

The allocator supports three operations: *alloc*, *allocAdjacent*, and *free*. It maintains a bitmap (each bit corresponds to a block), as well as a short list of recently freed blocks (so as to speed-up allocations when there is significant churn but little free space). *allocAdjacent* allocates space on the device adjacent to a given address; it is used to coalesce multiple inserts so that they can be made persistent with just one device IO.

To reduce the total number of device IOs, each key-value pair is written together with metadata consisting of: key and value lengths, a virtual timestamp, and a checksum. The timestamp is necessary to allow the insertion of an already existing key (but with a new value) to be done with just one IO: if the size of the serialized entry is larger, we allocate new space for it and mark the old space as free, and store persistently the new entry without modifying the old one; during startup/recovery, if a key is part of more than one entry, only the latest is retained. This makes recovery slower, as all device blocks have to be read; for us this was an acceptable trade-off, as even a large SSD can be read fairly quickly. Note that the metadata overhead, together with the block alignment requirements, result in non-negligible overhead for key-value pairs of small size. Again, we considered this space/performance trade-off acceptable since the majority of our writes are large.

3.2.2 In-memory Index

Since key-value pairs have temporal locality (as opposed to spatial locality based on the key values) we need to maintain an index in memory. Alphard uses an efficient in-memory index storing the following information: 8 bytes storing either a key smaller than 7 bytes and its length, or a pointer to a larger key; on-disk start address (4 bytes) and length (4 bytes), in alignment-sized blocks. Since for larger keys the index contains only a pointer to the key, extra memory is needed in these cases, adding to the per-entry overhead of 16 bytes. The in-memory index uses cuckoo hashing with 32 keys per bucket.

The above scheme was sufficient for our needs, as our application did not generate keys larger than 7 bytes; the tests presented in this paper used 32-bit integer keys. In the future we plan to investigate data structures that better accommodate a high number of large keys.

3.2.3 Asynchronous Queue Interface

Alphard exports a (non-blocking) asynchronous interface; all key-value store operations (*i.e.*, *insert*, *lookup*, and *delete*) are first placed in a logically centralized queue. Upon finishing any previous operation, a worker thread removes from the queue one or more operations that will be executed with one synchronous IO to the block device. Multiple worker threads service the queue, for which reason the queue is synchronized; in order to mitigate lock contention, multiple physical queues are used, with requests mapped to queues based on the key specified in the request. After executing the IO, the thread invokes the callbacks specified by the application during submission of those requests.

- **Insert:** The key and value of an insert request are serialized into the thread's pre-allocated buffer, followed by any padding necessary due to alignment requirements for each key-value pair, followed by the appropriate metadata. Space on the device is allocated for this entry. If other insertion operations exist in the queue, they are processed similarly, as long as there is space in the buffer and device allocations can be made contiguously to those of previous entries. The buffer containing one or more key-value pairs is written to the device with a single I/O; upon success, the key, device location, and entry length are recorded in the in-memory index.
- **Lookup:** First the location and value size of the entry on the device are retrieved from the in-memory index. Next, data is read from the device and checked for validity; the validated value is then returned.
- **Delete:** Alphard simply overwrites the first block belonging to the key-value entry on the device with zeroes, removes the associated entry from the in-memory index, and marks the device blocks as free. The durability of the delete is guaranteed since the on-disk entry now lacks a valid header.

4 Evaluation

4.1 Summary of Existing Key-Value Stores

We first present a summary of our findings for the performance evaluation of three popular local key-value stores: Berkeley DB, Tokyo Cabinet, and SQLite. Although a large number of distributed and local key-value stores are available, for use in HYDRAstor we were interested in the performance of the latter; we do not evaluate distributed key-value stores such as Voldemort, Cassandra, and memcached.

Berkeley DB: is highly customizable and supports fine-grained locking, although as we found, its transactional data store is prone to deadlocks for write-intensive workloads; we instead used its concurrent data store. In our evaluation we observed that Berkeley DB provides poor throughput for write-intensive workloads and exhibits a high latency for concurrent operations.

