Trie-based Data Structures for Persistent Key-Value Store on SSD

CMPS 278 Course Project

Yanan Xie
University of California, Santa Cruz
yaxie@ucsc.edu

ABSTRACT

Key-value store has became an important component for many scale-out businesses to implement high throughput applications, including social networks, online retail, and content management. Both academics and industrials have started paying more attention on studying and designing new systems for key-value store. As the price per unit storage goes down for high performance persistent storage hardwares, flash storage devices like SSD(Solid State Drive) are becoming increasingly common for enterprise-class data storage. In this paper, we describe 3 different trie-based data structures for persistent key-value store on SSD. We implement all of those data structures on top of our in-house key-value database BitBase and compare the space efficiency and throughput on different key distributions.

1 INTRODUCTION

There are mainly 3 ways to achieve key-value storage: tree, skip list and hash table. Although hash table is considered to be far more efficient than other ways in terms of throughput, it has two major problems. One is that hash table may take a very big space in the very beginning. And the space is highly related to the key range space which is often hard to estimate. It can be even harder to resize the hashing space once we run out of space. So we can hardly see any real-world applications of hash table-based key-value databases in industrial. Both tree-based approaches and skip list-based approaches have the virtue of space efficiency which means the size of database grows as the actual data grows. In this paper, we mainly address tree-based data structures for key-value store. In particular, we study 3 kinds of trie-based data structures for key-value store, namely classical trie, linked trie and hash trie.

Common tree-based indexing techniques like AVL, Red-Black Tree, B-Tree often rely on the order of keys to achieve high performance on looking up and sorting. Unlike traditional database, sorting is not necessary for key-value database and comparison between strings can be far more expensive than between pair of other data types like integer, double, date time, timestamp. Trie-based data structures offer rapid access to strings while maintaining reasonable worst-case performance[2]. They are successful in many applications like text compression[3], dictionary management[1]. There were many studies on building persistent key-value database with those data structures on devices with low random access performance like HDD (Hard Disk Drive). We like to try some of those data structures on new device like SSD and make some certain modification to adapt the performance of SSD.

In this paper, we will first introduce how we implement the basic components in our in-house database BitBase including low-level data operation and buffer manager. Then we will describe 3 different kinds of trie-based data structures for building a key-value database. As the last part of this paper, evaluations on those 3 different data structures will be made.

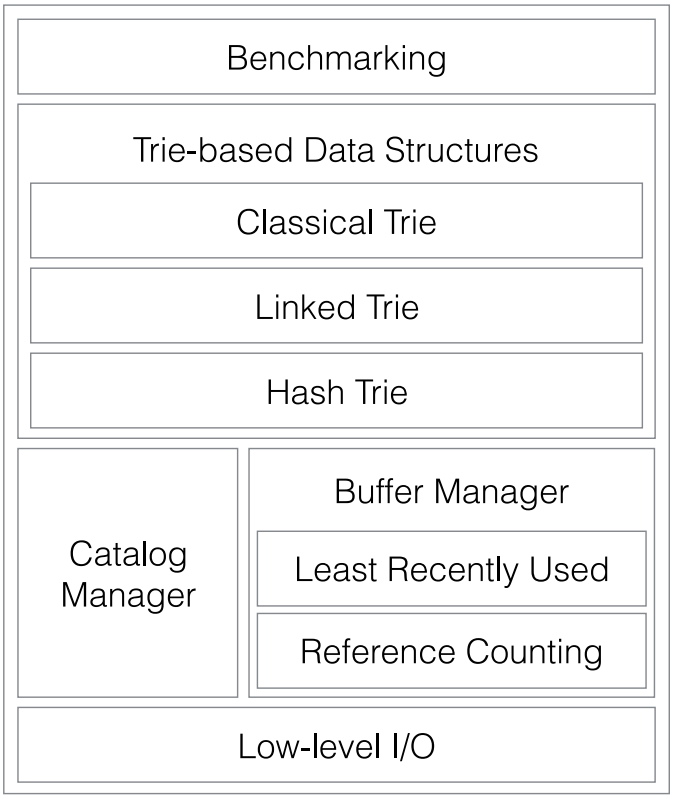

Figure 1: Architecture of BitBase system.

