Zing Database: high-performance key-value store for large-scale storage service

Article · February 2014

DOI: 10.1007/s40595-014-0027-4

2 authors:

Thanh Nguyen
Le Quy Don Technical University
12 PUBLICATIONS **17** CITATIONS
SEE PROFILE

Minh Hieu Nguyen
ENSTA ParisTech
4 PUBLICATIONS **13** CITATIONS
SEE PROFILE

Vietnam J Comput Sci
DOI 10.1007/s40595-014-0027-4

REGULAR PAPER

Zing Database: high-performance key-value store for large-scale storage service

Thanh Trung Nguyen · Minh Hieu Nguyen

Received: 31 March 2014 / Accepted: 4 August 2014

Abstract Nowadays, key-value stores play an important role in cloud storage services and large-scale high-performance applications. This paper proposes a new approach in design key-value store and presents Zing Database (ZDB) which is a high-performance persistent key-value store designed for optimizing reading and writing operations. This key-value store supports sequential write, single disk seek random write and single disk seek for read 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 using a data file structure organized as commit log storage where every new data are appended to the end of the data file. An in-memory index is used for supporting random reading in the commit log. ZDB architecture optimizes 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 · Zing database · ZDB

1 Introduction

Key-value store is a type of nosql databases. It has simple interface with only one two-column table. Each record has two fields: *key* and *value*. The type of value is string/binary, the type of key can be integer or string/binary. There are many implementation and design of key-value store including in-memory based and disk persistent. In-memory based key-value store is often used for caching data, disk persistent key-value store is used for storing data permanently in file system.

High-performance key-value stores have been given large attention in several domains, equally in industry and academics. E-commerce related platforms [15], data de-duplication [13,14,23], photo merchants [10], web object caching [7,9,16] etc. Attention paid to key-value stores proves the importance of the key-value store that has already been used. Before this research, we had used some popular key/value storage libraries using B-tree and on-disk hash table for building persistent cache storage system for applications. When the number of items in database increases 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 popular key-value storages 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 [22,32], 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 [7,22] so the strategy for the new key-value store is to support sequential data writing, and minimize number of disk seeks in every

T. T. Nguyen (✉) · M. H. Nguyen
Information Technology Faculty, Le Quy Don University,
No 236 Hoang Quoc Viet Street, Hanoi, Vietnam
e-mail: trungthanhnt@gmail.com; thanhnt@vng.com.vn

T. T. Nguyen
R&D Department, VNG Corporation, Trung Kinh, Hanoi, Vietnam

Published online: 17 August 2014

operation. 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. This research is done to optimize disk reading/writing operation in data services of applications.

Understanding the characteristics 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 a 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.
- Find and remove some disadvantages in previous research in design and implementation key-value store such as SILT [20] and FAWN-DS [8]. disadvantages from FAWN-DS and SILT: hash keys using SHA and use hash values as string keys in key-value store. It is difficult to iterate the store.

2 Zing Database architecture

Zing Database (ZDB) is designed for optimizing both reading and writing operations. It needs at most one disk seek for the operations. In ZDB, all writings must be sequential. The data file structure is organized as commit log storage, every new data are 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.

2.1 Data index

The data index is used to locate position of key-value pair in data file. Dictionary data structure [29] 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.

For storing auto-incremental 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 contains 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 lookup time of O(1) [29]. 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 are written to the end of a file and the index is already predetermined, the array is used for storing position of key-value entry in the data file for random reading. To keep the array index persistent and low-access latency, fast recovery, File mapping is used. File mapping [28] is a shared memory technique supported by most modern operating systems or runtime environments. POSIX-compliant systems use *mmap*() function to create a mapping of a file given a file descriptor. Microsoft Windows uses *CreateFileMapping*() function for this purpose. File mapping is a segment of virtual memory assigned a direct byte-to-byte correlation with some portion of a file. The primary benefit of File mapping is increasing performance of I/O operations. Accessing file mapping is similar accessing to program's local memory and it is faster than using direct read and write operations because It reduces number of system calls.

ZDB optimizes the index for auto-incremental integer keys, and uses array to store this index for minimize memory usage which has zero overhead for keys. ZDB flat index is an array of entry positions. File mapping is used to access ZDB flat index.

