ZDB-High performance key-value store

Thanh Nguyen Trung, Minh Nguyen Hieu
Information Technology Faculty
Le Quy Don Technical University
Ha Noi, Viet Nam
thanhnt@vng.com.vn, hieuminhmta@ymail.com

***Abstract*—Nowadays key-value stores play a critical part in large-scale high performance applications. Attention paid to key-value stores prove the importance of the key-value store that have already been used. This paper presents ZDB which is a high performance persistent key-value store designed for optimizing reading and writing operations. This key-value store support sequential write and single disk seek for read and write operations. Key contributions of this paper are the principles in architecture, design and implementation of a high performance persistent key-value store. This is achieved by using a data file structure organized as commit log storage where every new data is appended to the end of the data file. An in memory index is used for random reading. ZDB architecture optimize the index of key-value store for auto incremental integer keys which can be applied in storing many real life data efficiently with minimize memory overhead and reduce the complexity for partitioning data.**

***Keywords*-key-value; nosql; storage; zdb**

I. Introduction

High-performance key-value stores have been given large attention in several domains, equally in professional and academics. E-commerce related platforms [12], data deduplication [17], [11], [10], photo merchants [9], web object caching [6], [8], [13] etc. Attention paid to key-value stores prove the importance of the key-value store that have already been used. Before this research, we had used some famous key/value storage libraries using B-tree and on-disk hash table for building persistent cache storage system for applications. When the number of item in database increase and the data of the application grow to millions of items, the libraries we used worked more slowly for both reading and writing operations. It is therefore important to implement a simple and high performance persistent key-value store which can perform better than the existing key-value stores both in memory consumption and in speed.

Some famous key-value storage such as Berkeley DB [5] (BDB) used B-tree structure or hash table often store the index in a file on the disk. For each database writing operation, it needs at least two disk seeking [23], [16], the first seeking for updating B-tree or hash table, and the second for updating data. In case of re-structured B-tree, it needs more disk seek in reading/writing operations. Consequently data growth means writing rate increases thus making B-tree storage slower.

With popular commodity hard disk and SSD nowadays, sequential disk writing has the best performance [6], [16] so the strategy for the new key-value store is to support sequential data writing, support random writing, and minimize seek operations.

To use all capacity of limited IO resources, achieve high-performance and low latency, key-value storage must minimize number of disk seeking in every operation and all writing operations should be sequential or append only on disk. This research presents algorithms that implement efficient storage of key-value data on drive. They will minimize the required number of disk seeking. The algorithm applications are quite general, hence applicable many other applications as well. This research is done to optimize disk reading/writing operation in data services of applications.

Understanding the specification of data types especially the type of key in key-value pair is important to design the scalable store system for that data. There are several popular key types: variable-length string, fixed size binary, random integers, auto incremental integer... In popular applications, incremental integer keys are used widely in database design. For example: the identification of Users, Feeds, Documents, Commercial Transactions... So optimizing the key-value store for auto incremental integer keys is very meaningful.

This research firstly optimizes memory consumption of index of key-value store for auto incremental integer keys. It also reduces the complexity of partitioning data. This research also extends the work for supporting variable length string keys in simple way.

These are main contributions of this paper:

- The design and implementation of flat index and random readable log storage that make high performance, low latency key-value store
- Minimize memory usage of the index and optimize for auto incremental integer keys and make the zero false positive rate of flash/disk reads key-value store.
- Remove some disadvantage of previous research in design and implementation of key-value store.

II. ZDB Key-Value Storage System

ZDB is designed for optimizing reading and writing operations. It needs at most one disk seek for the operations.