Tokyo Cabinet: a high-performance and scalable key-value store used by Mixi, a popular Japanese social networking site. Tokyo Cabinet performed by far the best of the three; however, its performance was not close to that of the underlying SSD.

SQLite: the least impressive in terms of performance, often performing an order of magnitude worse; the graphical results omit SQLite to retain readability.

4.2 Performance Evaluation of Alphard

Due to lack of space, we highlight with a few results Alphard's performance improvements, the key insight to which is the reduced number of device I/Os needed – at most one per operation.

4.2.1 Experimental Setup

We conduct all experiments on a FusionIO 80 GB SSD formatted to 50 GB; the remaining space provides a large enough pool of free blocks to sustain performance as recommended [7]. The test machines have 2 Intel Xeon X5450 Quad Core 3GHz CPUs, 8 GB of memory and run Linux 2.6.18. In the graphs, $Alphard_{raw}$ denotes the performance of Alphard over a raw device, and $Alphard_{fs}$ that over a file system. Since the choice of using raw device or filesystem may depend on factors other than performance, we include both numbers. We also ran similar experiments with mirroring to another Alphard instance and observed less than 5% drop in performance.

4.2.2 Write-only Performance

Figure 3 compares performance in the absence of reads; deletes are introduced at the same rate as writes to ensure steady progress. In the first graph, as the target load IOPS increase, Alphard's performance peaks at 8000 IOPS, Tokyo Cabinet's at 6500, while Berkeley DB's at 3800. The second graph shows average latency for write requests as the load IOPS increase. Alphard can achieve higher IOPS with a much lower latency then either Tokyo Cabinet or Berkeley DB. For example, the average write latency at 8000 IOPS for Alphard_{raw} is just under 10ms, while it is just over 10ms for Alphard_{fs}. For the same value of load IOPS, both Tokyo Cabinet (4000 ms) and Berkeley DB (48000 ms) suffer from a significantly high write latency. The third graph shows achieved IOPS as the size of the *value* blocks changes. At higher block sizes, achieved IOPS is limited by the maximum device throughput. For example at 256 KB block size, Alphard achieves a throughput of 450 MB/s, compared to 300 MB/s for Tokyo Cabinet, and 56 MB/s for Berkeley DB.

4.2.3 Read and Write Performance

Figure 4 compares performance when the load includes a mix of reads and writes. As the load IOPS increase, the achieved IOPS and write latency do not change significantly in case of Alphard or Berkeley DB. Performance of

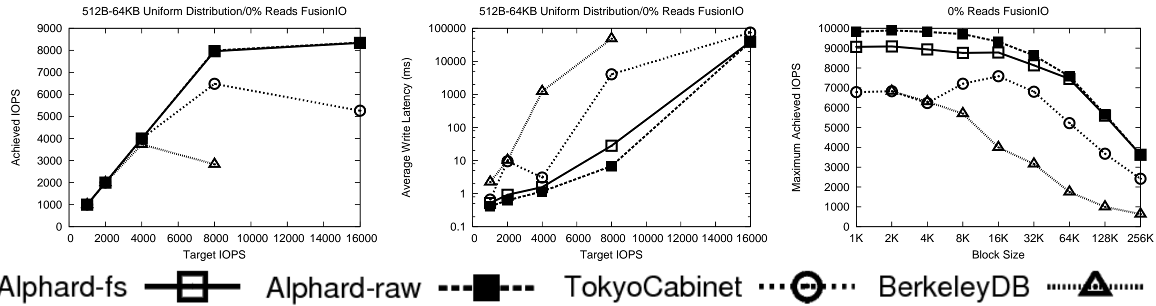

Figure 3: Write Performance . *Results for workloads consisting 50% writes and 50% deletes. The first two graphs show the achieved IOPS and write latency as the target IOPS is varied. The third graph shows achieved IOPS for various block sizes.*