2.1.1 Zing Database 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.

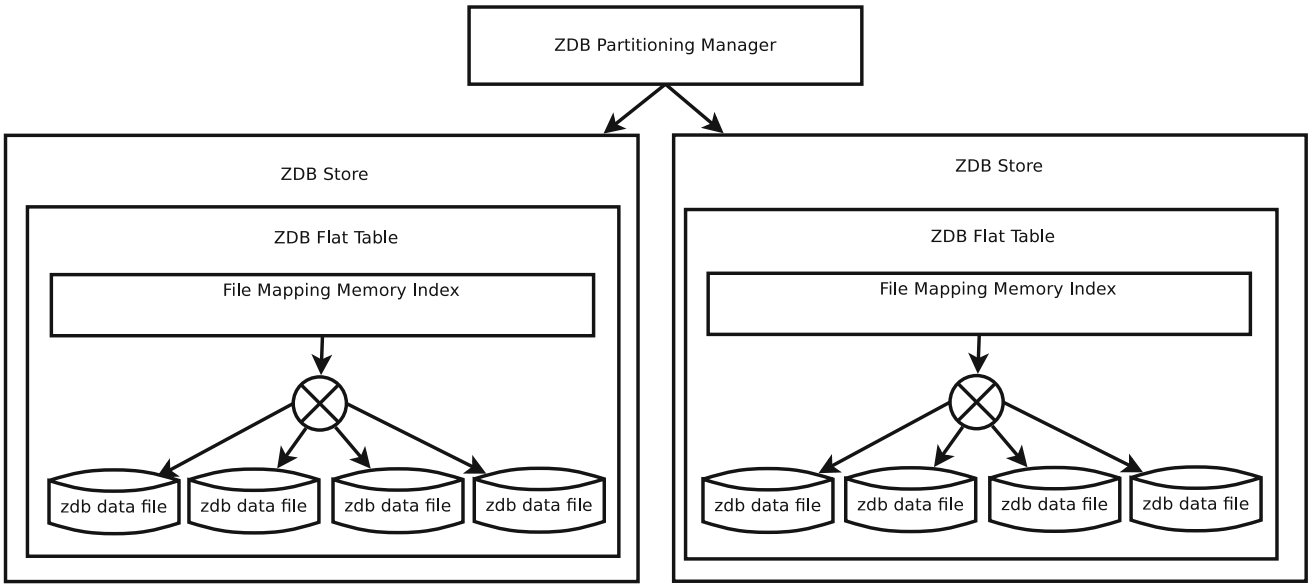

Fig. 1 ZDB architecture

– **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:

$$\text{ArraySize} = k_{\max} - k_{\min} \tag{1}$$

Basing on the values of the range, the ith item in the array refers to the position of the key $(i + k_{\min})$ in the data file. It is also important to note that the size of each item in the array depends on the maximum file size we want to support. In ZDB, this may be 4, 5, 6, 7 or 8 bytes for easy configuration and for tuning the memory usage and maximum data file size of the key-value persistent store. Comparing ZDB and FAWN [8], 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 are 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.

2.1.2 Index example

In social networks such as Facebook [1] and Flickr [2], and in email hosting websites such as GMail [17], 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 [31] 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 range of keys for example [0, 1,000,000) in a partition. The size of the array is 1,000,000. 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 64 GB for all the files. Since the index size is 1,000,000, the memory size for the index is 4*1,000,000 bytes (about 4 MB). In one partition, the size of the index table can be hundred millions.

2.2 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.

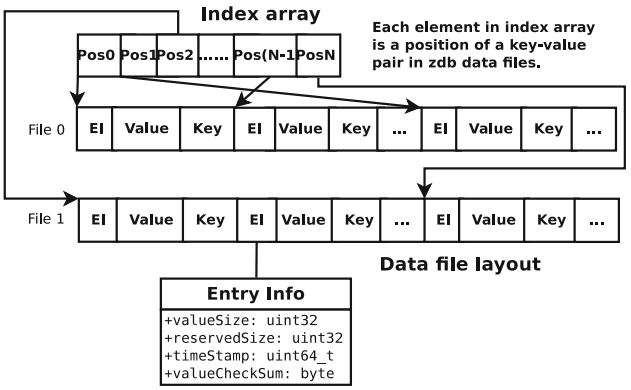

Fig. 2 Data file layout with 2 data files

Entry information consists of: value size: 4 bytes, reserved size: 4 bytes, time stamp: 8 bytes, value check sum: 1 byte. The layout of ZDB log storage files is described in Fig. 2.

2.3 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 has some interfacing commands to interact with the data store that include get, put, and remove. ZDB flat table also has two iterating commands: key-order iterating and insertion order iterating. Using iterating commands, it is able to scan through the table to get all key-value pairs (Fig. 3).

– Put key-value pair to store

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 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 $(\text{key} - k_{\min})$th item in the index array. The entry is then appended to the end of the data file.

– Get operation

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 $(\text{key} - k_{\min})$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.