2 ARCHITECTURE

In this section, we first describe the overall architecture of our system. And then, we will introduce two key components in our system before we get into next section.

To eliminate the influence from different platforms, we choose GNU-like operating system as the base system to build our key-value database, run experiments and draw conclusions. Also for simplification, we follow a common design principle of key-value database of employing a single thread model to avoid high implementation difficulty and complex techniques and data structures. And we only support set, get, cas and inc atomic operations on integer data type.

As shown in Fig. 1, the very top of our system is benchmarking which simulates requests and measuring performance. For convenience, we ignore a key component in key-value database system, network interface, in this architecture, although we have built one based on libevent, our evaluations will not depend on the network interface. The component below is our trie-based data structures. We will introduce those data structures in the next section. The very bottom of our system is low-level I/O. Both catalog manager and buffer manager are built on top of low-level I/O. We will get into details of those two components in the rest of this section.

2.1 Low-level I/O

It has been long argued that persistent I/O(Input/Output) is the bottleneck to most data system. Many designs in this area were based on the properties of the actual I/O abilities that exist. Popular database management systems all try to maximize the actual I/O performance in various ways. So the first thing we want to do is to figure out a good way to read and write data into a disk.

In most high-level applications, data is stored in some text format(e.g., XML, JSON). Stream-level I/O is more flexible and usually more convenient in those cases. programmers generally use the descriptor-level(low-level) functions only when necessary[8]. These are some of the usual reasons:

(1) For reading binary files in large chunks.
(2) For reading an entire file into core before parsing it.
(3) To perform operations other than data transfer, which can only be done with a descriptor. (You can use fileno to get the descriptor corresponding to a stream.)
(4) To pass descriptors to a child process. (The child can create its own stream to use a descriptor that it inherits, but cannot inherit a stream directly.)[8]

In case of building high performance database management system where data files are binary and large, low-level functions are necessary.

To achieve high performance random access to persistent storage, GNU provides functions like lseek, read, write, pread, and pwrite. Those functions allow applications read a certain size of data from some specific offset and write a piece of data into some position in a file. File is described by using an integer called file descriptor. in all low-level functions.

Although the previous mentioned functions have been far faster than traditional steam-level functions, we do have some other alternatives. On modern operating systems, it is possible to mmap (pronounced “em-map”) a file to a region of memory as shown in Fig. 2. When this is done, the file can be accessed just like an array in the program. This is more efficient than read or write, as only the regions of the file that a program actually accesses are loaded. Accesses to not-yet-loaded parts of the mmapped region are handled in the same way as swapped out pages. Memory mapping only works on entire pages of memory. Thus, addresses for mapping must be page-aligned, and length values will be rounded up. Typical page size on modern systems is 4096 bytes.

Besides common file descriptor functions, there are mainly 3 functions we need to use to achieve memory-mapped I/O. The first function is mmap which map a certain region in the persistent device into memory and returns a memory address which points