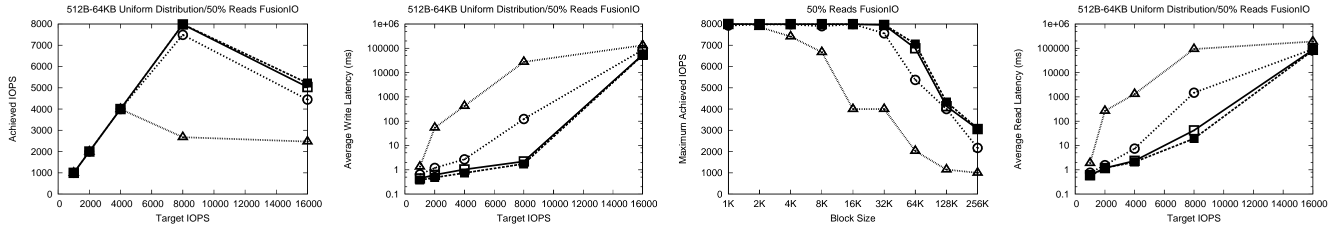

Figure 4: Reads+Writes Performance . *Results for workloads consisting 50% reads, 25% writes and 25% deletes. First three graphs are as above; the fourth plots the observed read latency for various target IOPS.*

Tokyo Cabinet does improve with an increase in the number of read requests in the mix. Even when read requests are added to the mix, Alphard outperforms Berkeley DB and Tokyo Cabinet in both throughput and latency.

5 Related Work

The number of existing key-value stores is far too many to have a comprehensive comparison in this paper; we instead direct the reader to a recent survey of popular key-value stores [11]. FAWN-KV [2] comes closest to Alphard in terms of being specifically designed for use on SSDs; they both have in-memory data structures to speed lookups and reduce the number of I/Os, but are designed for very different environments. FAWN-KV is designed for good read performance on "wimpy" nodes having a few Gigabytes of flash satisfying an energy consumption budget; in contrast, Alphard is designed for write performance on "beefy" nodes having at least an order of magnitude more flash. FAWN-KV's evaluation focuses on relatively small objects (< 1KB) as compared to Alphard that expects larger objects on average.

6 Conclusions

In spite of the popularity of key-value stores, benchmarking tools to evaluate them lag behind. We developed KVZone specifically for key-value store evaluation, born out of our own experiences in identifying a suitable key-value store for write-intensive workloads on an SSD; we have released it for public use at `http://www.nec-labs.com/research/robust/robust_grid-website/software.php`. Using KVZone we were able to quantitatively assess the strengths and weaknesses of popular local key-value stores such as Berkeley DB, Tokyo Cabinet, and SQLite, and discovered they achieve less than half of the expected performance. We thus developed Alphard to close this gap, particularly well-suited for write-intensive workloads on SSDs. Our evaluation shows that it beats the competition comfortably for all relevant workloads, providing a useful addition to the repertoire of local key-value stores. We believe more systemic effort is needed to build better tools to support the evaluation and design of key-value stores in the future.

References

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *FAST '09*, San Francisco, California, February 2009.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. In *SOSP '09*, pages 1–14, New York, NY, USA, 2009. ACM.

[3] Berkeley DB database. `http://bit.ly/6PM33q`.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS '08*.

[5] B. Chapman. Evaluating key-value and document stores for short read data. `http://bit.ly/14wX7T`.

[6] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: A Scalable Secondary Storage. In *FAST 2009*.

[7] FusionIO Recommends O_DIRECT. Personal Communication.

[8] IOzone Filesystem Benchmark. `http://bit.ly/7dwE8W`.

[9] R. Jones. Anti-RDBMS: A list of distributed key-value stores. `http://bit.ly/JXtx`.

[10] MongoDB. `http://bit.ly/8sg86h`.

[11] M. Seeger. Key-value stores: a practical overview. `http://bit.ly/6iASTz`, 2009.

[12] Amazon SimpleDB. `http://bit.ly/4Etslc`.

[13] SQLite SQL database engine. `http://bit.ly/68lwas`.

[14] Tokyo cabinet. `http://bit.ly/5WwhUI`.

[15] C. Ungureanu, A. Aranya, S. Gokhale, S. Rago, B. Atkin, A. Bohra, C. Dubnicki, and G. Calkowski. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST 2010*, San Jose, California, Feb. 2010.