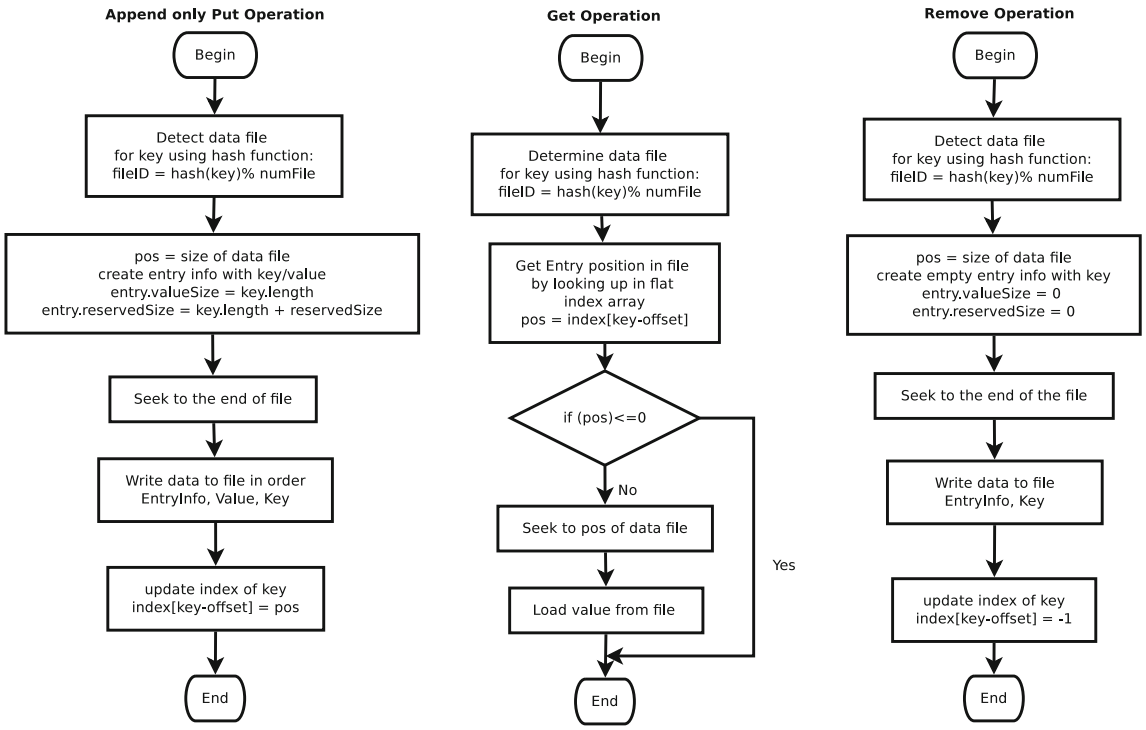

Fig. 3 Put, get, remove algorithms of ZDB flat table

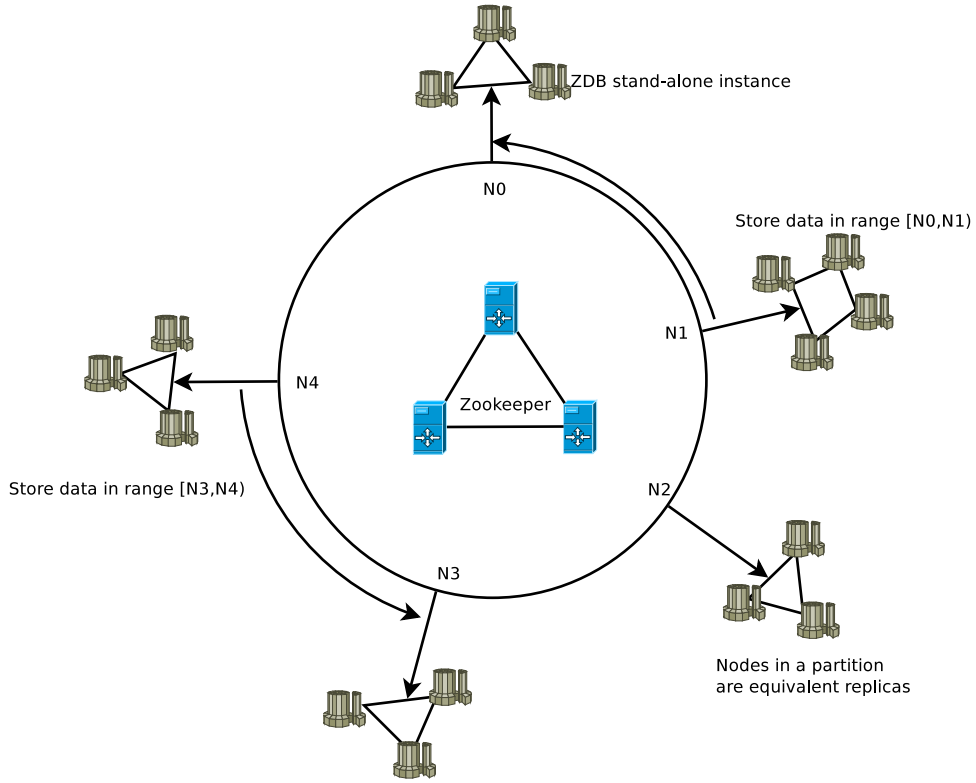

Fig. 4 Data partitioning

– 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. An entry info that indicates the pair with the key was removed is created and append to the 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 is scanned, if the item in array is greater than or equal to 0, the key associated with that item has the value in the ZDB log storage, and the value is 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.

