SASS: A High-Performance Key-Value Store Design for Massive Hybrid Storage

Jiangtao Wang, Zhiliang Guo, and Xiaofeng Meng(✉)

School of Information, Renmin University of China, Beijing, China
{jiangtaow,zhiliangguo,xfmen}@ruc.edu.cn

Abstract. Key-value(KV) store is widely used in data-intensive applications due to its excellent scalability. It supports tremendous working data set and frequent data modifications. In this paper, we present SSD-assisted storage system (SASS), a novel high-throughput KV store design using massive hybrid storage. SASS meets three exclusive requirements of enterprise-class data management: supporting billions of key-value pairs, processing thousands of key-value pairs per second, and taking advantage of the distinct characteristics of flash memory as much as possible. To make full use of the high IOPS of sequential write on the SSD, all modification operations are packaged as operation logs and appended into SSD in the time order. To handle the tremendous number of key-value pairs on hard disk, a novel sparse index, which can be always kept in the SSD, is proposed. Moreover, we also propose an in-memory dense index for the operation logs on SSD. Our evaluation mainly characterizes the throughput of read and write, namely the ops/sec(**get-set** operations per second). Experiments show that our SASS design enjoys up to 96806 write ops/sec and 3072 read ops/sec over 2 billion key-value pairs.

Keywords: Key-value · Solid state disk · Cache · IOPS

1 Introduction

With the rapid development of Internet technologies, many web applications, such as internet services, microblogging network, and multi-player gaming, need to consistently meet the service requests of user within fast response time. The traditional disk-based relational database systems can hardly support the high-concurrent access gracefully. Recently, a lot of server-side applications have preferred to use noSQL databases implemented by key-value stores to provide high-throughput performance. Compared to the traditional relational database, key-value storage exhibits better scalability, efficiency and availability. Without

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61379050,91224008); the National 863 High-tech Program (No. 2013AA013204); Specialized Research Fund for the Doctoral Program of Higher Education(No. 20130004130001), and the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University(No. 11XNL010).

complex command parse or execution plan optimization, key-value storage systems can enjoy excellent ops/sec performance. Hence, a key-value storage system is a better choice for the web applications which need to meet the data durability and high performance requirements. Furthermore, the technology of flash memory offers an alternative choice for storage system designers.

Over the past decades, flash-based solid state disk(SSD) is making deep inroads into enterprise applications as its increasing capacity and dropping price. Many web service providers have used flash memory to improve their system performance. However, the comparatively small capacity and high price hinder flash memory from a full replacement of hard disks. Although SSD RAID technology[1,2] makes the high-capacity flash memory device possible, the hybrid storage is still a prevalent mode. A key challenge in the hybrid storage is how to take full advantage of the flash memory to maximize the system performance.

In this paper, we present the design and evaluation of SSD-assisted storage system (SASS), a key-value store design supporting massive hybrid storage and high throughput. SASS has a three-part storage architecture that integrates main memory, SSD, and hard disk, in which we take flash memory as the write cache for the hard disk. In the main memory, we allocate a cluster of separate log buffers. All data modification operations (*insert*, *delete*, and *update*) are not immediately written back to the hard disk. Instead, they are stored in these log buffers as *operation logs*. When these buffers become full, these logs are appended to the log file on the SSD, and eventually merged with the original data on hard disk under certain conditions. As a result, we can take advantage of the high IOPS of sequential write on SSD and maximize the write throughput. In order to process random *get* query, we propose a sparse index, a hierarchical bloom filter residing in the SSD, to manage the tremendous number of key-value pairs on hard disks. Furthermore, we also design an in-memory index, *operation list*, for the operation logs on SSD. In general, a random *get* query can be answered by one flash read in the best case and one extra hard disk read in the worst case. The contributions of this paper are summarized as follows:

1. We present a novel key-value store design, called SSD-assisted storage system (SASS), which aims to support large scale data-intensive applications. In SASS, SSD serves as a write cache for the hard disk, and the key-value pairs are organized into blocks on hard disks, while the recent modifications of the key-value pairs are buffered in the SSD as operation logs. The query processing procedure is further accelerated by two novel index mechanisms.
2. We implement an industry-strength SASS system and conduct extensive experiments on it. The evaluation results demonstrate that SASS enjoys up to 96806 write IOPS with key-value pairs log buffers, which outperforms BerkeleyDB to 9.46x. Moreover, as the introduction of SSD and hierarchical bloom filter index, SASS provides 2.98x speedup over BerkeleyDB when measuring the IOPS of read operation.

