Nylon: Efficient Scheduler for Heterogeneous Storage System

Shean Kim, Heejin Yoon
UNIST (Ulsan National Institute of Science and Technology)

***Abstract*— Key value store is main component of cloud computing to process big data efficiently. Performance of key value store is highly related to one of storage material. Therefore, performance of storage material is important. While storage material is developed rapidly, people expect that it will bring better performance of key value store. However, this development gives 2 side effects. First, user can be burden with high cost, because cost of developed storage material is too expensive. Second, fast storage material doesn't bring better performance. Design of most key value store is based on Log-Structured Merge tree[1]. To maintain this structure, there are internal operations such as compaction and flush. Since these operations consider only slow storage, CPU can be sacrificed. However, in case of fast storage, too high overhead of CPU can be bound overall performance of key value store. This situation is called "bottleneck shifting" which means that reason of bottleneck is changed from disk to CPU. Therefore, we have to consider how to design key value store which is cost effective and can solve described problems.**

We suggest new scheduling policy of key value store, not modifying structure. In particular, we focus on internal operations in key value store and re-schedule them. SILK[2] which is highly related with our work pointed out rough scheduling of conventional key value store. However, it didn't consider heterogeneous storage system. Therefore, we explore observations which is shown with developed storage material and extend previous work to use heterogeneous storage system. By experiment, write throughput is improved 11.61%, 11.49% over RocksDB[3], Autotune RocksDB[4]. Other performance and limitations will be described in evaluation section.

I. INTRODUCTION

Key value store is database which supports NoSQL. Required data type of this database is only key and value. Because of this simplicity, there are many advantages compared with conventional RDBMS database. Key value store supports high scalability, superior performance and flexibility.

To design key value store, Log structured merge tree(in short, LSM)[1] is commonly used. LSM based key value store considers only slow storage like HDD. Because HDD is very slow and supports low bandwidth, LSM sacrifices CPU by giving more overhead. For example, compaction can merge sort multiple unsorted files at one. It can reduce disk random IO compared to B-tree. Instead, merge sort overhead is connected to CPU, so CPU overhead is increased. However, increased CPU overhead can be ignored because of too poor performance of disk. Therefore, LSM based key value store is adopted to many companies and used to manage production data.

Recently, storage material is rapidly developed. In past, HDD was main storage material. However, SATA SSD is commonly used and even NVMe SSD which performance is better than SATA SSD and supports higher bandwidth is also used. Moreover, persistent memory, which is byte addressable and supports both persistence and memory-like performance, is released. The advent of new storage materials causes bottleneck shifting problem of key value store. Traditional key value store sacrifices CPU and hidden limitation of disk. These efforts make disk under-utilized and give too high overhead to CPU. It causes bottleneck shifting, which means main reason of bottleneck is changed disk to CPU with fast storage. To solve this problem, related researches are actively proceeded.

During solving problems which is related to use fast storage material, some researches focused on heterogeneous storage system. Although fast storage such as persistent memory and NVMe SSD can guarantee better performance, they are too expensive. As we know, budget is not infinite. Therefore, we have to suggest new technique which can support high performance and keep budget limitation.

Most of related works focused on modifying structure of key value store, not scheduling policy. However, Latency critical application is sensitive to client tail latency. Therefore, we have to consider new scheduling policy to reduce client tail latency. SILK[2] which is our related work pointed out indifference about tail latency of key value store. And it suggest new scheduling policy to minimize tail latency. However, SILK considered only single storage, not heterogeneous storage system.

Technique with heterogeneous storage system is necessary, because its performance is higher than one with slow storage. Moreover, it is cost effective. Therefore, we explore behavior of conventional key value store with heterogeneous storage system. And we increase overall performance and try to solve bottleneck shifting problem by re-scheduling internal operations which maintain key value store.

To summarize, there are 2 main contributions.

- We explored CPU and disk utilization with traditional key value store using heterogeneous storage system. And we found that bottleneck shifting problem exists.
- Nylon suggests new scheduling policy and achieve high performance of key value store with heterogeneous storage system.

