Improving LSM-trie performance by parallel search

Abstract

[Summary] LSM-trie-based Key-Value (KV) store is often used to manage an ultra-large data set in reality by introducing a number of sub-levels at each level, its linear growth pattern can fairly reduce the write amplification in store operations. Although this design is effective for the write operation, the last level holds a large proportion of KV items, leading to the extreme imbalance of data distribution. Therefore, to support efficient read, we need to carefully consider this imbalance. On the other hand, to ensure that acquired data is latest, the LSM-trie needs to search the data set at different levels one by one, and this search method may take a lot of unnecessary time. When the number of items is ultra-large, the random lookup performance may be poor due to the imbalance data distribution. To address this issue, we improve the read performance of the LSM-trie by changing its *serial search* to *parallel search*, using two threads to simultaneously search at the last level and other levels, respectively. Our experiment results show that the read performance of the LSM-trie can be improved up to 98.35% and on average 71.55%.

Improving LSM-trie Performance by Parallel Search

Wen Cheng, Tao Guo, Lingfang Zeng*[1], Yang Wang*[2], Lars Nagel[3], and Tim Süß, André Brinkmann[4]

[1]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Hubei, China
[2]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Guangdong, China
[3]Department of Computer Science, Loughborough University, UK
[4]Department of Computer Science, Johannes Gutenberg-University Mainz, Germany

1 Introduction

Key-Value (KV) store has become an important part of NoSQL databases, and made many important achievements in the field of scientific research and various application areas in recent years. For example, Google's BigTable [10] and LevelDB [12] are often used in web indexing, MongoDB [26] and SkimpyStash [14] in online gaming, FlashStore [13] in data deduplication. Other examples include Dynamo [15] Amazon uses to solve the large amount of redundant data produced in e-commerce, LinkBench [4] and RocksDB [8] Facebook adopts to maintain social networking, and Atlas [18] Baidu leverages to store cloud data.

For I/O-intensive workloads, the KV stores based on Log-Structured Merge-Trees (LSM-trees) [23] have been extensively research and widely deployed [12, 27, 29, 8, 17, 16, 5]. The main advantage of the LSM-trees over other indexing structures (such as B-trees) is that they can deliver high performance for sequential (batch KV pairs) write access patterns on either Solid-State Drives (SSDs) or Hard-Disk Drives (HDDs) [20] by maintaining the ordered keys and values for compaction at different levels in background. Since the same data could be read and written multiple times, the I/O amplification in a typical LSM-tree can reach a factor of $50\times$ or higher throughout its life-time [30, 20].

Both WiscKey [20] and LSM-trie [30] focus on minimizing the I/O amplification and the disk-level improvement [5].WiscKey minimizes the I/O amplification on SSDs by separating keys from values, keeping only the keys in the LSM-tree and the values in a separate log file on SSD devices. With this design, the LSM-Tree generated by WiscKey is much smaller than that of LevelDB when the number of KV key pairs is fixed. Thus, WiscKey can substantially reduce the write amplification. However, this merit is greatly compromised when storing massive small data since the size of values relative to the keys in the proportion is not that large. What more serious is that the large number of pointers (to value) occupy most of the storage space.

For the storage of large volume of small data, LSM-trie is advocated, which exploits a *trie* structure and a "sub-level" linear growth pattern in compaction to greatly alleviate its write amplification problem. LSM-trie exploits only linear growth pattern at the fifth level (i.e., *Level4*), most of the KV-items are stored at this last level for a large store, especially for many terabytes. However, LSM-trie uses a serial mode to search data from the first four levels (i.e., *Level0-Levle3*) to the last level (i.e., *Level4*), as such there is still a lot of rooms for further improvements in terms of random read performance. We show a more detailed analysis later (see Section 2.2).

To improve the random read performance of the LSM-trie, we change its search process by introducing *parallel search* among *Level0-Level3* and *Level4*. Our experiments show that the random read performance can be improved up to 98.35% (71.55% on average), compared with the original LSM-trie. As Section 4.1 shows, the improved LSM-trie can obtain better read performance than its original across a wide range of workloads. Nonetheless, this optimization does come at a cost–we serve one read request by using two threads. Although our test results (Section 4.2) demonstrate that the effects are small and acceptable, a large number of intensive reads may consume most of CPU resources. As such, our optimization might be more practical on multi-core platform and beneficial to the applications of LSM-tree-based KV in various latency sensitive services such as those in location-based services (LBS) [25, 22] and online gaming [21], where the efficient processing of spatial-temporal workloads and fast generating of suitable game scenarios are highly desired.

The remainder of the paper is organized as follows: Section 2 overviews LSM-trie, our motivation, and the related work. Section 3 describes the overall design of the improved LSM-trie. Section 4 presents the evaluation results, and Section 5 concludes the paper.

2 Background and Motivation

2.1 LSM-trie

An LSM-tree (Log-Struct Merge-tree) is a persistent structure capable of efficient indexing for Key-Value store with a high rate of inserts and deletes [23]. To this end, it defers and batches data writes into large chunks in memory, fully using the characteristics of higher sequential write bandwidth than random writes on hard disk. When reaching a specified threshold, the data blocks will be written to disk sequentially. Comparing with the traditional B-Tree, the LSM-tree can make greater improvements on data writes. However, there still exist challenges in reducing the write amplification during the LSM-tree compaction [30].