The rest of the paper is organized as follows. In Section 2, we present an overview of SASS and some critical components of SASS, including data organization, hierarchical bloom filter and operation list. Some system maintenance operations and how to achieve concurrency control are explained in Section 3. Section 4 gives the results of our experiment evaluation and Section 5 surveys the related work. Finally, Section 6 concludes this paper.

2 SASS Design and Implementation

2.1 Overview of SASS

Fig. 1 gives an overview of SASS, which employs a three-part storage architecture integrating main memory, SSD, and hard disk. To support massive data volume, we take hard disk as the main storage medium considering its large data capacity. SSD is used to the write cache for the hard disk due to its high IOPS performance. Some recently modified key-value pairs are cached in the SSD, which will be merged to hard disk eventually. Thus, for a query request, SASS always check the SSD at first to see if the most fresh key-value pair exists. If it does, SASS just reads it from SSD. Otherwise, SASS checks the hard disk.

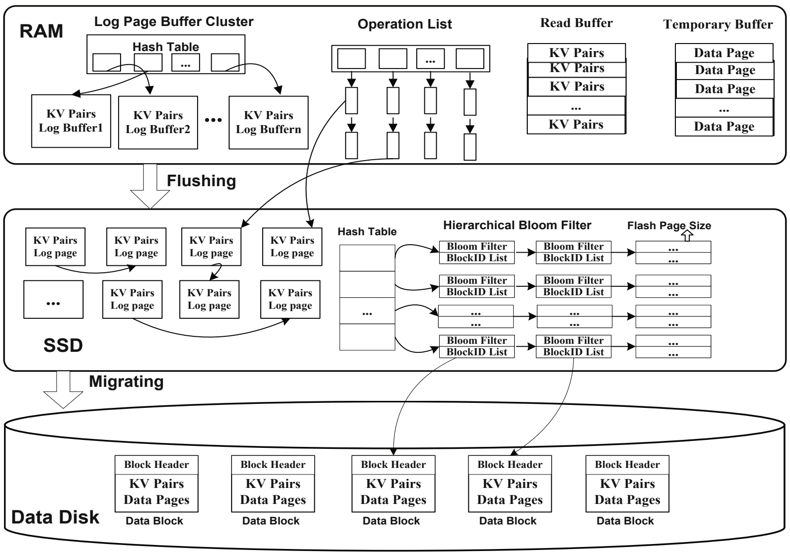

Fig. 1. An overview of SSD-assisted storage system (SASS)

2.2 Block, Page and Record

In SASS, the disk-resident key-value pairs is managed at a block granularity. All the incoming key-value pairs are distributed into different data blocks by a hash function. However, the big block size introduces a problem, that is, we have to read a block even if we just want to get a single key-value pair, which is inefficient and memory-consuming. So, we introduce data page into SASS. A block consists of multiple data pages, and a block header structure is designed for each block to summarize the key-value pair distribution in a data block. We can get the requested key-value pair through checking its block header. The key-value pair is stored as a variable-length record. Fig. 2(a) gives the layout of a record, which consists of three fields: *Record Size* stores the size of the record, *Key Size* stores the size of the key, and *Data Value* stores the key-value pair.

2.3 Operation Log, Operation Log Page

In SASS, all key-value pair modification operations are first stored in the SSD as operation logs. The operation logs are organized into log pages following the order of timestamp. As more and more operation logs are flushed to the SSD, the first few log pages of the list will be merged and moved to the hard disk over time. By calculating the hash value of a given *key*, the key-value pairs which share the same hash value are accumulated to form a data block. All the data blocks with the same hash value are clustered into a partition. In log page, a key-value pair is stored as a variable-length record. Fig. 2(b) gives the layout of an operation log record on SSD, which also consists of three fields: *Log Size* stores the size of the log record, *Partition_ID* identifies the target partition and *Record* stores the key-value pair. We store the new key-value pairs in the *Record* field for insert and update operations and *null* for delete operation.

2.4 Log Page Buffer Cluster, Read Buffer and Temporary Buffer

In the main memory, there are three types of buffers: log page buffer, read buffer and temporary buffer. The log page buffer cluster consists of a set of log page buffers, each log page buffer is a fixed-size data structure that is used to buffer the dedicated key-value pairs by a hash function. Specifically, when a new key-value pair is generated, SASS uses a hash function to locate an associated partition, and assigns a proper log page buffer to hold it. Each log page buffer shares the same size with a data page, when the log page buffer is full, the accumulated logs will be appended to the SSD as a log page. Read buffer is also a fixed-size data structure to cache the recently read data, including data pages, log pages and block headers. The least recently used (LRU) pages will be evicted when read buffer runs out of free space. Temporary buffer, as its name suggests, is a temporary data structure used for the merge operations.