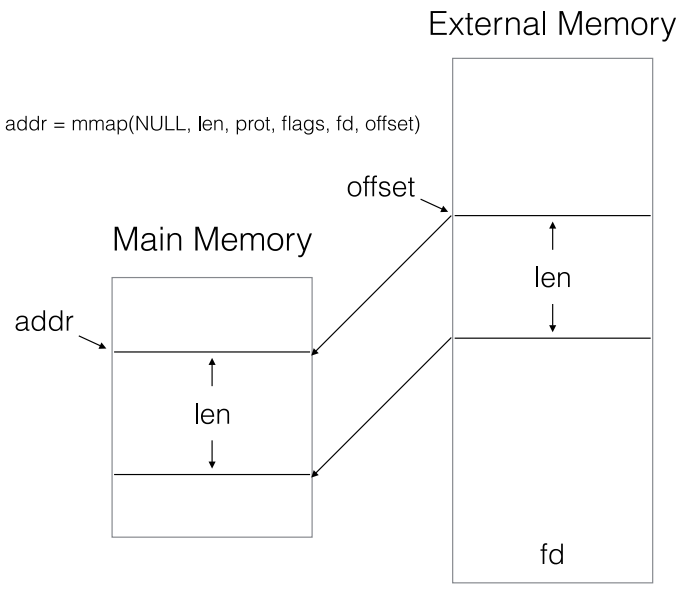

Figure 2: Usage of mmap function

to the mapped memory. With the memory address, we can read and write data like operating an object in memory. Since mapping uses memory addresses, it does have limitation on the total size of mapped data. So we will need another function called munmap to unload mapped memory and release the memory address for other uses. Since mmap only works for mapping regions within the range of some file, we are going to need a function called truncate to resize the file when we want to append data to the data file. However, frequently resizing data file can be very expensive. In BitBase, we resize data file every 2^{10} pages(i.e., $2^{10} * 4096$ bytes).

2.2 Cache algorithm

Caching, a fundamental technique in modern computing, has been widely used in storage systems, database, web servers, middleware, processors, operating systems, file systems. Although we have located a way to achieve high performance read and write with memory-mapped functions, frequently mapping and un-mapping persistent storage can be expensive and is still, for sure, the bottleneck in the whole system. Thus, a cache algorithm that fits our design is needed.

Cache algorithm has been well studied[4, 5]. Different cache algorithms base on different assumption on the distribution of data accessing. Factors that contribute to the distribution can be the database itself and the type of workload. LRU(Least Recently Used) has been proved to be one of most powerful and common cache algorithms in databases and storage systems. There are still a lot of recent researches working on designing LRU-based cache algorithm to better adapt their particular systems, workloads orand other needs.

However, since cache algorithm is not the main part of key-value database we want to address. In our system, we only exploit a minimum LRU that meets our requirements. The basic idea of LRU is to discard the least recently used items first when cache is

full. In order to doing that, the algorithm requires keeping track what was used when to make sure the algorithm always discards the least recently used. Traditional implementation of LRU cache algorithm require keeping "age bits" for in-cache items. Each time the application access an item from via the cache, the algorithm updates those "age bits".

In our implementation we use two main-memory data structures, list and unordered map, from STL(Standard Template Library) to achieve O(1) overhead LRU cache algorithm. The list, being double linked, stores the cached items(pages) in the order of age. The item in the head of the list is the least recently used one. Since we do not actually need to know the exact age of each item, all we need to know is the relative age order among those cached items. So we can just track their order by appending newly cached item to the end of the list and moving hit item also to the end. Appending an item to a linked list is obviously O(1) in time complexity. We achieve O(1) time complexity moving hit item by using unordered map(with O(1) look up and O(1) erase operations). Unlike other caching scenarios that free memory when discarding an item, we un-map a page when discarding an item.

However, since updating the list and map mentioned above is still expensive if we update them every time we read and write a single byte of the cached item. We allow users of our cache keep a reference to the cached item(mapped page), so that users can read and write multiple times after one access to the cache. Order updating only requires when users gain the reference of the mapped page. One major issue raised by this change is that cache algorithm may um-map a memory that is still being referenced by some user. This issue may cause the whole system crush. In order to solve this problem, our cache algorithm maintains a reference counter for each cached item. Accessing an item increases the corresponding reference count by one. Users are required to manually manage the reference count by calling release function to tell cache algorithm that they no longer reference some mapped memory. Every time the cache if full before inserted an item, the algorithm look at the head of the list. If the reference count of head item is 0, we can discard it. Otherwise, we should move it to the end of the list and look at the first item again until we successfully discard one item.

The reference counting is also page-level. Our experiments show that reference counting does not bring any significance in performance change while making it more flexible to implement other parts that are based on the cache.

3 PERSISTENT TRIE-BASED DATA STRUCTURES

Trie-based data structures offer rapid access to strings while maintaining reasonable worst-case performance[2]. Researchers have proposed various kinds of trie-based data structures for building persistent key-value database[10–12].

In this section, we will describe 3 kinds of persistent trie-based data structures for building key-value database and discuss the main difference between them and some properties of applying them on persistent storage devices. We will start by introduce classical trie data structure followed by linked trie data structure while hash trie data structure will be presented in the last part of the section.