LSM-trie [30] uses trie structure to organize the keys and reduces the write amplification by an order of magnitude for LSM-tree-based KV systems. Since trie is not a balanced structure, the LSM-trie-based KV store uses a SHA1-generated 160 bit hashkey, which determines the location of the KV item, to guarantee a uniform distribution of the KV items at each (sub)-level, regardless of the distribution of the original keys. LSM-trie usually has 5 levels (e.g., *Level0-Level4*). The *Bloom Filter* [7] for *Level0-Level3* are all in memory, while for *Level4*, the Bloom Filter and the data are stored in SSD. All the *Table-Containers* of the same depth in trie constitute a level in the trie structure, which is equivalent to a level in LevelDB. Each container has a pile of *HTables*, a hash-based KV-item organization. A trie level consists of a number of *HTable* piles. All *Htables* at the same position of the piles of a trie level constitute a sub-level (total 8 sub-levels in each level) of the trie, which corresponds to a sub-level in LevelDB. As each KV item is also identified by a binary (i.e., hashkey), its location at a certain level is determined by matching the hashkey's prefix to the identity of a node at that level (see Fig. 1). This matching exhibits a linear (for a new sub-level at each level) or an exponential (for a new level) growth pattern, which avoids the lower level data involved in the compaction operation, so the LSM-trie solves the write amplification problem. Suppose a hashkey of a KV item is 101011100010..., each 3 bits determines a *Table-Container*, then the KV item will be stored in the *Level1* 6th *Table-Container*, according to the initial 3 bits '101'. Similarly, when the data is merged into *Level2*, the KV item will be located in a new *Table-Container*, according to the next 3 bits '011'.

2.2 Motivation

Since the LSM-tree divides all the data into many small data sets, and each random lookup operation has to query the data sets in order, the read speed of the LSM-tree-based KV stores is in general far lower than the write speed. Thus, the random lookup performance will be poor when the number of items are ultra-large. LSM-trie also has the same problem. In order to ensure that the acquired data is the latest

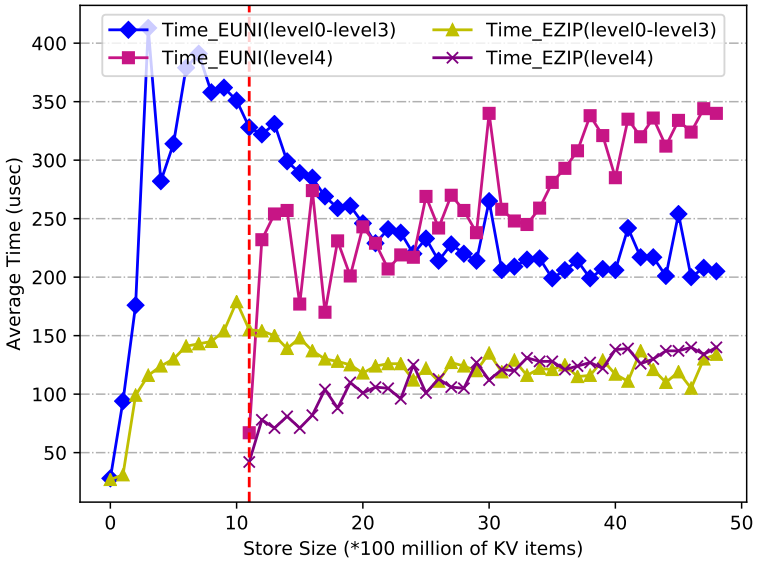

Figure 1: Architecture of **LSM-trie**.

version, LSM-trie needs to search the data set at different levels one by one, which might take a lot of unnecessary time. For example, suppose the target data is located at *Level4*, the search for *Level0-Level3* will spend a lot of unnecessary time. Fig. 2 and 3 show the comparisons between the average time to complete a search at *Level0-Level3*, and the average time to complete a search at *Level4* in two different memory conditions (32GB and 4GB). *EUNI* denotes the data is subject to uniform distribution and *EZIP* to Zipfian distribution, respectively. The value size of the KV items is 100 bytes and the key size is 16 bytes. It is worthwhile to point out that the red dotted line indicates that *Level4* begins to store data when the volume of data of the KV pairs reach 1.2 billion.

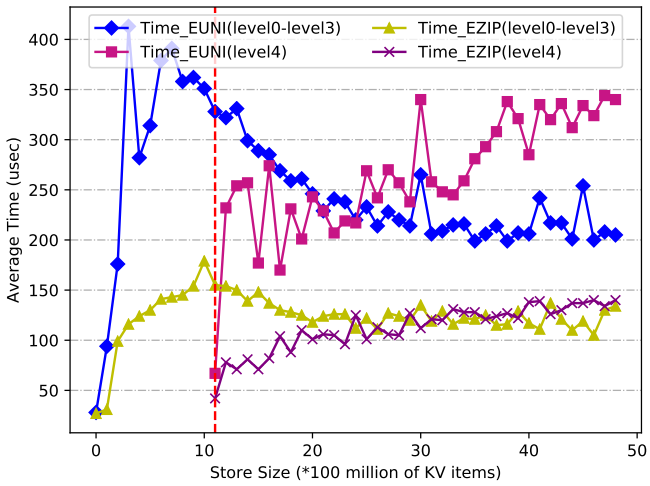

Figure 2: *Level0-Level3* and *Level4* search time with 32GB memory.

As shown in Fig. 2, when the memory size is 32GB, the average time to complete a search at *Level0-Level3* and at *Level4* with the *Zipfian* distribution is *similar*, and when the data is subject to the *uniform* distribution, the average time to complete a *Level4* search is higher than that at *Level0-Level3*.

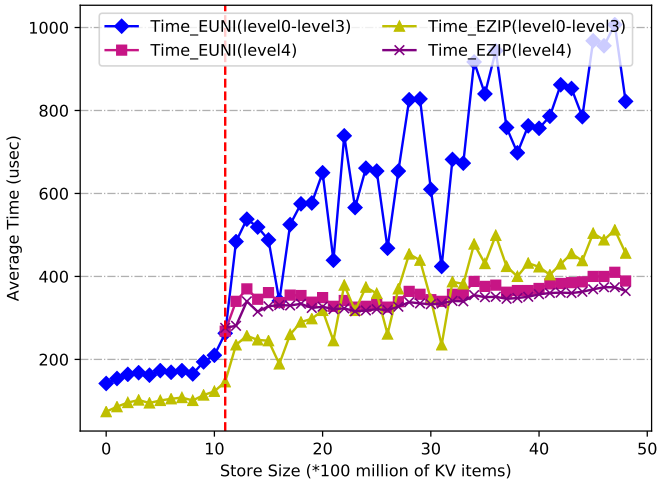

Figure 3: $Level0$-$Level3$ and $Level4$ search time with 4GB memory.