2.5 Operation List

In SASS, we store all the recent updated data on SSD as operation logs, just depicted in Fig. 2(b). Whenever a query request arrives, SASS firstly checks the operation logs cached in the SSD. Upon a miss on the SSD, the query continues to look up the key-value pairs on hard disks. Consequently, an index for the operation logs on SSD is definitely necessary to accelerate the check. We design an index for the operation logs on SSD, namely operation list. Fig. 3 shows the structure of operation list. Operation list is an array of doubly linked lists of operation elements, which uses a mapping table to maintain all the operation logs on SSD. In the mapping table, the key is the *Partition_ID* while the mapped value is OpHeader. As soon as an operation log is flushed to the SSD, a new operation element pointed to that operation log will be created and linked to the corresponding doubly linked list. Actually, we just need one doubly linked list for a partition. However, we make an array of double linked lists for each partition to avoid a double linked lists with too many elements which can be a nightmare for query. For each operation element to be linked, we compute a HashCode using division method with the key at first and then link the element to the double linked list with the same HashCode. In this way, we can transform a long list into many short lists and reduce the query cost. Fig. 2(c) describes the layout of an operation element, which contains five fields: *Operation Type* marking the type of the operation, *Log Address* keeping the exact address of the operation log record on SSD, *PrevOpElement* and *NextOpElement* pointing to the previous and next OpElements respectively, *Key* representing the key of the operated record. In this way, we can arrange the operation log records targeting on a certain partition to a doubly linked list from the head to the tail in the order of their arrival time. For a query with specific key, we can find the corresponding list and traverse the list from tail to head to look up the first operation element with the same key.

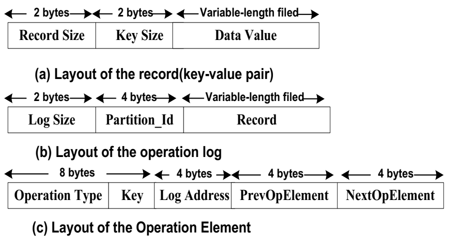

Fig. 2. Layout of record, operation log and operation element

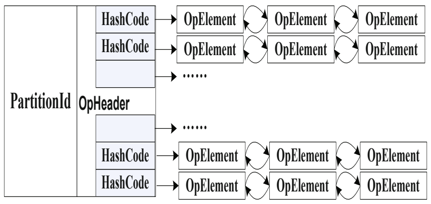

Fig. 3. Structure of operation list

2.6 Hierarchical Bloom Filter

Hierarchical bloom filter is a sparse index, it is designed for indexing the key-value pairs migrated to hard disk. Because the in-memory index needs to take up a considerable amount of buffer space, so, we use SSD to store the hierarchical bloom filter. Bloom filter is a space-efficient probabilistic data structure which supports set membership queries. One single bloom filter designed for flash memory may suffer from the drawback that inserting a key almost always involves a lot of flash page writes, since the k bit positions may fall into different flash pages. Considering the poor random write of flash memory, we design a hierarchical bloom filter index structure. Fig. 4 shows the hierarchical bloom filter, which can be taken as a bloom filter tree. On the lowest level of the tree, namely the leaf level, each leaf node contains an independent bloom filter which occupies a single flash page. A leaf node summarizes the membership of the keys which are scattered in multiple disk-resident blocks. That is, each independent bloom filter is responsible for indexing one or more specific blocks. To insert or lookup an element, we employ a hash function to locate the sub-bloom filter that the requested key-value pair may reside in. Then, k bit positions are identified within the sub-bloom filter for setting or checking the bits. Thus, this design requires only one flash page read per element lookup. To further to accelerate the key lookup, we also add a block header list for each independent bloom filter. When a key is identified to fall into some sub-bloom filter, we can locate the block which the requested key-value pair resides in by searching the block header list.

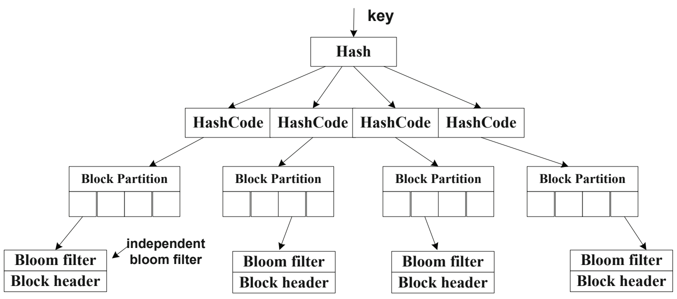

Fig. 4. Structure of hierarchical bloom filter

2.7 Key Set and Get Operations

SASS supports the following basic operations: *insert*, *update*, *delete*, *get*, as well as *merge*. In this section, we will explain how they work in SASS.