3.1 Classical trie

Trie was first described by De La Briandais in 1959[6] the name of trie was given two years later by Edward Fredkin[7]. In the simplest version of trie data structure, a trie node contains two components: one is the value of the current node while the second one is a dictionary that maps a char into a child pointer. Popular implementation of trie data structure usually use a array of 256 pointers to map an ASCII char to a pointer. In the architecture where a pointer uses 8 bytes, a trie node may spend approximately 8*256 = 2048 bytes to store. This method achieves less trie nodes to traverse, but results in a very low memory efficiency, especially when we want to use external memory where data need to be aligned. In order to store trie data structure on external memory, we use 8 bytes to represent a pointer. 4 bytes are used to represent page id which indicates which page of 4,096 bytes the pointer points to. Another 4 bytes to represent the index of node in the located page in case of a page containing multiple nodes. The original papers of trie argued that we can build a trie of n dimension by use an n dimensional key representation in order to improve memory efficiency where n is far less than 256. This kind of strategy is often called time-space trade off. In the later experiments, we will show the performance of classical trie data structure in terms of throughput and space efficiency with different n.

3.2 Linked trie

It has been shown in many researches that keys are usually generated in some pattern in many applications like social networks, content management systems. This means the distribution of key can be extremely unbalanced. For example, a content management system may uses keys followed by the pattern "n_view_$article_id" to store the number of views on one article. Those keys all share the same prefix "n_view_". In classical trie data structure, we have to either traverse a long path for this prefix with a smaller dimension n or spend a large space to shorten the path with a larger n. Developers will expect a solution that traverse shorter path and uses smaller space.

To attack the problem brought by unbalanced key distribution, we present a linked list-based trie data structure which we call it linked trie. A linked trie node contains a value of 4 bytes, a version of 4 bytes, a next pointer of 8 bytes, k children's pointers and k responding characters. The size of a linked trie node usually takes 16+9*k bytes to store in external memory. Thus, we can store approximately 4,096(16+9*k) nodes in a single page. Fig. 3 shows an example of linked trie where key "HI" maps to value 5 with version 2. This kind of data structure works best in case that all prefix contain less than k sub nodes, since a linked trie node can store at most k children. If some node is full, then we will need use another node pointed by the next pointer to store additional nodes. This data structure may increase comparison needed. However, comparison is considered much cheaper than read and write. We will see the experiments on linked trie node data structure with different k in the next section.

3.3 Hash trie

Another way to solve the key distribution issue is hashing. A hash function is any function that can be used to map data of arbitrary

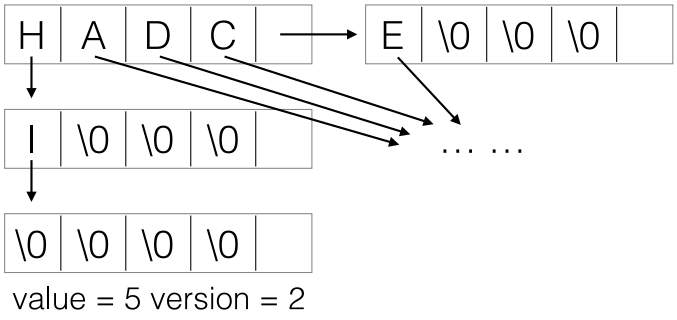

Figure 3: Linked trie data structure with trie_size=4.

size to data of fixed size. A good hashing function is supposed to redistribute original distribution to a more balanced distribution. Common hashing functions include MD5, MD6, SHA-1, SHA-256. Hashing often come with collisions. A collision means two different keys are mapped into the same hash value. Traditional way to resolve this problem is to store the full keys in the stored value. When collision happens, we look up all keys stored under the hash value and operate on the value that matches the target key. Since collision is very rare in real world application when we use a strong hashing function, in this paper, we do not implement the part of collision processing. But we will run test to ensure that the key space is enough to guarantee a low probability of collision.

4 BENCHMARKING

In this seciton, we will introduce 3 different types of workload we are using for measuring the performance of those data structures for presistent key-value store. After that we will evaluate their performance on different sizes of cache. The final part of this section will test the performance stability of those data structures to see if they can provide stable throughput as the data file grows.