As shown in Fig. 3, when the memory size is 4GB, the average time to complete a search at *Level0-Level3* is longer than the other three cases (these three cases spend almost similar time).

As shown from the above two test results, whether the memory size is 4GB or 32GB, when the data is subject to the Zipfian distribution, they spend almost the same amount of time to complete a search at *Level0-Level3* and at *Level4*. In contrast, when the data is uniformly distributed, in the case of 32GB memory, the search time at *Level4* is slightly longer than that at *Level0-Level3*, while in the case of 4GB memory, the search time for *Level0-Level3* is much longer than that at *Level4*. The main reason of this phenomenon is the Bloom Filters for *Level0-Level3* are centrally stored in memory, so when the memory is sufficient, it requires only an SSD read operation (for the target); otherwise, when the Bloom Filters for *Level0-Level3* cannot be completely stored in the memory, the data query time is longer jitter phenomenon (given the data is subject to uniform distribution). But the lookup operation at *Level4* is always two SSD reads, so its time is relatively stable.

The prototyped LSM-trie has just 5 levels, and uses 32MB *HTables* with an amplification factor of 8. For the first four levels, the LSM-trie uses both linear and exponential growth patterns, that is, suppose each level consists of 8 sub-levels, then 256MB for *Level0*, 2,048MB for *Level1*, 16,384MB for *Level2*, and 131,072MB for *Level3*, the total data size of the first four levels is around 146.5GB. However, when the amount of data at *Level4* reaches 1TB with one sub-level (total 8 such sub-levels) that has a capacity of 128GB, it can reach up to 10TB with 80 such sub-levels, according to the linear growth pattern of LSM-trie. So most of the KV items (10TB) are stored at the LSM-trie *Level4* (in contrast to 146.5GB at the first four levels). The large probability of search operation falls at *Level4* due to the serious imbalance in the data distribution. The LSM-trie serial searches (every search operation starts at the first four levels and ends at the fifth level) spend unnecessary time on searching *Level0-Level3* if the KV item is not at *Level0-Level3*.

2.3 Related Work

In order to improve the performance of random read, previous research focuses on how to build the index to facilitate the data access. For example, LevelDB [12] adds index and Bloom Filter. By adding the index and combining with the ordered arrangement of datasets, LevelDB can quickly locate a block, to which the lookup key belongs, avoiding the accesses to the dataset one by one. And the use of Bloom Filter can judge if a query key exists in a block on the basis of the located block. SILT [19] optimizes the read performance by reducing the amount of metadata, it reduces the number of levels, and uses only two levels of the structure to store the data, and thus, has fairly high read performance. However, the

Table 1: Related studies on the performance of random reads

Method	Storage system	Advantages	Disadvantages
Internal index	LevelDB [12]	Reduce disk access times	Internal metadata is verbose
Hash index	FAWN [3], FlashStore [13], BufferHash [2], SkimpyStash [14]	Reduce disk access times	Require a large amount of memory space
fractaltree index	TokuDB [9] [6]	Fast index merging	Require a large amount of memory space
HB+-trie	ForestDB [1]	Small overhead of storage space	Only index longer key
Reduce the amount of metadata	SILT [19], bLSM [27]	High speed & Small space cost	Serious write amplification
Using hardware features	WiscKey [20], DIDACache [28], FloDB [5]	Make full use of the underlying hardware features	Require hardware technical support

number of levels is too small, making it particularly serious for write amplification problems. In contrast, WiscKey [20] adopts another idea on this issue, which uses the multi-channel features of SSD to speed up the LSM-Tree lookups by means of parallel lookups.

There are many other types of KV storage systems to improve the read performance [3, 13, 14, 2]. FAWN [3] stores the KV pairs in a log file in SSD, which can only be appended, and the system maintains a hash table in memory to enhance the speed of data lookups. FlashStore [13] and SkimpyStash [14] also adopt the the idea of FAWN, but both perform some optimization on the hash table in memory. Specifically, FlashStore leverages the Cuckoo [24] hash for memory managements while SkimpyStash transfers part of index in memory to SSD. Unlike the discussed systems, BufferHash [2] uses multiple memory hash tables and establishes Bloom Filter for each hash table to facilitate the searches. TokuDB [9, 6] is based on the *fractaltree* index, which updates the data in the internal nodes, the data in node is unordered, and the system requires to maintain the large index in memory.

Again, SILT [19] is a partial in-memory key-value store system, which is organized by a three-tier indexing model: *LogStore*, *HashStore*, and *Sorted-Store*. The average size of the memory footprints per document increases from LogStore to HashStore, and then to SortedStore, while the capacity increases in reverse order. When the number of entries in HashStore exceeds a pre-defined threshold, the entries are merged into SortedStore, a trie-based structure. Note that it is impossible to partially update an entry in SortedStore, thus the merge operation from HashStore to SortedStore always involves a full revision of the entire index. bLSM [27] improves the overall read performance of LSM-trees by adopting Bloom filters and suggests a new compaction scheduler called *Spring* and *Gear*, to bound the write latency without impacting the overall throughput. ForestDB [1] uses *HB+-trie* to index long keys, effectively improving the read performance and reducing the space overhead. Table 1 summarizes the above mentioned systems and methods, as well as their advantages and disadvantages.

In contrast to the previous studies, we improved the read performance of the LSM-trie by changing its serial search to parallel search, employing two threads to simultaneously search at the last level and other levels, respectively.

3 Design and Implementation

As discussed in Section 2.2, the searches at *Level0-Level3* and at *Level4* roughly spent the same amount of time for random lookup operation. In this section, we propose to use *parallel search* in the LSM-trie to improve the random lookup performance.

3.1 Design space

We propose a solution that creates two threads for a random lookup, one for *Level0-Level3* and the other for *Level4*. The rationale behind this choice that we only use two threads, instead of three or more, is that the amount of search time spent at *Level0* to *Level3* and at *Level4* are almost the same, and in the meanwhile, the probability of target data at *Level4* is fairly high. On the other hand, the proposed parallel search is sequential, which is different from the multi-threads random search in the original LSM-trie where each thread services a separate random search request, rather than the same request as in our case. As such, compared to two threads, using more search threads will only waste more resources without obvious performance gains. To validate our choice, we first analyze why multiple threads are not effective and then show two search threads are sufficient for one request.