1)set: SASS transforms all the insert, update and delete operations into operation logs and appends them to the SSD. Subsequently, the corresponding operation elements are created and linked to the tails of the target list. In this way, SASS transforms the random write into sequential write and maximize the write throughput. Actually, all these operations will not return until the operation logs are flushed to SSD for guaranteeing the durability of data.

2) get: A *get* query uses a key to retrieve a key-value pair. Given the key in the *get* query, the id of the partition that contains the key is determined at first by checking the in-memory hash table. Then, we can get the corresponding double linked list to the partition id from the operation list. The first element with the same key can be found by traversing the list of operation elements from tail to head. Upon a hit on the list, we will check the operation type, *insert* or *update* indicates the record is resident on SSD, and *delete* indicates the record had been eliminated recently. Hence, the data address will be returned for inserting or updating and *null* for deleting. Otherwise, we will search the hierarchical bloom filter to locate the page in which the key-value pair resides.

3 Advanced Issues

This section discusses some advanced and important issues for SASS.

Algorithm 1. Evict_SSDPage()

```
Require: The operation list list that triggers the merge operation
Ensure: Merge the operation log with the key-value pair residing hard disk
 1: PartitionId=Lookup_OP(list);/*locate the partitionID of the given list*/
 2: MergeBlock=GetBlock(PartitionId);/*get the block with the PartitionId*/
 3: for (ele = list.head;ele! = list.tail;ele = ele − >nextOpElement) do
 4:     if (ele.type == insert) then
 5:         data=GetData(ele.dataAddress);
 6:         if (MergeBlock is full) then
 7:             allocate a new MergeBlock for the incoming key-pair page;
 8:         InsertData(MergeBlock,data);
 9:     if (ele.type == update) then
10:         data=GetData(ele.dataAddress);
11:         if (MergeBlock is full) then
12:             allocate a new MergeBlock for the incoming key-pair page;
13:         UpdateData(MergeBlock,data);
14:     if (ele.type == delete) then
15:         DeleteData(MergeBlock,ele.key);
16: FlushAllBlockData();/*migrate the SSD-resident KV pairs to disk*/;
17: deltaindex=BulidDiskDataIndex();/*build index for the evicted KV pairs*/;
18: update deltaindex to the hierarchical bloom filter;
19: RemoveOplistFromSSDIndex(list);/*remove the list from the operation list*/;
20: return;
```

3.1 Merge

As the data modifications consume the space of SSD steadily, the operation log records in the oldest log pages have to be merged with their original data on the hard disk periodically. This process is managed by a *merge* operation. In general, two conditions will trigger a merge operation: the number of operation elements in an operation list exceeds a threshold and the flash memory usage exceeds a certain threshold. During the merging process triggered by a large number of operation elements, the corresponding log pages will be read into the temporary buffer and the list will be traversed from head to tail to execute the merge operation. During the merging process triggered by the flash usage, the oldest log blocks are chosen to be recycled. The maximum number of operation elements in an operation list is very a critical configuration parameter. A relatively small operation element number setting reduces the traversal time on the list but makes the merge operation more frequent. A relatively large operation elements number accelerates the data access but reduces the space that can be used by SSD itself, which is considered necessary for some internal operations (e.g., garbage collection). Algorithm 1 gives the detailed description of the merge operation.

3.2 Concurrency Control

To achieve high throughput, SASS must support multi-thread operations, which require an effective concurrency control mechanism. Temporary buffer is a temporary structure allocated for each thread exclusively, so there is no need to protect temporary buffer. For other shared data structures, Table 1 lists the related operations and lock strategies.

Read Buffer: A *get* query may check the read buffer to see if the target data pages are already cached. Upon a miss, the least recently used (LRU) pages will be evicted and then the query thread reads the target pages from SSD or hard disk. In fact, we do not write the data pages back to hard disk, since they are never modified. The only thing should be ensured is that the data pages being accessed by some threads cannot be evicted by other threads. Consequently, each query thread must hold a read lock on the target pages and cannot evict any pages until it holds the write locks on them.

KV Pairs Log Buffers: KV pairs log buffers collect the operation log records created by *insert*, *update* and *delete* operations in the order of their timestamps. When the operation log records fill up the buffer, all the write threads will be blocked until all log records in the buffer are flushed to the SSD. As the write threads and the flush thread have a producer-consumer relationship on both buffers, we use a producer-consumer lock to protect them.