II. BACKGROUND

A. LSM based key value store

Log structured merge tree[1] is representative structure of key value store. When LSM was designed, performance

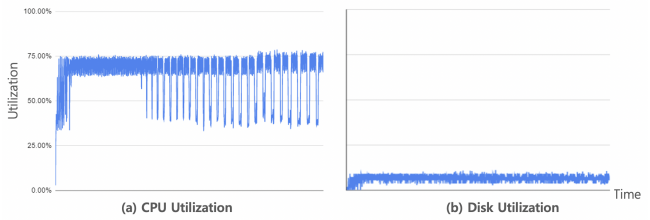


Fig. 1. Bottleneck shifting is serious with faster storage.

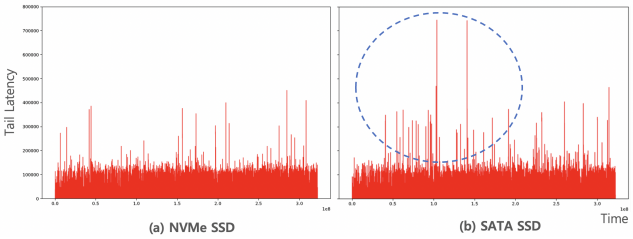


Fig. 2. With faster storage, Tail latency fluctuation is still remained.

of storage material is not that good. Therefore, it tried to minimize disk overhead. Instead, CPU was sacrificed to manage data.

LSM is composed of 2 component, memtable and SSTable. First component, memtable is used as buffer. Storage material is not byte addressable and too slow compared with memory. In other words, storing each data to storage can cause too high IO overhead. Therefore, data is accumulated in memtable until its size is exceed specific threshold and it will be flushed to disk. Flushed memtable is renamed as SSTable and it will be located in specific level. Fresh SSTable lives in upper level and obsolete one lives in lower level. Read operation to disk will find data from upper level to lower one. Since lower level stores obsolete data, it will be returned if data is found in upper level.

Except for Level-0, files of every level aren't overlapped. In other words, duplicated key is not allowed and every keys are sorted. To keep this rule, LSM based key value store uses particular operation, compaction. Compaction sorts files which is located between lower level and upper level. And the candidate files have overlapped range. As capacity of upper level is lower than one of lower level, LSM based key value store has to make room for upper level by frequent compaction.

Flush which moves files from memory to disk has to be done frequently, because memory capacity is also limited. Although these operations are progressed in background, overall performance is still influenced. Moreover, wrong scheduling policy can make client tail latency longer because of limited capacity of each component. Therefore, proper scheduling policy for LSM based key value store is necessary.

B. Heterogeneous storage

As times goes, various storage material is appeared. HDD is the slowest one, but it is very cheap. In contrast, SSD, especially NVMe SSD, is very fast but expensive. Even though block devices are developed rapidly, emergence of persistent memory(in short, PM) gives opportunity to extend applications which use block storage material. PM, which is unlike SSD and HDD, is byte addressable and persistent. Its performance is almost same with performance of DRAM, but non-volatile.

By IDC, worldwide data size will increased to 175 zettabytes in 2025, which is 161% of current data size. Therefore, we have to focus on how to store and manage this increasing data. However, by using fast storage, storing data is too expensive. In contrast, by using cheaper storage, its performance will be poor. Therefore, we have to consider both cost and performance. Its answer could be heterogeneous storage system which can guarantee good performance with reasonable cost.

III. MOTIVATION

Before we start project, we want to check whether problems which is pointed out from previous works are still existed in recent version of RocksDB. Version of RocksDB is 6.20.3.

To evaluate it, we use 2 core Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz, Samsung 970 Pro 512GB NVMe SSD, Samsung 860 Evo 512GB SATA SSD and 48GB DRAM. And db_bench which is microbenchmark of RocksDB is used as benchmark. Dataset size is 30GB, each data size is 108B which is common size of key value item.

Utilization is calculated result of dstat divided by maximum value. Maximum disk bandwidth is estimated as FIO. And maximum CPU utilization calculated number of total thread divided by number of total available cores. We used default number of total threads, 3.

A. Bottleneck shifting

Although LSM based key value store tries to improve performance by sacrificing CPU and relieving overhead of disk, overall performance of key value store depends on disk performance. However, using fast storage material makes main bottleneck changed to CPU. We evaluate the most recent version of RocksDB with NVMe SSD. Figure 1 shows over-utilized CPU and under-utilized NVMe SSD. In other words, recent version of RocksDB is still suffered from bottleneck shifting problem.