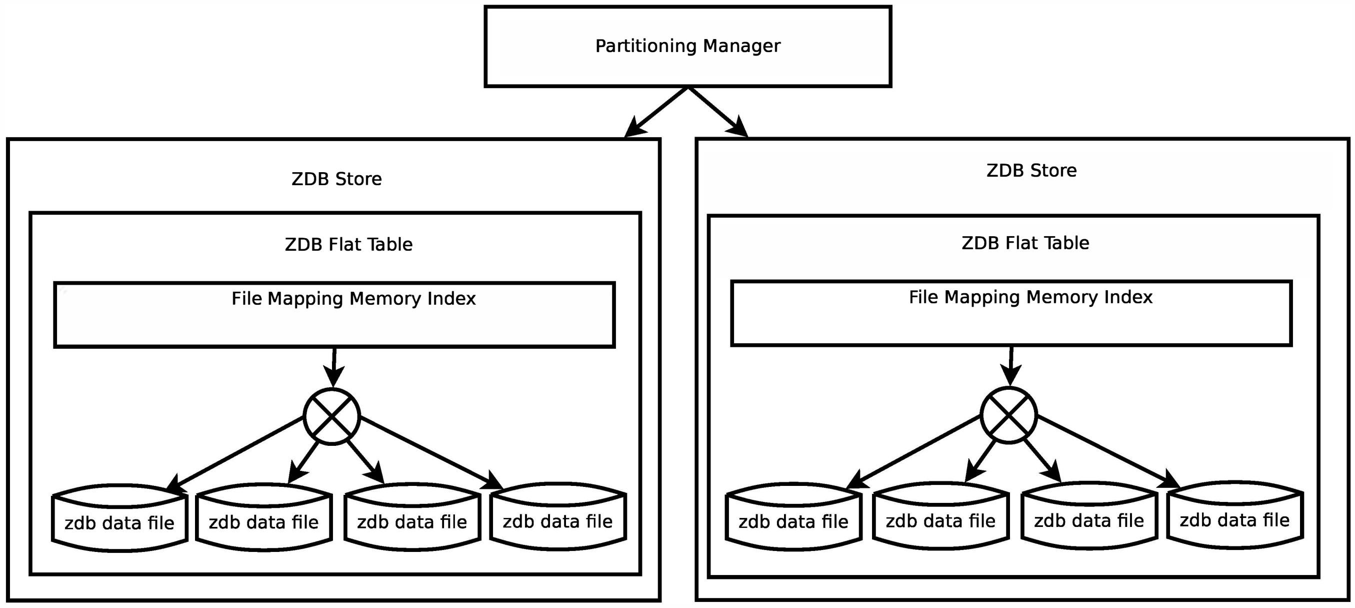

Figure 1. ZDB architecture

In ZDB, all writing must be sequential. Consequently, the data file structure is organized as commit log storage every new data is appended to the end of the data file. For random reading, an in-memory index is used to locate value position of a key in commit log storage. Commit log and the in-memory index is managed by ZDB Flat Table while the ZDB Flat Table is managed by ZDB Store. Hash function is used in calculating the appropriate file to store the key-value pair. Figure 1 shows the basic structure of ZDB architecture.

A. ZDB Flat Index

The index is used to locate position of key-value pair in data file. Dictionary data structure [20] such as tree, hash table can be used for storing index. But for auto incremental integer keys dictionary data structure is not optimal in memory consumption and performance.

With integer keys, there are advantages for using linear arrays over the use of trees or hash tables. The difference between a hash table and an array is that accessing an element in a plain array only requires finding an index of a particular element while hash tables using a hash function to generate an index for a particular key, then use the index to access the bucket that contain key and value in the hash table. In the structure of hash table both key and value are stored in memory. For integer keys, we can use key as the index of item in linear array and we can get item from key very simple without storing keys.

For an individual element, a hash table has an insertion time of O(1) and a look-up time of O(1) [20]. This is assuming that the hashing algorithm can work perfectly and collisions are managed properly. On the other hand, the access time of an array is O (1) for a given element. Arrays are very simple to use. In addition, there is no overhead in generating an index. Moreover, there is no need for collision detecting. ZDB uses append-only mode, the data is written to the end of a file and the indices is already predetermined, the array is used for storing position of key-value entry in the data file. To keep the array index persistent, file mapping is used.

ZDB optimize the index for auto incremental integer keys, and use array to store this index for minimize memory usage which have zero overhead for keys. ZDB Flat Index is an array of entry position.

1) ZDB Flat Index parameters: For each partition in ZDB, the index parameters describe characteristics such as the size of the array, the range of the array and the memory consumption ranges.

- Key range

Key range in a partition is called $[k_{min}, k_{max})$ where k_{min} is the start of the index while $k_{max} - 1$ is the last index in the array. The range is inclusive of the boundary value.

- Index Array Size

The size of the array is obtained from the range as this equation:

$$ArraySize = k_{max} - k_{min} \quad (1)$$

Basing on the values of the range, the i^{th} item in the array refers to the position of the key $(i + k_{min})$ in the data file. It is also imperative to note that the size of an item depends on the maximum file size. In ZDB, this may be 4, 5, 6, 7, or 8 bytes for easy configuration and for tuning the performance and maximum data file size of the key-value persistent store. Comparing ZDB and FAWN, the size of an item can only be 4 making it to be rigid not to provide options to tune the performance of the key-value store. In ZDB, data in a partition is stored in multiple files using a simple hash function to decide which file to store the key. The hash function must be efficient for better performance of the key-value store. The choice of the key and the basics of the key-value store are described in the sections below.