Operation List: For operation list, *get* traverses the list to find the target operation element and all the elements on the list will be checked upon a miss. *Insert*, *update* and *delete* always add an element to the tail of the target list. *Merge* traverses and frees a part of or the entire list. For this situation, we must prevent these operations from being disturbed by each other. So, we choose to use the reader-writer lock.

4 Experimental Evaluation

We implement a key-value store with about 20,000 lines of C code and perform a serial of experiments on this system. As SASS aims to be a high throughput storage system, our evaluation mainly characterizes the throughput of read and write, namely the ops/sec. Our experiments run on a server powered by Intel Xeon CPU E5645 at 2.40GHz with 16GB of DRAM. We use the Seagate hard

Table 1. Lock Strategy Of Shared Data Structure

Share Data Structure	**Operation**	**Lock Strategy**
Read buffer	*get*	Reader-Writer
KV pairs log buffer	*insert*, *update*, *delete*	Producer-Consumer
Operation list	*get*, *insert*, *update*, *delete*, *merge*	Reader-Writer

disk store all the key-value pairs. The storage capacity of hard disk is set to 10TB. A 256G GB Intel SSD serves as the write cache.

We pick two different data sets, i.e., post messages and pictures, as our evaluation datasets. The data items in the former dataset are mostly small(100bytes $\sim$ 1000bytes) while the data items in the latter one are relatively large(10KB $\sim$ 100KB). To make a thorough evaluation of SASS, we chose two extreme data traces, Random Set and Random Get, to squeeze SASS for its maximum write and read performance. In addition, we also chose a typical data trace, Canonical, which is a normalized read-world workload. Table 2 describes the properties of each test set, in which a suffix L stands for large data (i.e, thumbnail pictures).

Table 2. Experimental Data Trace

Trace	**Number of Operations**	**get:set:update:delete**	**Value Size(KB)**
Random Set	4billions	0:1:0:0	0.1 $\sim$ 1
Random Set-L	4billions	0:1:0:0	10 $\sim$ 100
Random Get	4billions	1:0:0:0	0.1 $\sim$ 1
Random Get-L	4billions	1:0:0:0	10 $\sim$ 100
Canonical	2.5billions	64:8:4:1	0.1 $\sim$ 1
Canonical-L	2.5billions	64:8:4:1	10 $\sim$ 100

4.1 Set and Get Performance

We compare SASS with a popularly used database system, BerkeleyDB or BDB. BDB is a software library that provides a high-performance embedded database for key-value data, we select hash table to build the index for BerkeleyDB. To make a fair performance comparison, we implement the BerkeleyDB with a non-transactional data store mode, both SASS and BDB run on the same machines that we described above. We compare the performance of BDB and SASS using the workloads listed in Table 2.

Fig. 5 and Fig. 6 show the random set ops/sec of BDB, SASS over two datasets. In both figures, the random set ops/sec of BDB and SASS decrease when the number of concurrent threads increases. An exception is that the random set ops/sec of SASS does not decrease until the number of test threads exceeds 128. Especially for the microblog messages dataset(shown in Fig. 5), SASS provides a speedup of 9.5 times relative to BDB when the number of test

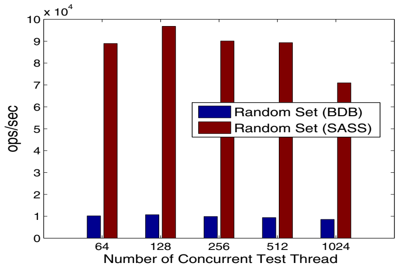

Fig. 5. ops/sec over Random Set

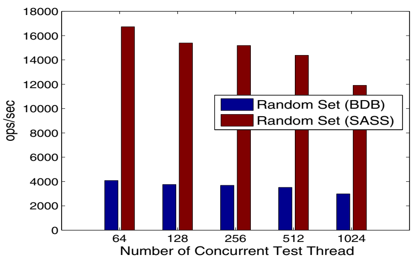

Fig. 6. ops/sec over Random Set-L

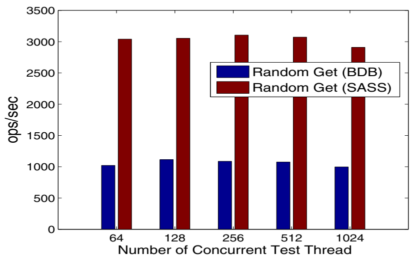

Fig. 7. ops/sec over Random Get

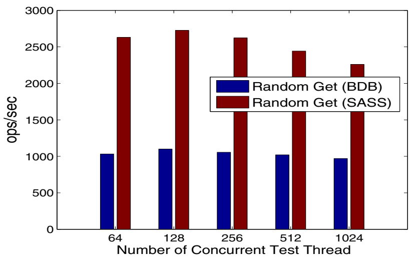

Fig. 8. ops/sec over Random Get-L

threads is set to 512. In addition, as SASS transforms all the random set into sequential write, SASS exhibits a higher ops/sec than that of BDB when dealing with the relatively large data items (shown in Fig. 6).