Multiple threads are ineffective for one request: Intuitively, we can employ multiple threads, one for each level, to speedup the search process serve a request. As the test results shown in Fig. 2 and 3 in Section 2.2, the searches at *Level0-Level3* and at *Level4* spend roughly the same amount of time in a random lookup operation. If the target data is at *Level4*, LSM-trie needs two SSD read operations, one is for the Bloom Filter at *Level4*, and the other for the target data page. Otherwise, if the target data is at *Level0-Level3* and the memory size is assumed to fully meet the demands of all the Bloom Filters at *Level0-Level3*, the completion of a random search requires only one SSD read operation. Since LSM-trie has only 5 levels, it can use up to 5 lookup threads to serve a random lookup request, that is, each thread will service a lookup for a corresponding level. The Bloom Filters for *Level0-Level3* are centrally stored in memory, there are three solutions if each level needs a thread for searching at the first four levels:

Solution1: When the target key for a random search request is at *Level0-Level3*, only one thread executes a SSD read operation, the other three (search) threads, if available, execute the judge operation of Bloom Filter in memory. In the memory, a judge operation of Bloom Filter is shorter than that of read operation in SSD, so the three threads will be finished in a quite short period of time. Compared to only one thread search for *Level0-Level3*, the above mentioned method cannot reduce the search time, rather it could spend more time due to the preemption of system resources. This solution is low cost-effective.

Solution2: When the target key for a random search request is at *Level4*, because the log-appending in the LSM-trie, a search thread for *Level0-Level3* only needs 4 judge operations for the Bloom Filters for *Level0-Level3* in memory. Fig. 2 and Fig. 3 in Section 2.2 show the time spent in the 4 judge operations is almost the same to that of a read operation in SSD. However, the system needs two read operations in SSD for *Level4*, required a long time, so in this case, a single thread is sufficient for the search at *Level0-Level3*, which needs less system resources.

Solution3: Finally, when the target of a random lookup request does not exist, a read operation in SSD is also required to read the Bloom Filter at *Level4*, so this case is the same as Solution2.

Two search threads is sufficient for one request: Considering the above three solutions, it is clear that using only one thread for *Level0-Level3* is most reasonable. In the following, we analyze why the proposed system uses another search thread for *Level4*. There are three cases.

Case1: When the target of a random search request is at *Level0-Level3*, the search thread for *Level0-Level3* needs to judge for the Bloom Filter in memory, and then executes a read operation in SSD, while the search thread for *Level4* requires only a read operation from SSD to obtain the Bloom Filter for *Level4*. The former is longer than the latter in the search time. In this case, the search thread for

Level4 does not elongate the search time, but in the meantime, the system does not gain any performance optimization.

Case2: When the target of random search request is made at *Level4*, the required time could be significantly longer than that for *Level0-Level3* because the search thread for Level4 needs two read operations from the SSD. The proposed system adopts one search thread for *Level4*, which skips the searches for *Level0-Level3*, saving the time of 4 judges from the Bloom Filter for *Level0-Level3*. Fig. 2 and 3 in Section 2.2 show that we save time about a read operation from the SSD. In this case, the method can greatly improve the system performance for random reads.

Case3: When the target of random lookup request does not exist, a read operation in SSD is also required to read the Bloom Filter for *Level4*, so the performance will be improved in the same way as Case2. Given the above three cases, the proposed system adopts an additional search thread for *Level4*, which improves the system's random lookup performance, and with the increasing amount of data at *Level4*, the performance optimization effects should be more obvious.

3.2 Parallel search algorithms

Based on the above description, in this section, we present our improved search process (compared with the LSM-trie) as shown in Fig. 4. In the search, there are two search threads, one for the first four levels (*Level0-Level3*) and the other for the last level (*Level4*).

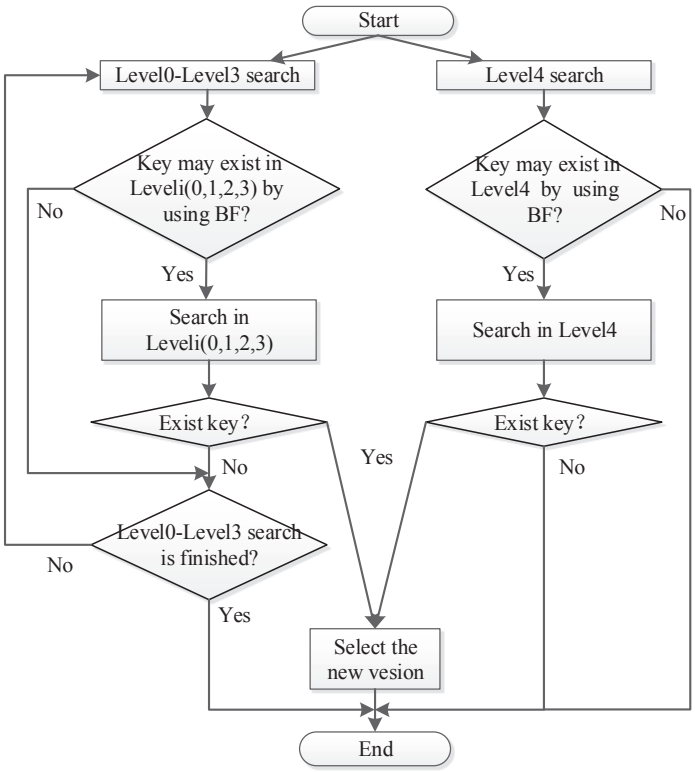

Figure 4: *Parallel search* process.

LSM-trie adopts a log-appending mode, it does not support in-place-update. Specifically, the data update is directly performed as new data is written—the old version of the data is replaced by a new version during the compact operation. Therefore, there may be multiple KV item versions of the same key in the LSM-trie, and the latest version always exists at the lower level. This results in that the

system cannot end up with the search thread for *Level0-Level3* when the search thread for *Level4* finishes quickly. We thus design three different choices for the parallel searches at *Level0-Level3* and at *Level4* as they usually finish at different times.