- Index Memory Consumption

In ZDB, the memory consumption is equal to the size of the array multiplied by the size of the array item. As aforementioned, memory is only used to store the position of the entry and not the key.

2) ZDB Flat Index example: In social networks such as Facebook [1] and Flickr [2] , and in email hosting websites such as GMail [14], the key may refer to the User ID while the value is the profile which is serialized to binary or string. The story is not different with Zing Me [22] because login information requires a User name and password before the user profile is displayed. By knowing the User ID which is the key, the profile of the user can be retrieved from ZDB. It should be understood that ZDB uses a predefined a range of keys for example $[0, 1000000)$ in a partition. The size of the array is 1000000. If the number of data files is 16, the data with key k would be stored in k modulus 16. Using 4 bytes for each index item in the index array, the maximum file size would be 4 GB and the total size would be 64GB for all the files. Since the index size is 1000000, the memory size for the index is 4*1000000 bytes (about 4MB). In one partition, the size of the index table can be several hundreds.

B. ZDB Log Storage

Key-value pairs are stored in ZDB data file sequentially in every writing operation. For each writing, the following data are appended to data file: Entry Information (EI), Value, Key.

Entry Information consists of: Value Size: 4bytes, Reserved Size: 4 bytes, Time stamp: 8bytes, Value check sum: 1 byte. The layout of ZDB Log Storage files are describe in Figure 2

C. ZDB Flat Table

The ZDB Flat Table consists of a ZDB Flat Index and multiple ZDB Log storage data files. The ZDB Flat Index is used for looking up the position of key-value pair in ZDB Log Storage data file. ZDB Flat table have some interfacing commands to interact with the data store include get, put, and remove. ZDB Flat Table also has 2 iterating command: Key-order iterating and insertion order iterating. With iterating commands, it is able to can through the table to get all key-value pair.

- Put

Put is used for add or update key-value pair to the table. This means that the value which is the data and the reference which is the key should be stored in the data files and the index array respectively. Consequently, the input for the

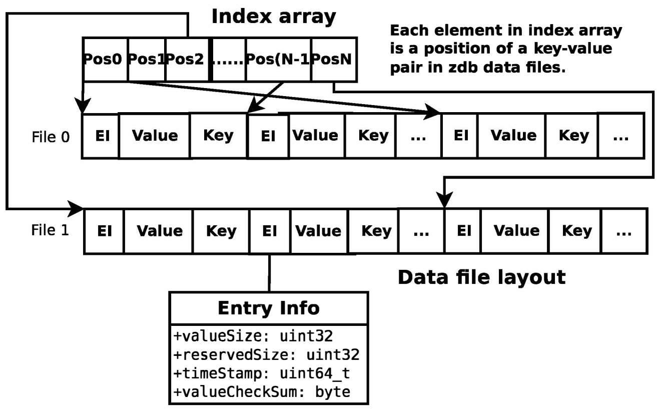

Figure 2. Data file layout

put command is the key and the value both provided. The data file to store the entry is determined by hash function. The current size of the data file is obtained and set to the $(key - k_{min})^{th}$ item in the index array. The entry is then appended to the end of the data file.

- Get

To get a value referenced in the ZDB Flat table by the index, the input to the get command is the key while the output is the value. The file that stores the value is determined by hash function. The position of the entry is looked up in the index array $(key - k_{min})^{th}$ item. The existence of the entry is determined by whether the position is greater than 0. If the position is greater than 0, the position of the file is sought in the array and the entry is read to produce the output which is the value. Get operation of ZDB has zero false positive disk read.

- Remove

The remove command is meant to eliminate the entry from both the array index and the data file. The input required to remove an entry is only the key. With the key, the hash function is used to calculate the data file holding the entry. The item is set to -1 in the index array. At the same time, an entry info that indicates the pair with the key was removed is created and append to data file. Entry Information for indicate removed key:

Value Size: 0, Reserved Size: 0, Time stamp:0, Value Check Sum: 0

- Iterate

Other important actions in the key-value store include sequence iterating which is done by scanning each ZDB Flat Table to iterate all the key-value pairs. A hash order or insertion order can be used to iterate through all the key-value pairs.