All experiments are done on a 2015 Macbook Pro 13 with 2.9 GHz Intel Core i5, 16 GB 1867 MHz DDR3 RAM, 500GB SSD. We run 1,000,000 requests from an empty data file to obtain the following results if there are no further description about number requests processed.

4.1 Workload

Trie-based methods are usually sensetive to the distribution of key. We model the distribution of key in terms of length of key, space(character set) of key and how key varies from time.

We use 3 different way to generate keys. The first one is **intkey(*m*)** which generates integer string keys. Each key is computed by rand() mod *m* where *m* is a parameter controlling the space size of key. The second way is called **editstr(*n*,*m*)** which takes 2 parameters: length of key *n*, and size of character set *m*, to generate string keys. Each time the method replace a character at a random position in the previous key by a randomly picked character in given character set. The last method is called **randstr(*n*,*m*)** where two parameters have the same meanings as in previous method. The only difference between the last two methods is that **randstr(*n*,*m*)** generates a new key randomly from blank each time which is independent to previous key.

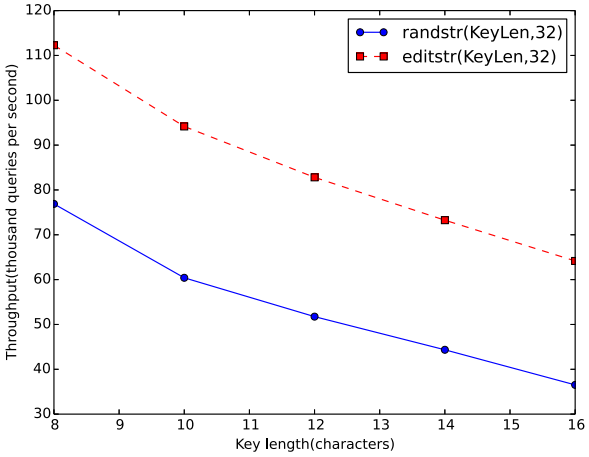

Figure 4: Performance of linked trie on two workloads. cache_size = 65536. trie_size = 12.

Performance is evaluated from two different aspects. One is throughput (QPS, queries per second) while another one is space efficiency (bytes per query).

Fig. 4 shows throughput of linked trie on two different workloads. It makes sense that *editstr* is easier than *randstr*, since keys generated by *editstr* are tend to share prefixes. However, in real world applications, it is pretty common that keys are following some patterns as shown in previous example. The result from Fig. 4 also indicates that linked trie benefits from mutual prefixes.

Fig. 5 shows a slightly different sensetivity to Fig. 4. Classical trie tends to have similar throughput as the length of key increases while linked trie does not.

Result from Fig. 6 is great different from previous results on classical trie and linked trie. Hash trie does not sensetive to the length of key or the way to generate keys. The difference between two ways of key generation is very small.

4.2 Size of node

Due to the requirement of memory alignment, hash trie and classical trie are hard to adapt difference size of node. As shown in Fig. 7, as the size of trie node increases, the space usage becomes less efficient. In the begininng, the increasement of space usage does bring throughput improvement. However, as the size of trie node become greater than 8, the throughput start decreasing while space usage is still going up.

4.3 Cache size

The performance of cache algorithm is highly depended on the specific workload and the way the application employs it. As shown in Fig. 8, both classical trie and hash trie obtain smooth throughput improvement as the size of cache increases. The curve of linked trie is completely different from the rest. Its improvement from increased cache is significantly more intense than the others. And we can see that its improvement become flat after cache size becomes

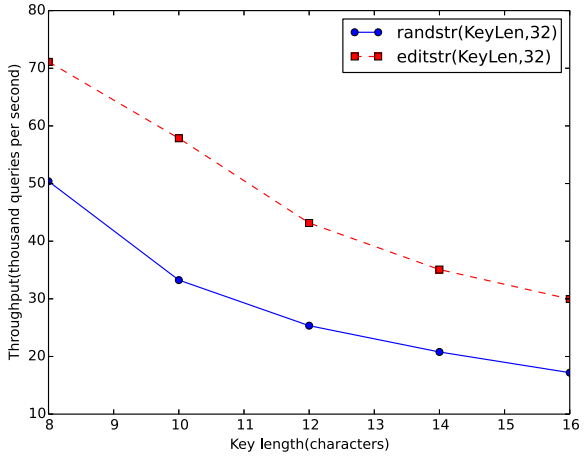

Figure 5: Performance of classical trie on two workloads. cache_size = 65536. trie_size = 16.