2.4 ZDB store

ZDB store uses ZDB flat table's functionality and handles all data store requests from applications. ZDB store uses thrift protocol [27] to serve request from clients. ZDB store also provides compact operation for release disk space used by multiple writing to a key. In normal mode, ZDB Store has one ZDB flat table for read and write key-value data.

2.5 Compacting

In append-only mode, after writing the value of a key, the old value will be unused, so the disk space for old values is wasteful. Compacting operation is used to cleanup the old values and get more free disk space.

Compacting operation works as follows:

– Create new ZDB flat table.
– Sequential iterate on HDD or old table and put data to created table.

When compacting, the ZDB store is still working as follows:

– Every put operation, write data to the new table.
– Every get, firstly, try to read from new table, if not found key-value entry in the new table, try to get the value from the old table.
– Every remove, remove the entry from both tables.

2.6 Data partitioning

For big amount of data items, large range of key space, it is necessary to distribute data to multiple ZDB instances and scale the system as data growing. ZDB distributes key-value pairs in clusters using consistent hash. Every key is hashed to get a hash value, we assume that all hash values are in range $[0, H_{\text{bound}})$, partition manager uses this hash value to decide which ZDB instances store the associate key-value pair.

2.6.1 Consistent hash

Logically, ZDB instances are placed in a ring, each instance has a mark value in $[0, H_{\text{bound}})$ that indicate its position in the ring. A partition consists of instances with the same mark value. Instances in a partition will store data of the same key range or they are replications of each other. Assume distinct mark values are $N_0 < N_1 < \cdots < N_p$. Instance with mark value, N_i will store key-value pairs which have hash value of the key in range $\left[N_{i-1}, N_i\right)$, the keys with hash value greater or equal to N_p and the keys with hash value in range $[0, N_0)$ are stored in the instances with hash value N_0. With auto-incremental integer keys, we can ignore hash function and get hash value equal to the key, each partition stores a range of continuous keys (Fig. 4).

2.6.2 Data range configuration

Each ZDB instance is configured to store data in a range. ZDB uses zookeeper [18] for co-ordination of the configuration of ZDB instances. Each instance registers a path in zookeper and the mark value of each instance with the following format:

/servicepath/protocol_mv:ip:port

where mv is the mark value of the instance. This path in zookeeper associates with the string value:

"protocol_mv:ip:port"

Partition manager monitors and watches paths in zookeeper and tells client the host and port of ZDB Services to access the data. For example with two instances:

/data/zdb/thriftbinary_10000000:20.192.5.18:9901
/data/zdb/thriftbinary_20000000:20.192.5.19:9901

In this case, the path */data/zdb* is watched by partition manager. Every change in that path's children will be captured and update the configuration to the clients.

2.7 Data consistency

ZDB uses chain replication [30] for replicating data in cluster. Every writing operation works on all nodes in the cluster asynchronously. ZDB applies Eventually consistent model from [15].

2.8 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 important to note that string keys in a bucket 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 consists 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.

2.9 ZDB service

ZDB instance is developed as a server program called ZDB service. It uses Thrift [27] to define interface and uses thrift binary protocol to implement rpc service. The interface of ZDB instance as Listing 1 follows:

Listing 1 ZDB Thrift Interface

```
typedef string KType
typedef string VType
typedef list<KType> KeyList
struct DataType{
    1: required KType key,
    2: required VType value,
}
typedef list<DataType> DataList

service ZDBService{
    ValueType get(1:KType key),
```

```
    DataList multiGet(1: KeyList keys ),
    i32 remove(1:KType key),
    i32 put(1:KType key, 2:VType value),
    void multiPut(1:DataList data),
    bool has(1:KType key),
}
```

ZDB service is written in C++, using thrift binary protocol with nonblocking IO. It has a small configurable Cache for caching the data. It has two writing modes: safe writing mode and asynchronous writing mode. In safe writing mode, all key-value pairs are written to Cache then flush to ZDB on disk immediately. In asynchronous writing mode, all key-value pairs are written to Cache, and the keys are marked as dirty, then flushing threads collect the dirty keys and flush the data to ZDB data file on disk asynchronously in background. The writing mode can be changed at runtime for the ease of tunning the performance. The cache of ZDB service is implemented using popular cache replacement algorithms such as least recent used (LRU), ARC [21].