For key-order iterating, ZDB Flat Index array are scanned, if the item in array are greater than or equal to 0, the key associated with that item has the value in the ZDB Log

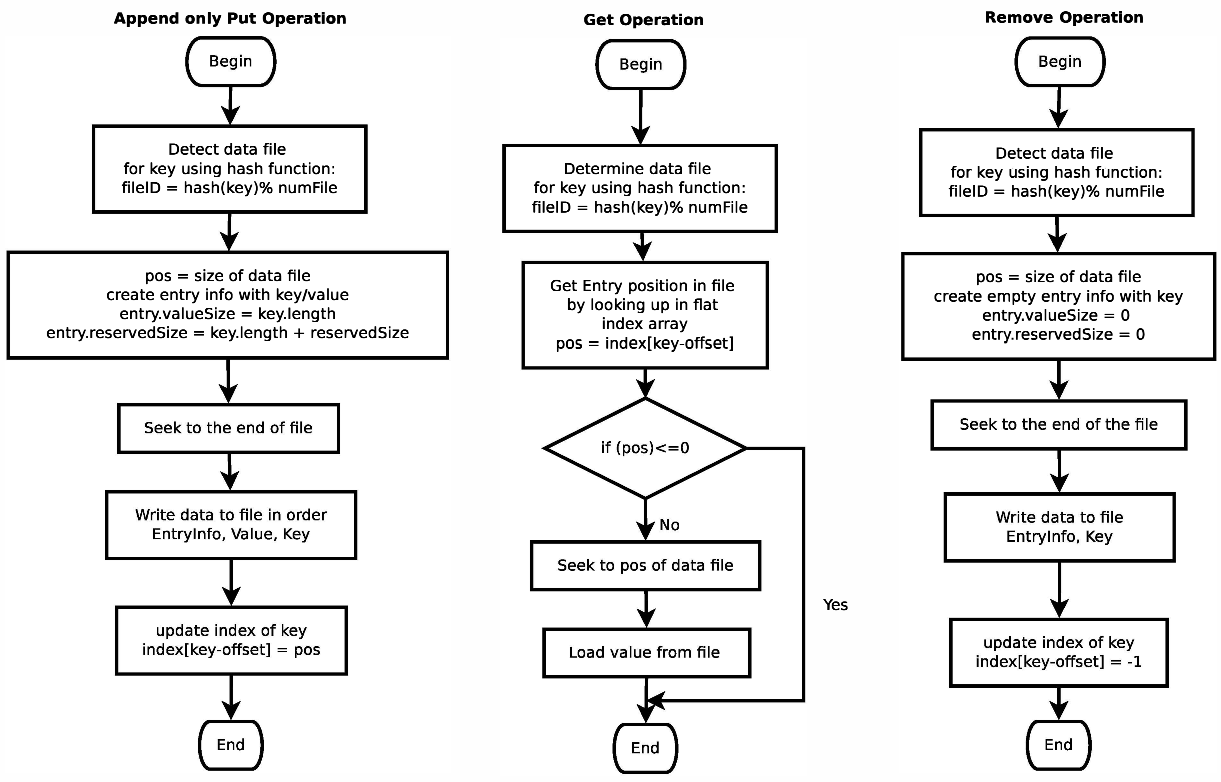
Append only Put Operation
Begin
Detect data file for key using hash function: fileID = hash(key)% numFile
pos = size of data file
create entry info with key/value
entry.valueSize = key.length
entry.reservedSize = key.length + reservedSize
Seek to the end of file
Write data to file in order EntryInfo, Value, Key
update index of key index[key-offset] = pos
End
Get Operation
Begin
Determine data file for key using hash function: fileID = hash(key)% numFile
Get Entry position in file by looking up in flat index array pos = index[key-offset]
if (pos)<=0
No
Yes
Seek to pos of data file
Load value from file
End
Remove Operation
Begin
Detect data file for key using hash function: fileID = hash(key)% numFile
pos = size of data file
create empty entry info with key
entry.valueSize = 0
entry.reservedSize = 0
Seek to the end of the file
Write data to file EntryInfo, Key
update index of key index[key-offset] = -1
End

Figure 3. Put, Get, Remove algorithms of ZDB Flat Table

Storage, and the value are read for returning to the iterating operation.

For insertion order, each ZDB Log Storage data file are scan and read each Entry Information and key-value pair sequentially. For each read key-value pair, if its position in ZDB Log Storage data file equal to the position value associated with the key in ZDB Flat index then it is a valid key-value pair, so return it to the iterating operation.