The lookup result is returned directly when the search for *Level0-Level3* is finished with a non-null result prior to the search for *Level4*. This is because the version of data at *Level0-Level3* is always newer than that at *Level4*, regardless of whether the result from *Leve4* is *NULL* or not.

If the search result from *Level0-Level3* is first arrived, but having a value of *NULL*, the system needs to wait for the search result from *Level4*, and determines the required return value, according to the results of the search thread.

When the search thread at *Level4* is completed first, the system has to wait for the completion of the search thread for *Level0-Level3*, regardless of whether or not the result of the search thread for *Level4* is *NULL*. As the data version at *Level0-Level3* is newer than that at *Level4*, there may be a new version of the data in *Level0-Level3*.

In our implementation, we create the two search threads from the main thread, and the property of the two searches is set to *PTHREAD_CREATE_JOINABLE* as shown in Algorithm1, where *func1* is a function pointer pointing to the *read_th1* function, which is designed to search at *Level0-Level3*, and *func2* is a function pointer to a function *read_th2*, which is responsible for the search for *Level4*.

$pthread\_t\ ths[2];$
$pthread\_attr\_t\ attr;$
$pthread\_attr\_init(\&attr);$
$pthread\_attr\_setdetachstate(\&attr, \textit{PTHREAD\_CREATE\_JOINABLE});$
$pthread\_create(\&(ths[1]),\ \&attr,\ \textit{func2},\ arg);$
$pthread\_create(\&(ths[0]),\ \&attr,\ \textit{func1},\ arg);$
for $(uint64\_t\ j \leftarrow 0;\ j < 2;\ j++)$ **do**
  $pthread\_join(ths[j],\ \textit{NULL});$
end for

Algorithm 1: conc_fork_reduce

The proposed system serves a random read request in the following steps:

Step1: thread *read_th1* generates a random search request *search_key*, wake-ups thread *read_th2* and calls function *db_lookup_Level0_Level3* to lookup level by level from *Level0* to *Level3*;

Step2: after thread *read_th2* is awakened, call *db_lookup_Level4* to lookup at *Level4*;

Step3: thread *read_th1* completes its lookup, and then returns result *KV1*;

Step4: thread *read_th2* finishes its lookup, and then returns result *KV2*;

Step5: to determine whether the test time is used up: If *yes* then jump out of the random lookup test, otherwise, jump to (Step1) to perform the next search.

The search algorithms of *read_th1()* for *Level0-Level3* and *read_th2()* for *Level4* are shown in Algorithm2 and Algorithm3, respectively, where $db\_lookup\_Level0\_Level3(search\_key)$ and $db\_lookup\_Level4(search\_key)$ are two internal subroutines to lookup *search_key* from *Level0-Level3* and *Level4*, according to the procedure implemented in LSM-Trie.

Algorithm2 read_th1

repeat
$search\_key \leftarrow generate\_search\_key()$;
$read\_th2\_start \leftarrow true$;
$pthread\_mutex\_lock(\&mutex)$;
$pthread\_cond\_signal(\&cond)$;
$pthread\_mutex\_unlock(\&mutex)$;
KV_L03 $\leftarrow db\_lookup\_Level0\_Level3(search\_key)$;
$t \leftarrow debug\_time\_usec()$;
$search\_num++$;
until $t \geq time\_finish$
$pthread\_exit(NULL)$;

Algorithm3 read_th2

repeat
$pthread\_mutex\_lock(\&mutex)$;
while $read\_th2\_start$ **do**
$pthread\_cond\_wait(\&cond, \&mutex)$;
end while
$pthread\_mutex\_unlock(\&mutex)$;
KV_L4 $\leftarrow db\_lookup\_Level4(search\_key)$;
$t \leftarrow debug\_time\_usec()$;
until $t \geq time\_finish$
$pthread\_exit(NULL)$;

4 Experiments and Evaluation

In this section, we present and analyze our experimental results on the random lookups of the improved LSM-trie. We also make some additional testing to demonstrate that our optimization does not bring serious adverse effects.

We used Yahoo's YCSB [11] benchmark suite to generate read and write requests. The average value size of the KV items is 100 bytes and the key size is 64 bits. In order to facilitate the description, we used "*LSM-trie-opt*" and "*LSM-trie-ori*" to represent the improved LSM-trie system and the original LSM-trie, respectively, in the rest of the section. As with the original LSM-trie experiments, we still used the keys that are randomly generated in a uniform form as it represents the least locality and minimal overwrites in the workloads. For the access pattern (i.e., read/write), we tested the uniform (*EUNI*) and Zipfian key distribution (*EZIP*), respectively. The software and hardware configurations are shown in Table 2:

Table 2: Software and hardware configurations

CPU	Intel Xeon E5-2620
Memory	DDR3 4/32GB
Operating system	Ubuntu 16.04.1
Kernel version	Linux-3.6.11
Solid-state disk	Intel SSDSC2BP480G4

4.1 Random read performance

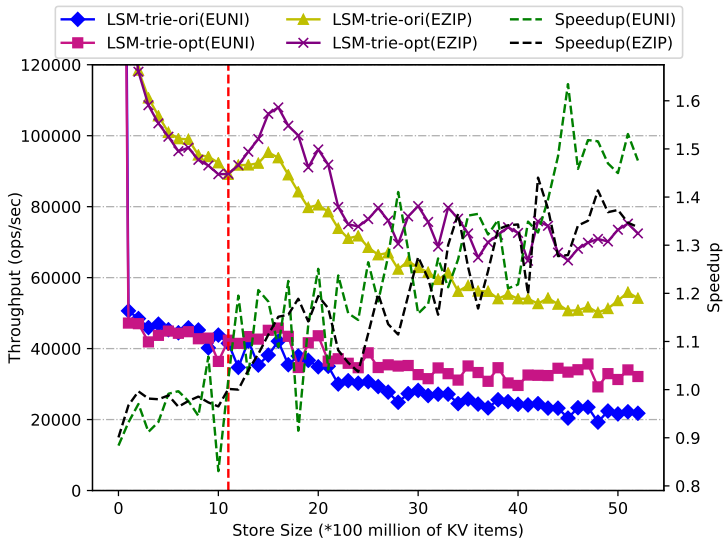

Figure 5: Random lookup performance with 4GB memory.