3 Related works

Small index large table (SILT) [20] 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. SILT's multi-store design uses a series of basic key-value stores optimized for different purposes. However, the basic design of SILT's LogStore works like ZDB in some respects. 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 [25] to map the keys to the buckets and still have false positive disk access while the ZDB has no false positive disk access. It is also important 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 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 creates 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 data file 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 HashStores because Each LogStore contains only 128 K 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) [8] 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 bit 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's 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 4 GB. Another difference between ZDB and FAWN lies in the hashing of original key in FAWN by SHA. It can not 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. With ZDB, there is no incorrect flash/hdd retrieval.

Cassandra [19] is a distributed column-based nosql store. Cassandra uses Thrift [27] to define its data structure and the RPC interface. There are some important concepts in Cassandra's data model: *Keyspace, ColumnFamily, Column, SuperColumn*. *Keyspace* is a container for application data. It is similar to a data schema in relational database. Keyspace contains one or more *Column Family*. The *Column Family* in Cassandra is a container for rows and columns. It is similar to a table in relational database. However, *Column Family* in Cassandra is more flexible than a table in relational database that each row in Cassandra can have different set of columns. *Column* is basic unit of data in Cassandra's *Column Family*, it consists of a name, a value and an optional timestamp. *SuperColumn* is a collection of Columns.

Each *Column Family* in Cassandra maintains an in-memory table and one or more on-disk structures called SSTable to store data. Every write operation to Cassandra firstly is recorded to a commit log, then apply the write to the in-memory table. The in-memory table is dumped to disk

and becomes a SSTable when it reaches its threshold which is calculated by number of items and data size. Every read operation in Cassandra, firstly lookup in the in-memory table. If the data associated with the key are not found in the in-memory table, Cassandra will try to read it from SSTable on disk. Although Bloom filter is used to reduce number of unnecessary disk reads when reading data from Cassandra, the latency when reading data from multiple SSTable is still relative high when data grow and the bloom filter has false positive. ZDB ensures that it needs maximum one disk seek in every read operation on disk. So, it can minimize the latency for mis-cache read operations.

Redis [26] is an in-memory structured key-value store. All data in Redis are placed in the main memory, redis also support persistent snapshot using disk data structure called RDB. Redis can dump data in the memory to RDB after specific time intervals. For recovery, Redis has a commit log called append-only file (AOF) which records all write operations and be played at server startup to reconstruct original data set. Both AOF and RDB are used to recover Redis's in-memory data when it crashs or restarts or moving data to another server. Although Redis has persistent ability using RDB and AOF, its maximum capacity is limited by the size of main memory. When the size of total data is bigger than Redis's maximum memory size, some data are evicted by eviction policies described as below [26].

- Noeviction: return errors when the memory limit was reached and the client is trying to execute commands that could result in more memory to be used.
- Allkeys-lru: evict keys trying to remove the least recently used (LRU) keys first, to make space for the new data added.
- Volatile-lru: evict keys trying to remove the least recently used (LRU) keys first, but only among keys that have an expire set, to make space for the new data added.
- Allkeys-random: evict random keys to make space for the new data added.
- Volatile-random: evict random keys to make space for the new data added, but only evict keys with an expired set.
- Volatile-ttl: to make space for the new data, Redis evicts only keys with an expired set, and tries to evict keys with a shorter time to live first.

LevelDB [4] is an open source key-value store developed by Google, originated from BigTable [11]. It is an implementation of LSM-tree [24]. It consists of two MemTable and set of SSTables on disk in multiple levels. Level-0 is the youngest level. When a key value is written into LevelDB, it is saved into commit log file firstly, then it is inserted into a sorted structure called MemTable. Memtable holds the newest key value. When MemTable 's size reaches its limit capacity, it will be a read-only Immutable MemTable. And a new MemTable is created to handle new updates. A background thread converts Immutable MemTable to a level-0 SSTable on disk. Each level has its own limit size, when the size of a level reaches the limit size, its SSTables will merge to create a higher level SSTable.

Table 1 Workload parameters

Workload name	Parameters	
	Put proportion	Get proportion
Write only	1	0
High read/low write	0.9	0.1
Low read/high write	0.1	0.9

4 Performance evaluation

The performance comparison of a key-value store is important especially if users have to choose among various available options. In this research, we use a standard benchmark system and a self-develop simple load tests to evaluate ZDB.

4.1 Standard benchmark

Yahoo! cloud system benchmark (YCSB) [12] is used to define workloads and evaluate and compare performance of ZDB and some popular key-value stores: LevelDB, HashDB of Kyoto Cabinet and Cassandra.