Therefore, scheduling internal operations as compaction which can give CPU overhead has to be researched. And by using this technique, we have to reduce CPU utilization and increase disk utilization.

B. Tail latency

For latency critical application, client tail latency is very important. However, research which focused on this perspective is only SILK. SILK, which is our related work, adjusted scheduling of internal operations and tried to reduce client

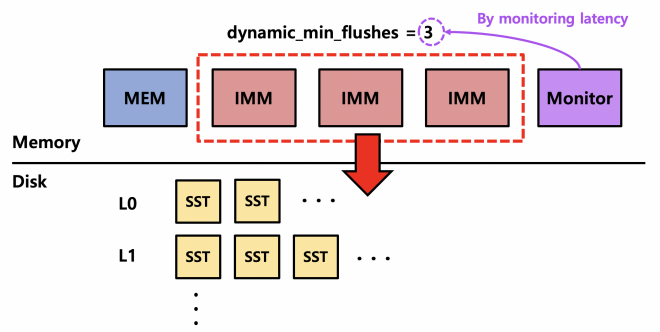


Fig. 3. Dynamic flush

tail latency. However, the research only focused on single storage material. Moreover, it wasn't interested in using fast storage material and disk under-utilization.

We estimate tail latency of RocksDB with NVMe SSD and SATA SSD. As shown in figure 2, using slow storage material makes tail latency fluctuation more serious. However, fast storage material isn't answer, because tail latency fluctuation still remained. Therefore, solving tail latency fluctuation and reducing tail latency is necessary.

IV. DESIGN

This section will explain 2 techniques of Nylon. First one is dynamic flush, which adjusts number of flushed memtable dynamically. This design is intended to solve disk under-utilization. Second one is new priority policy, which is modified priority policy to extend heterogeneous storage system and achieve better performance.

A. Dynamic flush

Traditional key value store allows only 1 memtable to be flushed. However, we think there are 2 problems of this policy. First, it cannot fully utilize disk. In other words, it is better to increase number of flushed memtable and disk utilization. Second, if only single memtable is flushed at once, memory overhead will be increased because memory has to store multiple memtables.

However, too large number of flushed memtable can cause L0-L1 compaction overhead. Unlike other level, L0 can store files which range could be overlapped. Therefore, candidate of L0-L1 compaction contains every files of L0. In other words, large number of files in L0 can increase compaction overhead. And if we allow too large number of flushed memtable, L0 can be full very quickly and it makes number of compaction candidate increased. Therefore, we have to find appropriate number of flushed memtable at once.

We want to set this number dynamically considering performance. Performance is latency in here, and it is estimated during runtime. If latency is longer than expected, it means that L0-L1 compaction brings overhead. Under this case, number of flushed memtable will be reduced. In contrast, number of flushed memtable will be larger for max utilization of disk. In implementation, default number of flushed memtable is 4.