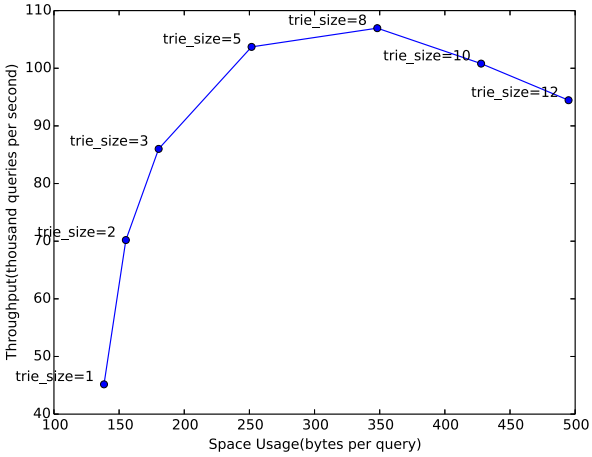

Figure 7: Throughput-space efficiency curve of linked trie with different number of children(trie_size) on int-key(2147483648) workload. cache_size = 65536.

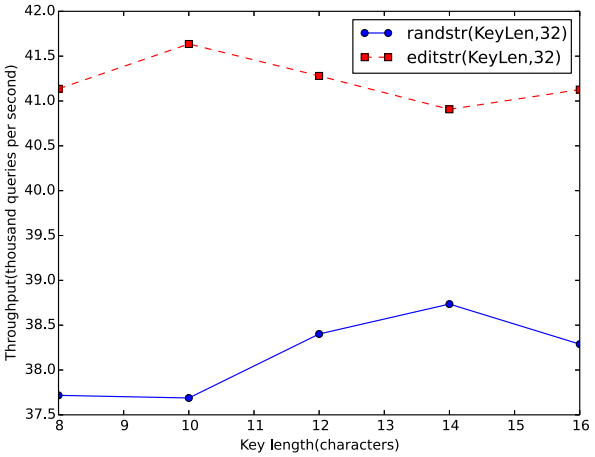

Figure 6: Performance of hash trie on two workloads. cache_size = 65536. trie_size = 16.

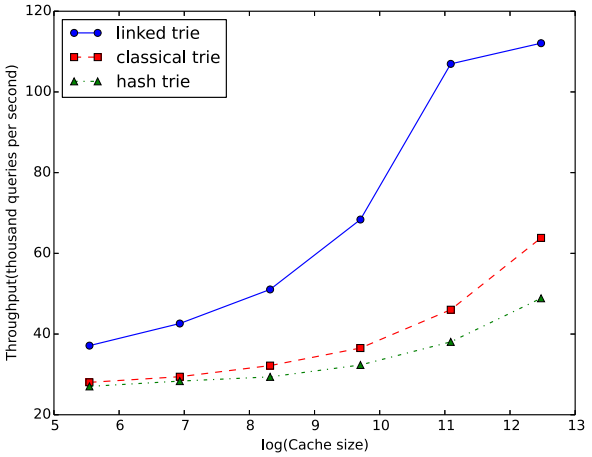

Figure 8: Throughput-cache size curve for 3 different trie-based data structures

greater than 2^{11} pages. The reason may be that linked trie is more workload-aware compared with other two data structures.

4.4 Space efficiency

The size of key space does not only influence the throughput of those data structures, it also bring impact on the size of data file they generate.

Fig. 9 shows the relation between the size of data file and the size of key space for 3 different data structures. The data file of classical trie grows the fastest while the data file of linked trie is the smallest among 3 data structures in the begining. The size of data file storing hash trie does not change as the size of key space changes. Those different features bring different applications to those data structures.