To minimize the difference of environment of ZDB and other key-value engines in this benchmark, popular open source persistent key-value stores engine is also wrapped into ZDBService: LevelDB [4] and Kyoto Cabinet's hash db [3]. The wrapping of LevelDB and KyotoCabinet is similar to MapKeeper [6]. We can change ZDBService's configuration to switch between ZDB, Kyoto Cabinet, LevelDB. We also compare them with Cassandra.

The comparison using two servers with configuration below

Operating System CentOS 64 bit
CPU Intel Xeon Quad core
Memory 32G DDR
HDD 600G ext4 filesystem
Network Wired 1 Gbps

We defined some workloads in YCSB for evaluation in Table 1.

We ran above workloads with different record sizes: 1 KB, 4 KB and tracked the performance when the number

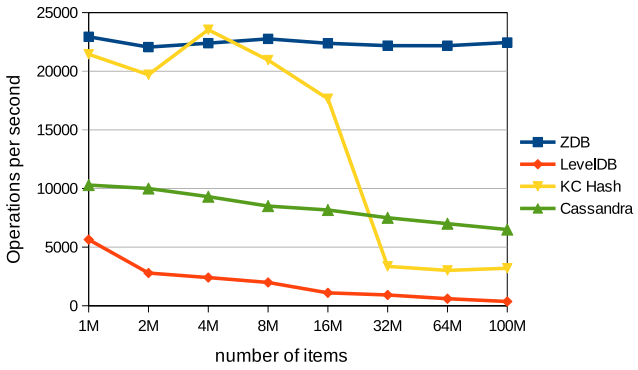

Fig. 5 Write only 1 KB records using YCSB

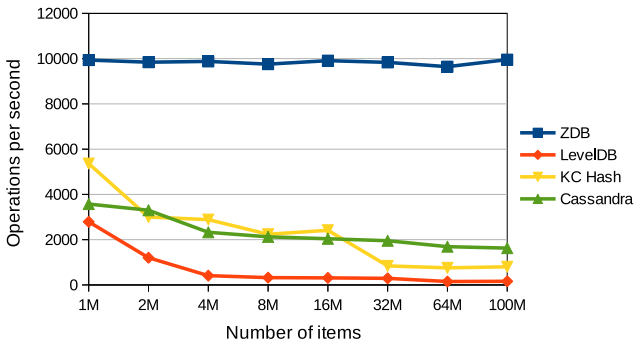

Fig. 6 Write only 4 KB records using YCSB

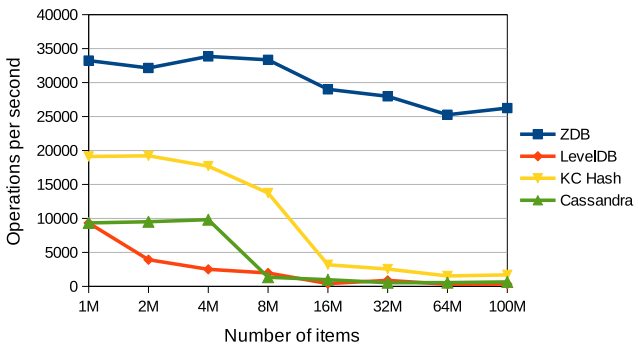

Fig. 7 High read/low write 1 KB records using YCSB

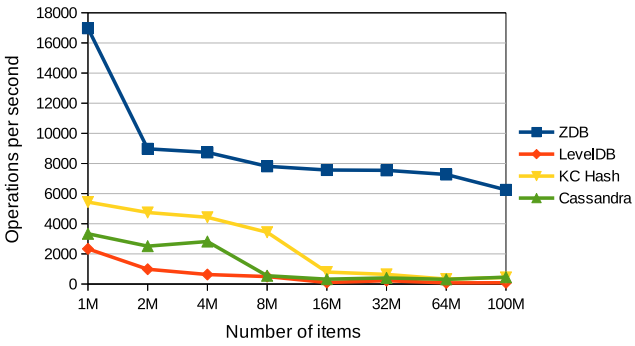

Fig. 8 High read/low write 4 KB records using YCSB

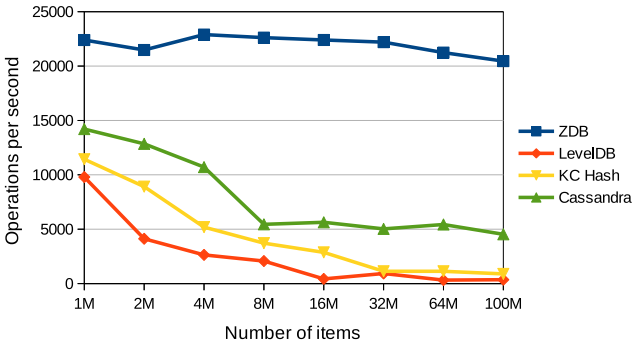

Fig. 9 High write/low read 1 KB records using YCSB

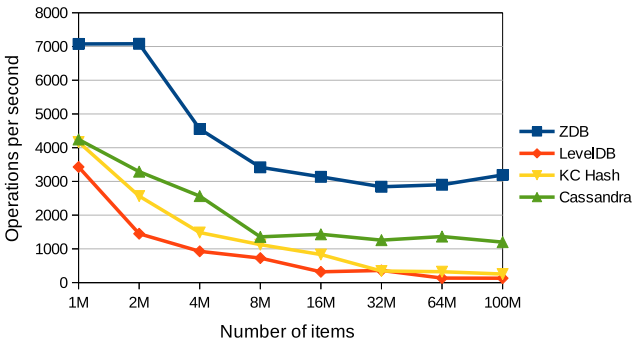

Fig. 10 High write/low read 4 KB records using YCSB

of records growing. We used two servers connected in high-speed LAN, the first server is used to run data services and another is used to run YCSB client with eight threads for the benchmark.

The benchmark results are shown in figures below. The horizontal axis shows number of items stored in data service. The vertical axis shows the number of operations per second we measured from running YCSB workload.

The benchmark results for Write-only workload are shown in Fig. 5 for record size of 1 KB and Fig. 6 for record size of 4 KB. ZDB writing performance is more stable than others. Figures 7, 8, 9 and 10 show that the result for transaction workloads consists of both read and write operations with portion parameter in Table 1 and with different record sizes.

4.2 Engine evaluation

We also use our simple benchmark tool written in C++ without using ZDB service to eliminate overhead from RPC framework and to avoid cost from Java-based code of YCSB to compare performance of ZDB with Kyoto Cabinet and LevelDB. The test cases are described as follows:

– Writing 100 million key-value pairs with variable value size in one thread.

Table 2 One writing thread

DBType	Cases		
	Key: 4 bytes Value: 4 bytes	Key: 4 bytes Value: 1 KB	Key: 4 bytes Value: 100 KB
LevelDB	347,246	5,360	61
KC	343,348	10,268	1,872
ZDB	294,796	108, 790	4,132

Table 3 Four writing threads

DBType	Cases		
	Key: 4 bytes Value: 4 bytes	Key: 4 bytes Value: 1 KB	Key: 4 bytes Value: 100 KB