D. ZDB Store

ZDB store uses ZDB Flat Tables functionality and handle all data store request from applications. ZDB Store use thrift protocol [19] to serve request from clients. ZDB Store also provides compact operation for release disk usage by multiple writing to a key. ZDB use chain replication [21] for replicating data in cluster. Every writing operation work on all nodes in the cluster asynchronously. ZDB use Eventually consistent model from [12].

E. Variable Length String Keys

Currently, ZDB Flat index works as an in-memory for storing position of key-value entry in data files. It has been tested to work more efficiently with auto incremental integer keys. However, it is not difficult to implement variable length string keys into the key-store. For instance the key can be indicated as a string key (skey) to differentiate it from integer keys (iKey). A list of the string keys can be stored in a bucket. It is imperative to note that string keys in a bucket, they must have the same hash value. For storage, an iKey and bucket pair is stored in ZDB as integer key and value pair. All changes to the record of skeys are effected to the bucket for updating the ZDB store. Each Flat Table is setup with a size of about 2^{27} for the string keys and Jenkins hash function used to hash skey. The best ZDB performance is obtained when the number of keys is estimated to the size of ZDB Flat Index. The implementation basics can be summarized as shown below:

- $skey : string, iKey = hash(skey)$
- $value : string$
- pair consist of skey and value : $\{skey, value\}$
- bucket: list of pair, all string keys in this list have the same hash value.

We cache and store $\{iKey, bucket\}$ in ZDB.

III. Compare To Other Key-Value Store

There are many key-value stores but each is based on its concept. The first that can be compared to ZDB is SILT (Small Index Large Table) [15]. SILT is a memory efficient, high performance key-value store based on flash storage. It scales to serve billions of key-value items on a single node. Like most other key-value stores, SILT implements simple exact-match hash table interface including PUT, GET, and DELETE. ZDB implements all the three although the Delete interface command is renamed to remove but it performs the same functionality. Unlike ZDB, SILTs multi-store design uses a series of basic key-value stores optimized for different purposes. However, the basic design of SILTs LogStore works like ZDB. This is because the LogStore uses a new hash table to map keys to candidates. The main difference is that the LogStore uses two hash functions [18] to map the keys to the buckets and still have false positive disk access while the ZDB have no false positive disk access. It is also imperative to compare how the stores filled LogStore in the case of SILT and a ZDB in the case of ZDB. When a LogStore is full, it is converted into a HashStore in order to handle the data and a new LogStore is created to handle the new operations. In the case of a ZDB, the ZDB Flat Table just care about the range of its key, for keys out of range, just simply create new partition associate to the new key range. ZDB can support large data file, and the maximum size of data file is configurable, with SILT LogStore the maximum size of data file is always 4G (because it used 4 bytes offset pointer in the index). The value size and key size of SILT are fixed; the value size of ZDB is variable.

In addition, there are situations where SILT has been used in high writing rate applications. Challenges facing SILT include difficulty in controlling the number of HashStore because Each LogStore contains only 128k items. Basing on the SILT paper, complexity on LogStore to HashStore conversion is unclear. The paper does not mention the complexity of memory consumption in the event of converting or merging. The complexity of the effect of converting to running SILT node is also not clear. As depicted in the SILT paper, it is good at fixed-size key value with large and variable length values. This is also the case with ZDB which has high performance with large value sizes. The difference comes in the complexity of SILT and ZDB SILT is difficult to organize and is more complex whereas ZDB is simple and easy to organize.

Fawn Data Store (FAWN DS) [7] is a log-structured key-value store. In FAWN DS, each store contains values for the key range associated with one virtual ID. It also supports interfacing such as Store, Lookup, and Delete. This is based on flash storage and operates within a constrained DRAM available on wimpy nodes. This means that all writes to the data store are sequential and all reads require a single random access. Unlike ZDB which uses an array index to store keys, the FAWN Data Store uses a hash index to map 160 bits keys to the actual key stored in memory to find a location in the log. It then reads the full key from the log and verifies the correctness of the key. ZDB is designed to minimize reads from the memory to improve performance. In that case, ZDB only uses one seek write and append only mode for compacting.

While FAWN has a fixed memory index, ZDB index is variable and can be tuned to improve the performance of the key-value store in FAWN the maximum size of data file is always 4G. Another difference between ZDB and FAWN lies in the hashing of original key in FAWN by SHA. It cannot be iterated to determine the original key. On the other hand, the original key in ZDB is not hashed and it can therefore be iterated to find the original key. It is imperative to note that with ZDB, there is no incorrect flash/hdd retrieval.