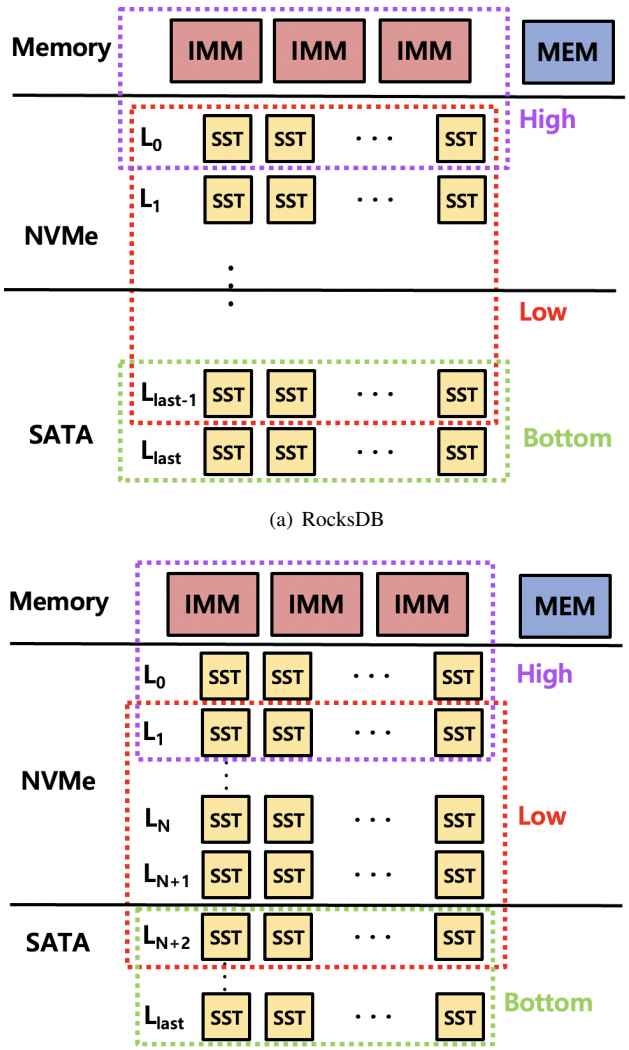


Fig. 4. Priority policy of RocksDB and Nylon

B. New priority policy

Unlike LevelDB and HyperLevelDB, recent version of RocksDB make internal operations work in background. And their priority are differentiated as shown in figure 4. Flush has the highest priority, compaction has middle priority and bottom level compaction has bottom priority (Flush > compaction > bottom level compaction). And their priority is reflected in scheduling policy.

However, we assumed heterogeneous storage system in default. Therefore, we adjust priority of each internal operations.

First, L0-L1 compaction has same priority with flush, so it will have the highest priority. Since candidate of L0-L1 compaction is larger than other level compaction, its overhead is significantly high. If this compaction is delayed, upper operations such as client operation and flush are also delayed. Therefore, we don't treat every compaction same. Priority of L0-L1 compaction is higher than others and it will be occurred more frequently to avoid the delay.

TABLE I

SYSTEM CONFIGURATION

	Description
CPU	Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz (32 cores)
DRAM	48GB
SATA SSD	Samsung V-NAND SSD 860 EVO (512GB)
NVMe SSD	Samsung V-NAND SSD 970 Pro (512GB)
OS	Linux Ubuntu 18.10 LTS (64bit) kernel v5.4.91

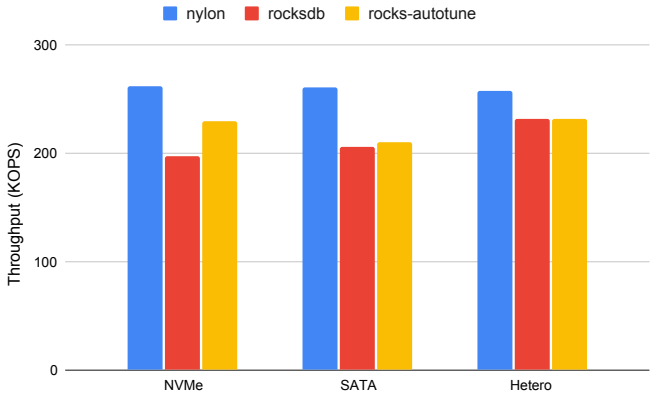


Fig. 5. Write throughput compared with RocksDB, Autotune RocksDB

Second, traditional RocksDB gives bottom priority to bottom level compaction. Unlike this, Nylon gives bottom priority to compaction which level is stored in lower(slower) storage material. The reason why compaction between lower level has to be done is that read amplification. If compaction isn't occurred at all, there are too many files and overlapped ranges between levels. This will bring serious read amplification. However, most data distribution is almost skewed, not uniform. By using this feature, most of access is concentrated to upper level. Therefore, we can focus on upper level compaction, not every level. Moreover, lower(slower) storage material supports poor performance and low bandwidth. Therefore, we don't need to treat them same with upper level compaction. This design makes compaction between upper level could be done more frequently.

V. EVALUATION

A. Experimental setup

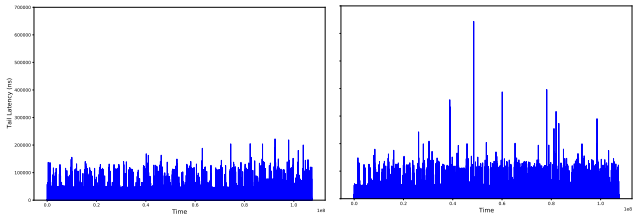
Nylon is implemented on RocksDB 6.20.3. And we compared Nylon with RocksDB and Autotune RocksDB. RocksDB uses rate limiter to limit bandwidth which can be used for internal operations and avoid latency spikes. However, setting parameter for rate limiter is too hard to general users. Therefore, RocksDB provide autotune option to adjust the limit of background IO bandwidth dynamically. And we will call RocksDB with this option as Autotune RocksDB.