LevelDB	369,760	15,004	90
KC	241,800	80,420	1,920
ZDB	537,204	128,220	5,248

Table 4 Random reading

DBType	Cases		
	Key: 4 bytes Value: 4 bytes	Key: 4 bytes Value: 1 KB	Key: 4 bytes Value: 100 KB
LevelDB	304,448	4,629	62
KC	1,176,300	45,234	5,075
ZDB	1,326,205	60,325	6,232

- Writing 100 million key-value pairs with variable value size in four threads.
- Random reading key-value from stores.

The benchmark results are shown in tables below, the number in the table shows the number of operations per second. ZDB has the highest number of operations per second in most scenarios. The results without overhead from RPC framework and YCSB workload generator are better than before.

In the first instance, the key-value store engines are setup with one writing thread with keys of 4 bytes and value of 4 bytes, keys of 4 bytes and values of 1,024 bytes, and keys of 4 bytes and values of 100 KB. The results in Table 2 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.

The benchmark was repeated with four writing threads and the results are shown in Table 3. It shows that ZDB works 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 4.

4.3 Discussion

As result presented above, the performance of both Kyoto Cabinet HashDB and LevelDB drops when data growing, while ZDB's performance is relative stable for both writing and reading. Kyoto Cabinet HashDB is organized as a hash table on disk. *mmap*() is used to map the head portion of data file for fast access (default mapping size is 64 MB) of the hash table. It used chaining for collision resolution. When number of item and total data size are small, HashDB of Kyoto Cabinet has a very good performance. But when data are big, the memory mapping is not enough to store all data, more over, every write operation for writing key-value pair of Kyoto Cabinet HashDB always needs to lookup the record in its bucket in the hash table. So, when the key value existed and being rewritten, it needs more disk IO than ZDB. That is why HashDB of Kyoto Cabinet is very fast with small data, and being slower when data growing. *LevelDB* is an Implement of LSM-tree. With small key-value items, LevelDB has a good performance. When the write rate is high and the key-value's size is relative big, the MemTable of LevelDB reaches its limit size rapidly, and it has to be converted to SSTable. And when many SSTables have to merge to higher level SSTable, the number of disk I/O operations increases, so the overall performance of LevelDB with big key-value drops. *Redis* is not in this comparison because It stores all data in main memory, when total size of data is bigger than main memory size, some data are evicted and lost. ZDB, LevelDB, Kyoto Cabinet and Cassandra can store data permanently on hard disk or SSD.