4.5 Compare with Redis

Redis is a popular persistent key-value database that has been widely used in many applications[13]. In addition to the previous experiements, we also test Redis in the same machine. With Redis's official benchmark tool, we get a result of 65,789.48 set requests per second and 68,965.52 per get requests per second with key space = 100,0000 and value size = 4 bytes. This shows that our system has achieved performance comparable to popular persistent key-value databases, although we only support a certain type of data, integer.

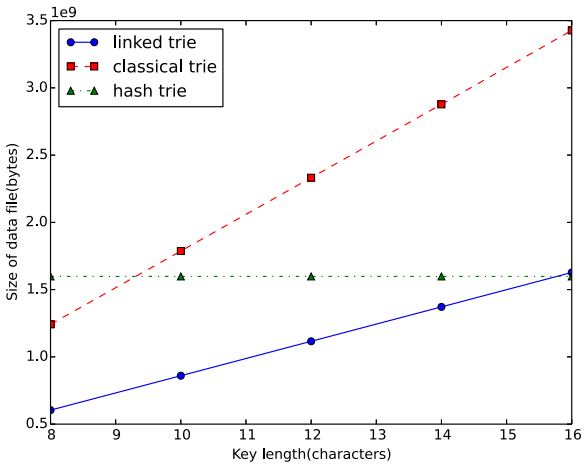

Figure 9: Data file size-key length curve for 3 different trie-based data structures

5 DISCUSSION

Besides throughput and space efficiency, there are many other aspects of key-value database that have not been addressed in this paper but are worthy of study. As the increasing speed of data generation in nowadays, scalability has becoming more and more important to many data management system. A lot of work has been done in scaling existing databases. However, key-value database is easy to scale. One simple solution to scale key-value database is employing a master server that maps requests to other slavers. Unlike other kinds of data management systems, key-value database does not have significant further issues raised by mapping requests.

Another important issue is consistency. Although we do not have ARIES-style log manager[9], we can still achieve consistency by updating version after updating value. If a set operation is interrupted by an error, either the target value has not been updated or target value has been updated but version has not. A get operation on the same key will get an invalid version which indicates the value is also invalid. The invalid version may also cause to restart a CAS(Compare and Swap) operation, thus overwrite the previous interrupted operation.

6 CONCLUSION

In this paper, we implemented a key-value databse engine called BitBase with 3 different trie-based persistent data structures, low-level I/O and a LRU-based buffer manager. We compared the performance of those 3 data structures under different workloads, cache sizes, and trie sizes. Experiements show that different trie-based data structures have different favored workloads and different sensetivities to trie size, key length and cache size. More importantly, our in-house key-value database archives comparable throughput to popular key-value database.

REFERENCES

[1] Jun-Ichi Aoe, Katsushi Morimoto, and Takashi Sato. 1992. An efficient implementation of trie structures. *Software: Practice and Experience* 22, 9 (1992), 695–721.

[2] Nikolas Askitis and Ranjan Sinha. 2007. HAT-trie: a cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*. Australian Computer Society, Inc., 97–105.

[3] TC Bell, JG Cleary, and IH Witten. 1990. Text Compression Prentice Hall. *Englewood Cli s, New Jersey* (1990).

[4] David I Bevan. 1987. Distributed garbage collection using reference counting. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 176–187.

[5] Pei Cao and Sandy Irani. 1997. Cost-aware www proxy caching algorithms.. In *Usenix symposium on internet technologies and systems*, Vol. 12. 193–206.

[6] Rene De La Briandais. 1959. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*. ACM, 295–298.

[7] Edward Fredkin. 1960. Trie memory. *Commun. ACM* 3, 9 (1960), 490–499.

[8] Sandra Loosemore, Richard M Stallman, Andrew Oram, and Roland McGrath. 1993. The GNU C library reference manual. (1993).

[9] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[10] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*. ACM, 23–32.

[11] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots. In *Acm Sigplan Notices*, Vol. 47. ACM, 151–160.

[12] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 71–82.

[13] Jeremy Zawodny. 2009. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine* 79 (2009).