To evaluate Nylon, we use 32 core Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz, Samsung 970 Pro 512GB NVMe SSD, Samsung 860 Evo 512GB SATA SSD and 48GB DRAM. And db_bench which is microbenchmark of RocksDB is used as benchmark. Dataset size is 20GB, each

TABLE II

AVERAGE OF 90TH AND 99TH PERCENTILE TAIL LATENCY

	90th(ns)	99th(ns)
Nylon	321.98	746.63
RocksDB	339.05	782.13



(a) Nylon (b) RocksDB

Fig. 6. 99th percentile tail latency. The x-axis is time sequence and y-axis is tail latency in nanoseconds.

data size is 108B which is common size of key value item. Except for described parameters, we used default parameter.

B. Write performance

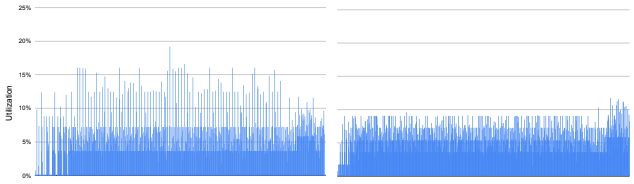
To evaluate overall write performance, we compared RocksDB and Autotune RocksDB. As shown in figure 5, write throughput of Nylon is 11.61%, 11.49% higher than RocksDB and Autotune RocksDB. Main reason of improved performance is priority adjustment. Giving higher priority to L0-L1 compaction can help upper level to make room and it prevents delay of operations such as client operation and flush. Moreover, number of flushed memtable is increased and it is adjusted considering performance. Therefore, it can use more disk utilization and help to improve overall write throughput.

C. Tail latency

We evaluate 90th and 99th percentile tail latency of Nylon and RocksDB and calculate average value of each tail latency in table II. 90th and 99th percentile tail latency of Nylon is improved 5%, 4.5% over RocksDB. Figure 6 shows 99th percentile of tail latency. Followed by this figure, tail latency of Nylon is less fluctuated than one of RocksDB.

D. Disk utilization

We tried to solve disk under-utilization by using dynamic flush. And we evaluated disk utilization of Nylon and RocksDB with heterogeneous storage system. As shown in figure 7, sometimes, Nylon uses more disk bandwidth. Increased number of flushed memtable can maximize utilization of disk. However, actual average value is almost same with one of RocksDB. We suggest the reason is Nylon won't use lower storage material. Because compaction between level of lower storage material is not that frequent, bandwidth of lower storage material could be decreased. The more detailed analysis will be remained as future work.



(a) Nylon (b) RocksDB

Fig. 7. Disk utilization. The x-axis is time and the y-axis is disk utilization as a percentage. For storage, only the results of Hetero using SATA SSD and NVMe SSD are shown, but the graph shows the same trend when SATA SSD or NVMe SSD is used alone.

E. Limitation

Although Nylon has better write performance, there are some limitations. The limitations are remained as future work.

1) Read performance: Lazy compaction of Nylon due to adjusted priority can cause increased read amplifications in lower level. In particular, files which are stored in lower storage material has low priority. Therefore, possibility of compaction between low level is very low. In other words, access to data which is located in lower level could be delayed as shown in figure 8. Moreover, used dataset follows uniform distribution, so it makes more frequent access to lower level. Therefore, compaction between lower level is also necessary and it requires more elaborate scheduling policy than Nylon.

2) CPU Utilization: Nylon tries to reduce CPU utilization and increase disk utilization. However, we couldn't reduce CPU utilization as shown in figure 9. Instead, CPU utilization is more fluctuated than conventional RocksDB. This means overhead of L0-L1 is increased even though priority of L0-L1 compaction is significantly high. Therefore, criteria of setting number of flushed memtable has to be re-considered. For example, CPU utilization will be estimated with very light-weight way during runtime. And it will be used to find proper number of flushed memtable to maximize disk utilization and reduce L0-L1 compaction overhead.