Fig. 7 and Fig. 8 show the random get ops/sec of BDB and SASS over two datasets. From both figures, we can see that both the random get ops/sec of BDB over microblog messages dataset and thumbnail pictures dataset display little difference. That is, the random get performance of BDB is non-sensitive to different datasets. In contrast, SASS shows better random get ops/sec over microblog messages dataset. Concerning the data items in microblog messages dataset are short, much more key value pairs can resident on SSD, so the performance improvement over microblog messages dataset makes sense. In general, a random get operation in SASS can be answered by one flash read for the best case and one extra hard disk read for the worst. However, a random get operation in BDB requires one hard disk read for the best case. Hence the random get performance of SASS always outperforms that of BDB.

Both random set and random get are extreme workloads and can hardly happen in practice, so we choose another typical workload, canonical. Fig. 9 and Fig. 10 exhibit the canonical ops/sec of BDB, SASS. For the canonical dataset, SASS gains the maximum of ops/sec when the number of test thread is set to 128, and provides a speedup of 8.49X compared to BDB.

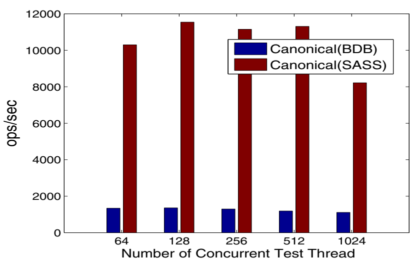

Fig. 9. ops/sec over Canonical

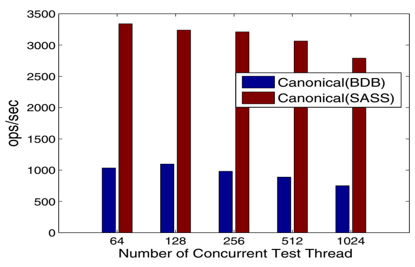

Fig. 10. ops/sec over Canonical-L

4.2 Impact of the Length of Operation List

Merge is the most expensive in all of the operations and it also affects the execution of other operations. As stated in Section 3, the two conditions trigger merge operation: the number of elements in an operation list exceeds a threshold and the amount of log pages usage on SSD as well exceeds another predefined threshold. Considering the relatively small capacity of SSD, the value of flash usage threshold is set to 90%. Consequently, we just tune the maximum number of elements in a OpHeader in the following evaluation.

With different maximum number of elements in a OpHeader, we conduct canonical workload again over two datasets. Besides, we count the ops/sec of set operations and get operations in canonical workload respectively, so that we can figure out how much the merge operations affect set operation and get operation. We vary the number of element in operation list from 512 to 2048. Fig. 11 and Fig. 12 display the ops/sec of set operation in canonical workload over two datasets. From the figures we can determine that the bigger number of elements the better ops/sec of set operation. Bigger number of elements in a OpHeader means less merge operations triggered by operation list. Merge operation holds an exclusive lock to prevent subsequent set operations, hence a bigger number of elements setting can improve the ops/sec of set operation.

Fig. 13 and Fig. 14 display the ops/sec of get operation in canonical workload over two datasets. From these figures, we can say that the bigger number of elements the lower ops/sec of set operation. Although we make an array of double linked list, the number of elements can be large if we choose a large number of elements setting. For every get operation, we have to choose a list from the operation list and traverse it. Accordingly, a bigger number of elements setting increase the overhead of traverse. Merge operations do not affect the get operation, because they hold share lock for each other.

4.3 Impact of Merge Operation

As shown above, the number of elements affects the set and get operation in different ways. We count the number of merge operation triggered as SASS processes more and more requests in canonical workload. In Fig. 15, we can