5 Conclusion

ZDB uses efficient 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 data file. Common interfacing commands such as Put, Get, and Remove are implemented in ZDB. It has a flexible item sizes to allow for tuning to enhance better performance. To reduce the number disk seeks, file appending is used and one-seek write is implemented. ZDB flat index is designed as a linear array on File Mapping is used to fast lookup without false positive the position of key-value pairs stored in data files. In all operations, ZDB needs at most one disk seek. In addition, all writing operations are sequential. For applications that require a high performance with optimized disk reading and writing operations, especially for big value, ZDB can be a good choice.

Acknowledgments Zing Me social network supported infrastructure and its data for this research's analysis and experiments. We thank VJCS reviewers very much for their meaningful feedbacks.

References

1. Facebook. http://facebook.com. Accessed 15 Jan 2013
2. Flickr. http://www.flickr.com. Accessed 15 Jan 2013
3. Kyoto Cabinet: a straightforward implementation of dbm. http://fallabs.com/kyotocabinet. Accessed 1 May 2013
4. Leveldb—a fast and lightweight key/value database library by google. http://code.google.com/p/leveldb. Accessed 23 Jul 2013
5. Oracle berkeley db 12c: persistent key value store. http://www.oracle.com/technetwork/products/berkeleydb. Accessed 30 Sep 2013
6. Mapkeeper. https://github.com/m1ch1/mapkeeper. Accessed 1 Jun 2014
7. Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., Nath, S.: Cheap and large cams for high performance data-intensive networked systems. NSDI **10**, 29–29 (2010)
8. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan V.: Fawn: a fast array of wimpy nodes. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 1–14. ACM (2009)
9. Badam, A., Park, K., Pai, V.S., Peterson, L.L.: Hashcache: Cache storage for the next billion. NSDI **9**, 123–136 (2009)
10. Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P., et al.: Finding a needle in haystack: Facebook's photo storage. OSDI **10**, 1–8 (2010)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
13. Debnath, B., Sengupta, S., Li, J.: Flashstore: high throughput persistent key-value store. Proc VLDB Endow **3**(1–2), 1414–1425 (2010)
14. Debnath, B., Sengupta, S., Li, J.: Skimpystash: Ram space skimpy key-value store on flash-based storage. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 25–36. ACM (2011)
15. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. SOSP **7**, 205–220 (2007)
16. Fitzpatrick, B.: A distributed memory object caching system. http://www.danga.com/memcached/ (2013). Accessed 4 Sept 2013
17. Google, Gmail. http://mail.google.com. Accessed 15 Jan 2013
18. Hunt, P., Konar, M., Junqueira F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, vol. 8, pp. 11–11 (2010)
19. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
20. Lim, H., Fan B., Andersen, D.G., Kaminsky M.: Silt: a memory-efficient, high-performance key-value store. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 1–13. ACM (2011)
21. Megiddo, N., Modha, D.S.: Arc: a self-tuning, low overhead replacement cache. FAST **3**, 115–130 (2003)
22. Min, C., Kim, K., Cho H., Lee S.-W., Eom, Y.I.: Sfs: random write considered harmful in solid state drives. In: Proceedings of the 10th USENIX Conference on File and Storage Technolgy (2012)
23. Mogul, J.C., Chan, Y.-M., Kelly, T.: Design, implementation, and evaluation of duplicate transfer detection in http. NSDI **4**, 4–4 (2004)
24. ONeil, P., Cheng, E., Gawlick, D., OGNeil, E.: The log-structured merge-tree (lsm-tree). Acta Inform. **33**(4), 351–385 (1996)
25. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
26. Sanfilippo, S., Noordhuis, P.: Redis. http://redis.io. Accessed 7 Jun 2013
27. Slee M., Agarwal A., Kwiatkowski, M.: Thrift: scalable cross-language services implementation. Facebook White Paper, 5 (2007)
28. Tevanian A., Rashid R.F., Young M., Golub D.B., Thompson M.R., Bolosky W.J., Sanzi R.: A unix interface for shared memory and memory mapped files under mach. In: USENIX Summer, pp. 53–68. Citeseer (1987)
29. van Dijk, T.: Analysing and improving hash table performance. In: 10th Twente Student Conference on IT. University of Twente, Faculty of Electrical Engineering and Computer Science (2009)
30. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. OSDI **4**, 91–104 (2004)
31. VNG. Zing me. http://me.zing.vn. Accessed 19 May 2013
32. Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., Najjar, W.A.: Microhash: an efficient index structure for flash-based sensor devices. FAST **5**, 3–3 (2005)