The proposed system adopts the parallel search to avoid the longer search latency from *Level0-Level3* to *Level4*. In these random read tests for both LSM-trie-opt and LSM-trie-ori, the data sets are subject to the uniform distribution (EUNI) and the Zipfian distribution (EZIP), respectively. During these random read tests, a random read test is launched for every added 20 million items and recorded the success (to return result) times in 30 seconds, and then calculate the average value. In Fig. 5, the red dotted line

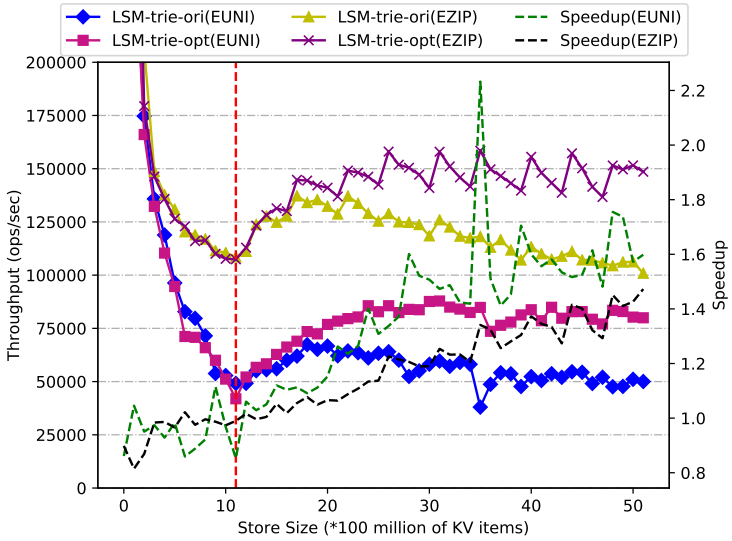

Figure 6: Random lookup performance with 32GB memory.

represents the beginning amount of 1.2 billion KV items, and when that amount is reached, *Level4* begins to store the data. According to the proposed parallel search, LSM-trie-opt outperforms LSM-trie-ori in random reads for both distributions.

Fig. 5 shows the comparison results between the performance of random reads by the LSM-trie-opt and by the LSM-trie-ori when the memory is 4GB. As shown in the figure, the random read performance decreases sharply at the earlier stage when the data are uniform distribution, the reason is that memory size is too small to store all the *Level0-Level3* Bloom Filters. Also, for these random reads, a large number of pages are swapped between disk and memory, causing jittering overhead. The read performance is relatively stable after the rapid decline. And moreover, after 1.2 billion items, the random read performance of LSM-trie-opt has been improved over LSM-trie-ori, and this performance is improved gradually after 2 billion items.

In comparison, when the data are subject to the Zipfian distribution, in the early stage, the random read performance of LSM-trie-ori and LSM-trie-opt do not exhibit a sharp decline as in the uniform test. The main reason is part of front and back data are the same for the Zipfian distribution, these search data are confined to a small portion of the data, as such the number of page replacement between disk and memory with respect to the uniform distribution is greatly reduced. At the same time, the test shows the LSM-trie-opt performance improvement relative to the uniform distribution is obvious, but the performance is not stable, the random read performance after optimization are volatile, mainly due to the background compaction operations, which consume a large amount of CPU and IO resources.

As shown in Fig. 6, in the earlier period, two kinds of data distribution show a downward trend. For uniform distribution, the declining degree of the random read performance is significantly lower than that of 4GB memory, which confirms our earlier analysis about the sharp decline of the random read performance for 4GB memory. In our experiments, 1.2 billion items can be looked as a fluctuation threshold. Before this fluctuation threshold, the test results of the two data distributions show a downward trend and reached a minimum at the fluctuation threshold, and the performance of LSM-trie-opt is not improved relative to LSM-trie-ori. The reason is that at this fluctuation threshold, the amount of data in the Level0-Level3 and the memory associated Bloom Filter are saturated. And after this fluctuation threshold, Level4 begin to store data, the thread responsible for the search at Level4 begin to work, and the optimization is obvious, and the random read performance is improved up to 98.35% (71.55% in average), compared with the original LSM-trie. Similarly, when the test data obeys the Zipfian distribution, the random read performance of LSM-trie-opt is not smooth, and the reason is the same as the case of

4GB memory, which is caused by the compaction operation in the background.

In summary, the proposed parallel search greatly improves the random lookup performance of LSM-trie regardless of the memory size. Apart from the above measured throughput, we also tested the latency of operations. In Section 2.2, Fig. 2 and 3 use the *average latency* metric. But here, we adopted another metric, high latency searches are measured at the 95th, 99th, and 99.9th percentiles, respectively (i.e., the observed latency is worse than 95%, 99%, or 99.9% of all the latencies), and we tried to show the worst random read performance.

To this end, we list the 95th percentile for the different distributions in Fig. 7 and 8. Fig. 7 shows the time required with 4GB memory when 95% of the data searches are completed. No matter how the data is distributed, either uniform distribution or Zipfian distribution, the LSM-trie-opt needs shorter time than the LSM-trie-ori for the data searching when 1.2 billion items are reached. When the data items reach 2 billions, the random search time required for the LSM-trie-opt is more obviously shorter than that of the LSM-trie-ori. Moreover, the performance of Zipfian distribution is more obvious than that of uniform distribution in data searching. Also, Fig. 7 shows both the LSM-trie-opt and the LSM-trie-ori have larger fluctuations of search time for the two data distributions. This phenomenon is mainly due to the limited memory space. If the random read is executed simultaneously with the compaction operations in background at the same time, the memory limitation would obviously become a constraint for the read performance.