Table I
ONE WRITING THREAD

DBType	Cases		
	Key:4bytes Value:4 bytes	Key: 4bytes Value:1KB	Key: 4bytes Value:100KB
LevelDB	347246	5360	61
KC	343348	10268	1872
ZDB	294796	108790	4132

Table II
FOUR WRITING THREAD

DBType	Cases		
	Key:4bytes Value:4 bytes	Key: 4bytes Value:1KB	Key: 4bytes Value:100KB
LevelDB	369760	15004	90
KC	241800	80420	1920
ZDB	537204	128220	5248

The performance of a key-value store comparatively is important especially if users have to choose among various available options.

The comparison in the performance of ZDB and two famous open source persistent key-value stores: LevelDB [4] and Kyoto Cabinet [3] using standard environment with:

Operating System: CentOS 64 bit , CPU: Xeon Quad core, Memory: 8G DDR , HDD: 600G connected via SATA and formatted with ext4 filesystem

These scenarios are used to evaluation:

- Writing 100 million key-value pair with variable value size in one thread.
- Writing 100 million key-value pair with variable value size in 4 threads.
- Random Reading key-value from stores

The benchmark results are shown on tables above, the number in the table show the number of operations per second. ZDB has the highest number of operations per second in most scenarios. It is imperative to note that keys of 4 bytes and values of 100 Kilobytes have the lowest number of operations because of the size of the values.

In the first instance, the key-value store engines are setup with 1 writing thread with keys of 4 bytes and value of 4 bytes, keys of 4 bytes and values of 1024 bytes, and keys of 4 bytes and values of 100KB. The results in TABLE I above show that ZDB has the highest number of operations per second and would take a shorter time writing the key-value pairs in all the parameters except for values of 4 bytes

Table III
RANDOM READING

DBType	Cases		
	Key:4bytes Value:4 bytes	Key: 4bytes Value:1KB	Key: 4bytes Value:100KB
LevelDB	304448	4629	62
KC	1176300	45234	5075
ZDB	1326205	60325	6232

The benchmark was repeated with four writing threads and the results are shown in TABLE II. It shows that ZDB work better in concurrent environment.

The benchmark was also set up for reading operation on the data and the results show that ZDB had a higher number of operations per second compared to Kyoto cabinet and LevelDB. These results are shown in TABLE III

IV. CONCLUSION

ZDB uses simple techniques to create a high performance persistent key-value store. To store a key-value pair in a file, the evenly distribution hash function is used in selecting the most appropriate file. Common interfacing commands such as Put, Get, and Remove are used in ZDB. It has a flexible item sizes to allow for tuning to enhance better performance. To reduce the number transfers to and from memory, file appending is used and one-seek write is used. It makes use of a ZDB Flat Index to map key to position of key-value pairs stored in data files. In all operation, ZDB needs at most one disk seek. In addition, all writing operations are sequential. For applications that require a simple high performance with optimized disk reading and writing operations, especially for large value, ZDB can be a good choice.

REFERENCES

[1] Facebook. http://facebook.com, 2013.

[2] Flickr. http://www.flickr.com, 2013.

[3] Kyoto cabinet: a straightforward implementation of dbm. http://fallabs.com/kyotocabinet, 2013.

[4] Leveldb - a fast and lightweight key/value database library by google. http://code.google.com/p/leveldb, 2013.

[5] Oracle berkeley db 12c: Persistent key value store. http://www.oracle.com/technetwork/products/berkeleydb, 2013.

[6] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large cams for high performance data-intensive networked systems. In *NSDI*, volume 10, pages 29–29, 2010.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[8] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. Hashcache: Cache storage for the next billion. In *NSDI*, volume 9, pages 123–136, 2009.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[10] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[11] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36. ACM, 2011.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[13] B. Fitzpatrick. A distributed memory object caching system. http://www.danga.com/memcached/. Accessed September 4, 2013.

[14] Google. Gmail. http://mail.google.com, 2013.

[15] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[16] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: Random write considered harmful in solid state drives. In *Proc. of the 10th USENIX Conf. on File and Storage Tech*, 2012.

[17] J. C. Mogul, Y.-M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in http. In *NSDI*, volume 4, pages 4–4, 2004.

[18] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[19] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5, 2007.

[20] T. van Dijk. Analysing and improving hash table performance. 2009.

[21] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[22] VNG. Zing me. http://me.zing.vn, 2013.

[23] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST*, volume 5, pages 3–3, 2005.