VI. RELATED WORKS

A. Bottleneck shifting

Fast storage is not the reason of bottleneck, instead, CPU is the main reason. There are researches about considering the bottleneck shifting and solving this problem. Representative research is KVell[5]. This research emphasized that compaction which is necessary operation in LSM based key value store is the main reason of CPU overhead. And it pointed out that modern storage material is significantly fast. Therefore, LSM is not proper structure for key value store with modern storage material. They applied B+ tree to key value store and it solves bottleneck shifting by reducing CPU overhead. Moreover, its performance is better than other state-of-art key value store design.

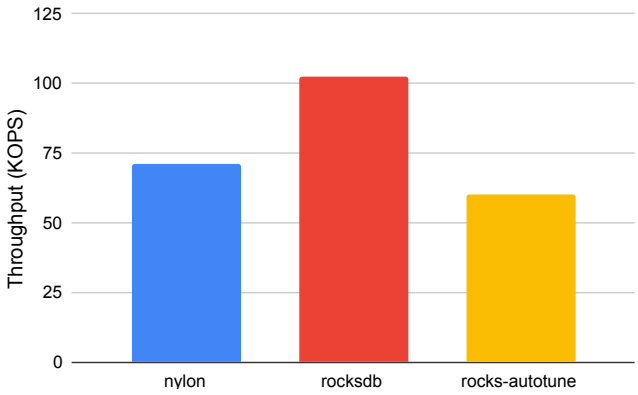
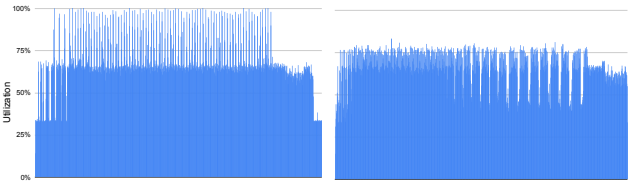


Fig. 8. Read throughput compared with RocksDB, Autotune RocksDB



(a) Nylon (b) RocksDB

Fig. 9. CPU utilization. The x-axis is time and the y-axis is CPU utilization. As with disk utilization, only the result is shown when heterogeneous storage system is used.

B. Heterogeneous storage system

There are researches which uses heterogeneous storage system to make cost-effective and high performance key value store. Heterogeneous storage system can be new paradigm, but traditional key value store doesn't consider heterogeneous storage system. Sometimes, its performance even gets worse than one with single slow storage. To solve this problem, some researches tries to use advantages of each storage material and maximize performance. Representative example of these researches is SpanDB[6]. SpanDB is key value store with storage system which is composed of NVMe SSD and SATA SSD. By using SPDK, SpanDB can reduce WAL overhead and it pipelines asynchronous request processing to maximize performance of NVMe SSD.

C. Scheduling mechanism

Most researches of key value store tries to modify overall structure. However, there are researches which focus on scheduling policy and optimize its performance. SILK[2] is the representative research, which adjust scheduling policy of conventional LSM based key value store. Priority policy of SILK is very similar with one of Nylon. Operations which are close to client gets higher priority. For example, flush gets the highest priority. And the goal of this research is to make latency critical key value store by reducing tail latency. However, this research has limitation because it doesn't consider heterogeneous storage system.

VII. CONCLUSIONS

Nylon tries to point out problems of conventional key value store with modern storage material. And we suggested new scheduling policy to solve this problem. The

most focused problems are 1) bottleneck shifting 2) poor performance with high cost. To solve these problem, we suggested dynamic flush and new priority scheme. Overall write performance is better than one of RocksDB and Autotune RocksDB. However, due to lazy compaction, Nylon is suffered from read amplification. Moreover, rough scheduling of Nylon make CPU utilization more fluctuated. Solution of these problem will be remained as future work.

REFERENCES

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[2] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in log-structured merge key-value stores," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 753–766. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/balmau

[3] "Rocksdb." [Online]. Available: https://github.com/facebook/rocksdb

[4] "Autotunerocksdb." [Online]. Available: https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html

[5] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: The design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 447–461. [Online]. Available: https://doi.org/10.1145/3341301.3359628

[6] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "Spandb: A fast, cost-effective lsm-tree based KV store on hybrid storage," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 17–32. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/chen-hao