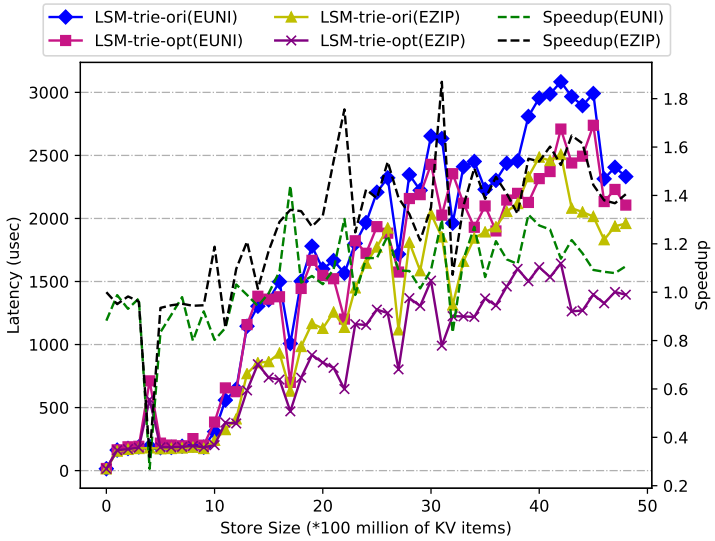

Figure 7: Read latencies with 4GB memory (95th percentile).

In Fig. 8, when the number of items is around 0.5 billion, the (95%) search completion time of both the LSM-trie-opt and the LSM-trie-ori for the two kinds of data distributions are longer than that of the fluctuation threshold, the main reason is the compaction operations at the background, that is, *Level2* data are merged to *Level3*, which affects the random read performance. This phenomenon also confirms the results of the sequential read tests in Section 2.2, where the performance of sequential read for 0.5 billion items is lower than that of 1 billion items. After the number of items reaches 2.5 billions, the search time required for the LSM-trie-opt is significantly lower than that of the LSM-trie-ori. This is also consistent with the test results shown in Fig. 6. Fig. 8 also reveals that the time fluctuation degree is much lower because, compared to Fig. 7, the 32GB memory space is sufficient for this test scenario. And as storage capacity continues to increase, the time required to search is further shortened.

Also, we tested the random read latency of 99th percentile and 99.9th percentile, respectively. Table 3 briefly describes the test results when the memory is 32GB and storage capacity is 5 billions. In

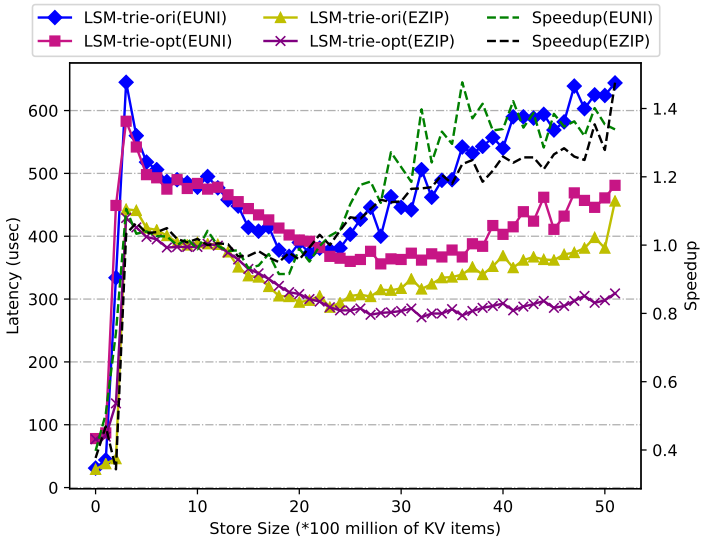

Figure 8: Read latencies with 32GB memory (95th percentile).

summary, these experiments show that the improved LSM-trie ensures a consistently optimized and low-latency performance.

Table 3: Random read latency

	99% (EUNI)	99% (EZIP)	99.9% (EUNI)	99.9% (EZIP)
LSM-trie-opt	644us	460us	1667us	655us
LSM-trie-ori	803us	588us	2298us	941us
Speedup	1.25	1.28	1.38	1.44

4.2 Impact on CPU Workload

We used multi-threads to improve the read performance of the LSM-trie, so it brings additional workloads to both CPU and I/O. To understand their performance effects on the CPU and I/O workloads, we showed the related experimental results in Table 4. When testing the I/O workloads, the worst case should be taken into account, namely all the data of *Level0-Level4* stored in SSD, rather than the separated data at *Level0-Level3* and *Level4*.

In the evaluation, we tested the LSM-trie-opt and the LSM-trie-ori systems with real-time monitoring by using *iostat* command. Table 4 lists the experimental results of 40 time points, including the average values. In order to have the accurate experimental results on the I/O read workloads, the experimental results should completely avoid the impact of writes on the I/O workloads, so the data of *w/s* and *wKB/s* options have to be approached to zero or even equal to zero. As with the I/O case, the experimental results on CPU workloads are listed for 40 time points, these results are the average value of samples. In the test, 32 thread lookups are performed, each of 2 threads serving as a group. Not surprisingly, both the I/O workloads and the CPU workloads are acceptable with the introduction of multi-threads in the process of read operations.

In Table 4, *r/s* is the number of read requests issued to device per second while *avgqu-sz* is the average queue length of the requests issued to device. Obviously, the smaller the value of *avgqu-sz*, the

Table 4: Experimental results of different I/O workloads

	iostat -d -x -k 2 > IO_test.txt					
Items	r/s	avgqu-sz	await	r_await	svctm	%util
Results	26483	5.88	0.232ms	0.232ms	0.0364ms	89.8

better the performance is. In this test, the value is 5.88, indicating that the I/O requests are processed quickly. *await* is the average time for the I/O requests issued to the device to be served, which includes the time spent by the requests in the queue and the time spent to service the requests. If the value of *await* is greater than 5ms, the performance of the whole system could be noticely low. In our test, the value is less than 5ms, indicating extremely fast response time. *r_await* is the average time for the I/O read requests and *svctm* is the average service time for the I/O requests issued to device. In our experiments, the value of *svctm* approaches to the value of *await*, implying there is almost no waiting overhead and good SSD performance. *%util* is the percentage of CPU time during which the I/O requests are issued to device (bandwidth utilization for the device). This format indicates the *busyness* level of SSD, whose saturation occurs when this value is close to 100%. Our test results reveal that the SSD is not fully running in the experiments.