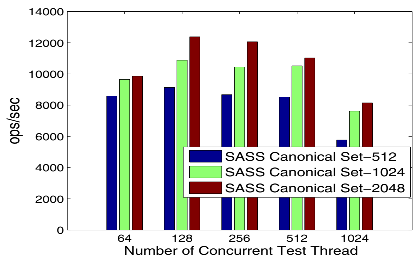

Fig. 11. Set ops/sec over Canonical

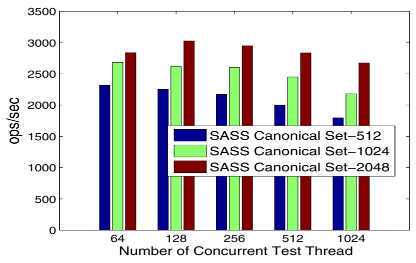

Fig. 12. Set ops/sec over Canonical-L

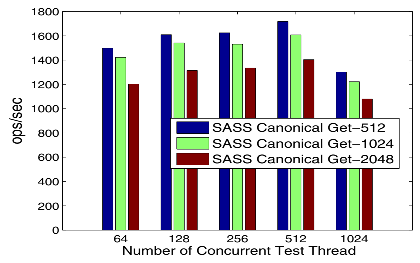

Fig. 13. Get ops/sec over Canonical

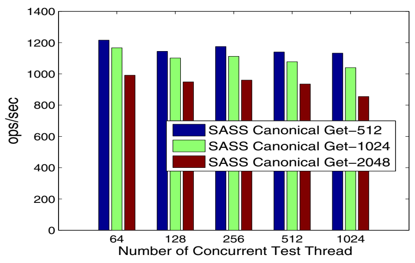

Fig. 14. Get ops/sec over Canonical-L

see that as the data size accumulates gradually, SASS with maximum number of elements setting 512 incurs most merge operations. SASS with maximum number of elements setting 1024 incurs the least merge operations.

For the best case, SASS with maximum number of elements setting 1024, we analyze the merge operations and count the number of merge operations triggered by operation list and the number of merge operations triggered by SSD respectively. From the Fig. 16, we can see that there is no merge operations triggered by SSD until the size of test data reaches up to 200GB. The reason is that the SSD has a 256GB capacity, so it can hold about 200GB operation logs and trigger few merge operations. However, as the size of test data exceeds 400GB, more and more merge operations are triggered by SSD.

We also vary the size of data block from 2MB to 16MB, and appreciate their impact on the throughput performance when the number of test thread is set to 128. We find that a block with the size of 8MB provides the maximum of throughput. We discuss the reason why different block sizes can exhibit different performance improvements. When we select a small block size, the size of SSD-resident index grows rapidly, which increases the cost of maintaining the index on SSD. If we use a larger data block, SFHS can reduce the seek latency of the hard disk, which can improve the I/O performance significantly. However, for a given key, SFHS has to spend more system resource to answer the requested key-value pair in a block.

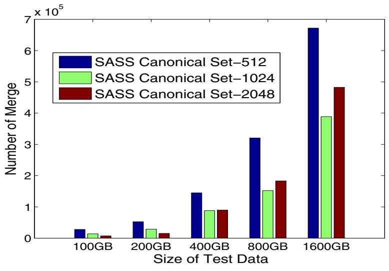

Fig. 15. Number of merges over Canonical

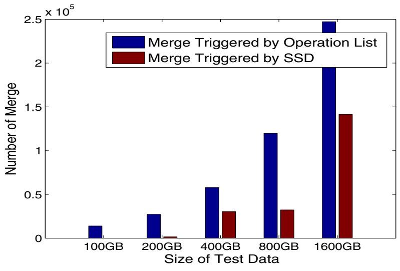

Fig. 16. Number of merges triggered by operation list and SSD

5 Related Work

As a novel storage medium that is totally different from magnetic disk, flash memory is getting more and more attention in recent years. Lots of work emerged to solve different problems. Some work focus on an intrinsic component of the flash, namely flash translation layer *(FTL)*, which is an important firmware in flash-based storage[3,4]. Some work focus on the measurements on flash memory [5,6], and some other work focus on how to adjust the traditional methods in DBMS to take full advantage of the unique characteristics of flash memory [7–9]. We can't cover all those excellent work here, so we just give a brief review for the work related to key-value storage in this section.

FAWN[10] is a cluster architecture for low-power data intensive computing. It uses an array of embedded processors together with small amounts of flash memory to provide efficient power performance. FAWN uses an in-memory hash table to index key-value pairs on flash while SASS adopts hierarchical bloom filter for blocks on hard disk and doubly linked list for operation logs on SSD.

FlashStore[11] is a high throughput persistent key-value store that uses flash memory as a non-volatile cache between RAM and hard disk. It stores the working set of key-value pairs on flash and indexes the key-value pairs by a hash table stored in RAM. FlashStore organizes key-value pairs in a log-structure on flash to obtain faster sequential write performance. So it needs one flash read per key lookup. The hash table stores compact key signatures instead of full keys to reduce RAM usage. FlashStore provides high performance for random query. However, FlashStore employs an in-memory index to record the key-value pair residing in the hard disk, the memory overhead for implementing the index may exceed the available memory when handling the billion-scale keys.