Table 5: Experimental results of CPU workloads

	mpstat 2 > CPU_test.txt				
Items	%usr	%sys	%iowait	%irq	%idle
Results	11.23	8.81	25.88	1.33	52.75

Apart from the above I/O workload tests, we also tested the workloads of CPU using *mpstat* command. This test is made to show that the parallel search scheme does not consume lots of the CPU resources, which implies that the CPU resources are unlikely a performance bottleneck of the system.

The experiments are run on a server with two 6-core processors. As with the previous experiments, we selected 40 time points and calculated their average value in the test whose results are shown in Table 5. The parameters in the table indicate the percentage of CPU occupied over a period of time where *%iowait* and *%idle* shows the percentage of time that the CPUs are idle during which the system has or does not have the outstanding SSD I/O requests. In our test, the value of *%idle* is 52.75, indicating that the parallel lookups do not take up lots of the CPU resources.

The test results of the above two groups of tests and analysis show that for the read process, the change from the original single thread processing to the improved 2-thread processing does not take up serious I/O and CPU resources, and the parallel search scheme does not occupy lots of I/O and CPU resources, preventing them from becoming the system performance bottleneck.

5 Conclusion

Key-Value store has become a fundamental platform for the storage of big data. The linear and exponential growth patterns of LSM-trie reduces the write amplification, but introduces a huge number of sub-levels at the last level, so the last level holds a large proportion of KV items, leading to imbalance for read efficiency. To address this problem, we focused squarely on the improvements of LSM-trie for random read operations by using a parallel search in two threads, We observed that the improved LSM-trie achieves significantly faster throughput, compared to the original LSM-trie over most workloads.

Since it uses hashed key to guarantee a uniform distribution of KV items, LSM-trie can enable distinct key ranges in compaction, but it fails to support range scan. On the other hand, given that a trie is not a typical balanced structure, LSM-trie could be unnecessarily skewed under specific key

patterns. To address these issues, we plan to propose a new key organization scheme, instead of using the hashedkey, so that the range scan is possible.

Acknowledgments

This work was sponsored in part by the Key-Area Research and Development Program of Guangdong Province (2020B010164002, 2019B010137002), National Natural Science Foundation of China (NSFC) under grants 61472153, 61672513, 61572377, and also in part by the National Science Foundation of Hubei Province (2015CFB192).

References

[1] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers*, 65(3):902–915, 2016.

[2] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high-performance dataintensive networked systems. Proc. of the 7th Symposium on Networked Systems Design and Implementation (NSDI), San Jose, California, April 2010.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of Wimpy nodes. Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP), pages 1–14, Oct. 2009.

[4] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. Proc. of the ACM International Conference on Management of Data (SIGMOD), pages 1185–1196, New York, New York, USA, June 2013. ACM.

[5] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking memory in persistent key-value stores. Proc. of the Twelfth European Conference on Computer Systems (EuroSys), pages 80–94, 2017.

[6] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. Proc. of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 81–92, 2007.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] D. Borthakur. Under the hood: Building and open-sourcing RocksDB. `https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920`, Nov. 2013.

[9] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. Proc. of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 859–860, 2000.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI), November 2006.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. Proc. of the 1st ACM Symposium on Cloud Computing, pages 143–154, NY, USA, 2010.

[12] J. Dean and S. Ghemawat. LevelDB: A fast and lightweight key/value database library bygoogle. `https://code.google.com/p/leveldb/`, 2013.

[13] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. Proc. of the 36th International Conference on Very Large Databases (VLDB), 2010.

[14] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 25–36, June 2011.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. Proc. of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 205–220, Stevenson, Washington, USA, October 2007. ACM.

[16] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. Proc. of the Tenth European Conference on Computer Systems (EuroSys), April 2015.

[17] HyperLevelDB. A fork of leveldb intended to meet the needs of hyperdex while remaining compatible with leveldb. `https://github.com/rescrv/HyperLevelDB`, 2014.

[18] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidus key-value storage system for cloud data. Proc. of the 31st International Conference on Massive Storage Systems and Technology (MSST), Santa Clara, California, USA, May 2015.

[19] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP), pages 1–13, Cascais, Portugal, October 2011. ACM.

[20] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. Proc. of the 14th USENIX Conference on File and Storage Technologies (FAST), pages 133–148, 2016.

[21] L. Luo, H. Yin, W. Cai, J. Zhong, and M. Lees. Design and evaluation of a data-driven scenario generation framework for game-based training. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(3):213–226, 2017.

[22] P. Memarzia, M. Patrou, M. M. Alam, S. Ray, V. C. Bhavsar, and K. B. Kent. Toward efficient processing of spatio-temporal workloads in a distributed in-memory system. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*, pages 118–127, 2019.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[24] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[25] M. Patrou, M. M. Alam, P. Memarzia, S. Ray, V. C. Bhavsar, K. B. Kent, and G. W. Dueck. Distil: A distributed in-memory data processing system for location-based services. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '18, pages 496–499, New York, NY, USA, 11 2018. Association for Computing Machinery.

[26] E. Plugge, T. Hawkins, and P. Membrey. *The definitive guide to MongoDB: The noSQL database for cloud and desktop computing*. Apress Berkely, CA, USA, 2010.

[27] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. Proc. of the 2012 ACM International Conference on Management of Data (SIGMOD), pages 217–228, May 2012.

[28] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Didacache: A deep integration of device and application for flash based key-value caching. Proc. of the 14th USENIX Conference on File and Storage Technologies (FAST), pages 391–405, 2017.

[29] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-Trees. Proc. of the 11th USENIX Symposium on File and Storage Technologies(FAST), 2013.

[30] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. Proc. of the USENIX Annual Technical Conference (ATC), 2015.