ChunckStash[12] is a key-value store designed for speeding up inline storage deduplication using flash memory. It builds an in-memory hash table to index data chunks stored on flash. And the hash table can help to identify duplicate data. ChunkStash organizes the chunk data in a log-structure on flash to exploit fast sequential writes and needs one flash read per lookup.

SkimpyStash[13] is a RAM space skimpy key-value store on flash-based storage designed for server applications. It uses a hash table directory in RAM to index the key-value pairs stored on flash. To reduce the utility of RAM, SkimpyStash stores most of the pointers that locate each key-value pair on flash. It means that SkimpyStash uses linear chaining to resolve hash table collisions, and stores the link list on flash. SkimpyStash may need multiple flash read for one key lookup. In addition, because the flash memory is more expensive than the hard disk, the cost of using flash memory to handle the large-scale key-value pairs is very huge.

From the related work stated above, we can conclude that most existing key-value store designed for hybrid storage adopt in-memory data structure to index the key-value pairs. However, with 2 billion or more key-value pairs stored on each machine, these designs consumes memory excessively. Furthermore, the design of SASS is based on a slim SSD capacity(256GB) and a chubby hard disk capacity(10TB) while most other designs are based on a comparative SSD capacity with hard disk. By placing the hierarchical bloom filter index on SSD, SASS reduces the memory consumption and answers any query request with one flash memory read for the best case or an extra hard disk read for the worst case.

6 Conclusion

We propose SASS, a key-value store design supporting massive data set and high throughput. Experiment results show that SASS takes full advantage of the flash memory and enjoys excellent read/write throughput. Actually, there are still some interesting problems to be studied and researched. For example, what on earth the role that flash memory should play, using for logging or caching. Furthermore, how to setup a hybrid storage strategy in a distributed system, hybrid on each node or some are flash nodes(only flash memory adopted in the machines) and some others are hard disk nodes(only hard disk adopted in the machines). All these problems are realistic in industry and valuable for researchers.

References

1. Jeremic, N., Mühl, G., Busse, A., Richling, J.: The pitfalls of deploying solid-state drive RAIDs. In: 4th Annual Haifa Experimental Systems Conference, pp. 14:1–14:13. ACM Press, Haifa (2011)
2. Balakrishnan, M., Kadav, A., Prabhakaran, V., Malkhi, D.: Differential RAID: Rethinking RAID for SSD reliability. In: 5th European Conference on Computer Systems, pp. 15–26. ACM Press, Paris (2010)
3. Gupta, A., Kim, Y., Urgaonkar, B.: DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In: 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 229–240. ACM Press, Washington (2009)
4. Lee, S., Shin, D., Kim, Y.J., Kim, J.: Last: locality-aware sector translation for nand flash memory-based storage systems. ACM SIGOPS Operating Systems Review. **42**(6), 36–42 (2008)

5. Bouganim, L., Jnsson, B., Bonnet, P.: uFLIP: Understanding flash IO patterns. In: Online Proceedings of the 4th Biennial Conference on Innovative Data Systems Research, pp. 1–12, Asilomar (2009)
6. Chen, F., Koufaty, D.A., Zhang, X.D.: Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: 11th International Joint Conference on Measurement and Modeling of Computer Systems, pp. 181–192. ACM Press, Seattle (2009)
7. Chen, S.M.: FlashLogging: exploiting flash devices for synchronous logging performance. In: ACM SIGMOD International Conference on Management of Data, pp. 73–86. ACM Press, Rhode Island (2009)
8. Nath, S., Kansal, A.: FlashDB: dynamic self-tuning database for NAND flash. In: 6th International Conference on Information Processing in Sensor Networks, pp. 410–419. ACM Press, Massachusetts (2007)
9. Trirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L., Graefe, G.: Query processing techniques for solid state drives. In: ACM SIGMOD International Conference on Management of Data, pp. 59–72. ACM Press, Rhode Island (2009)
10. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: FAWN: a fast array of wimpy nodes. In: 22nd Symposium on Operating Systems Principles, pp. 1–14. ACM Press, Montana (2009)
11. Debnath, B., Sengupta, S., Li, J.: FlashStore: high throught persistent key-value store. Proceedings of the VLDB Endowmen. **3**(2), 1414–1425 (2010)
12. Debnath, B., Sengupta, S., Li, J.: ChunkStash: speeding up inline storage deduplication using flash memory. In: 2010 USENIX Conference on USENIX Annual Technical Conference, pp. 1–12. USENIX Association, Boston (2010)
13. Debnath, B., Sengupta, S., Li, J.: SkimpyStash: RAM space skimpy key-value store on flash-based storage. In: ACM SIGMOD International Conference on Management of Data, pp. 25–36. ACM Press, Athens (2011)