Wayfinder: Speeding up Key-Value Separation by Avoiding I/O Based Indirection

Guy Khazma
guykhazma@cs.toronto.edu
University Of Toronto
Toronto, Canada

Myles Thiessen
mthiessen@cs.toronto.edu
University Of Toronto
Toronto, Canada

Eyal de Lara
delara@cs.toronto.edu
University Of Toronto
Toronto, Canada

Niv Dayan
nivdayan@cs.toronto.edu
University Of Toronto
Toronto, Canada

ABSTRACT

LSM-Trees are the backbone of modern key-value stores, supporting write-intensive workloads with balanced performance for point and range queries. Compaction in LSM-trees optimizes queries but poses significant overhead on the write path, especially for medium to large values. Key-value (KV) separation addresses this by storing values in a separate value log and pointing to them from the LSM-tree. This reduces write amplification as values are no longer rewritten during compaction. This KV separation, however, presents its own challenges. First, query performance suffers as the LSM-tree must first be searched for an address followed by querying the log for the associated value. Second, the value log requires expensive garbage collection as (1) the LSM-tree must be queried to determine whether a given value in the log is still the most recent version, and (2) a *reinsertion* is needed to update the LSM-tree with the addresses of the relocated KV pairs. To address these challenges, this paper investigates how to map values in the log via an in-memory hash table while using the LSM-tree to store small values and handle range queries. We implement Wayfinder on top of RocksDB and show its effectiveness in improving throughput while reducing space and write amplification.

CCS CONCEPTS

• **Information systems → Point lookups**; **Data scans**.

KEYWORDS

LSM Trees, Key-Value Separation

ACM Reference Format:
Guy Khazma, Myles Thiessen, Eyal de Lara, and Niv Dayan. 2024. *Wayfinder:* Speeding up Key-Value Separation by Avoiding I/O Based Indirection. In *2nd Workshop on Simplicity in Management of Data (SiMoD '24), June 14, 2024, Santiago, Chile.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3663351.3663880

1 INTRODUCTION

LSM-Tree. LSM-Tree [10] has become the backbone data structure for Key-Value (KV) stores that seek to optimize write-intensive workloads [7]. It ingests data efficiently by first buffering it in memory and sequentially writing it to secondary storage as sorted files. Simultaneously, it compacts (i.e., sort-merges) these sorted files to restrict the number of files that a query has to search.
Compaction Overhead For Medium/Large KV Pairs. Compaction is an I/O-intensive operation that consumes CPU and storage bandwidth. This is especially true for applications that store medium to large values (ranging from 100B to 16KB) [3, 8, 12]. In such cases, compaction becomes exceptionally costly due to the repetitive rewriting of large values across levels [3, 9].
KV Separation. A common technique to address the compaction overhead for medium/large values is KV separation, pioneered by WiscKey [9]. The idea is to separate keys and values across different data structures. Only the keys are sorted within the LSM-tree, while the values are stored in an append-only circular log. This substantially reduces compaction overheads for medium/large KV pairs as they are no longer rewritten during compaction. Figure 1 showcases the implementation of KV separation in RocksDB's BlobDB [1], which structures the value log as a collection of so-called *blob files*. When flushing the memtable, as indicated by the orange arrows, KV pairs with value sizes exceeding a preconfigured threshold are stored in blob files, while the key and address information (such as blob file number and offset) are stored within the LSM-tree's files.
Problem 1: Point Queries. Retrieving a KV pair from the value log involves a two-step process as illustrated in Figure 1 by the solid blue arrows. First, the LSM-tree is queried to retrieve the value's address within the log. Second, the relevant blob file is accessed to retrieve the value. This extra layer of indirection at least doubles point query latency when the LSM-tree does not fit in memory.
Problem 2: Garbage Collection (GC) Overheads. As values in the log are updated, the log must be garbage-collected to clean space occupied by deleted or superseded KV pairs. This process can be costly as it requires (1) a *Validity Check*, where the GC process verifies the validity of each KV pair by querying the LSM-tree, and (2) *Reinsertion*, which involves updating the LSM-tree with the new addresses of valid KV pairs relocated during GC [3, 8].

A recent approach to obviate these overheads is to couple GC with compaction [8]. The idea is that keys involved in compaction are already read and rewritten during the process, hiding the costs

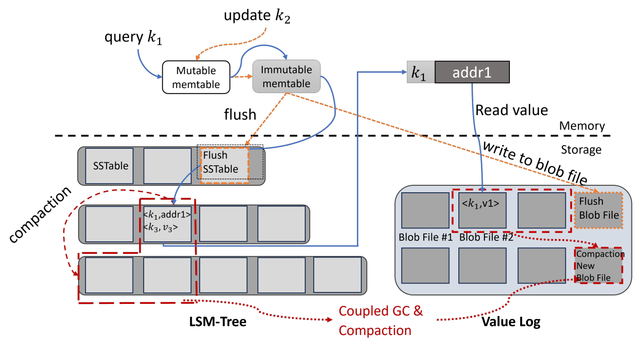

Figure 1: BlobDB. KV pairs with values exceeding a preconfigured size (e.g. k1) are stored in the blob files with their address stored in the LSM. Query, flushing, and coupled compaction & GC workflows are denoted with solid blue arrows, orange dashed arrows, and dashed red arrows respectively.

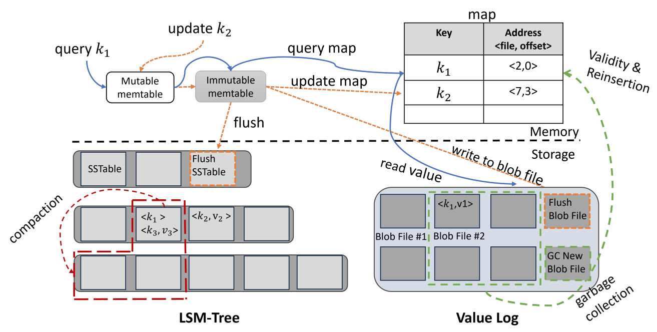

Figure 2: Wayfinder. KV pairs with value sizes exceeding a preconfigured threshold (e.g. k1) are stored in the blob files with their address stored in the in-memory map. Compaction and GC are independent. Query, flushing, compaction and GC workflows are denoted with solid blue arrows, orange dashed arrows, dashed red and dashed green arrows respectively.

of independent validity check and reinsertion. BlobDB implements this strategy, depicted in red in Figure 1, transferring valid KV pairs from old blob files to new ones during compaction, and subsequently deleting empty blob files. Nevertheless, this entails a write overhead to force compactions to occur so that a given blob file can be garbage-collected, or it entails a space overhead as a blob file cannot be deleted until all entries within it have been compacted.

Problem 3: Scans for Medium/Small KV Pairs. KV separation compromises scan performance, especially for smaller-to-medium-sized values [8, 11]. This issue stems from the dispersed placement of values within the log, necessitating random I/O for retrieval during range queries. Previous solutions tackle this challenge by employing parallel I/Os for prefetching during range queries [9], loosely sorting the value log to leverage the higher throughput of SSDs for sequential reads, as demonstrated in [8, 11] or storing small KV pairs solely in the LSM-tree [3].

Wayfinder. This paper investigates the concept of KV separation using an *in-memory mapping* from keys to their addresses on the value log. This study is inspired by ongoing research and development at Pliops [2, 5]. Our design, Wayfinder, addresses the indirection overhead for point queries by retrieving addresses from its in-memory map, thereby bypassing expensive secondary storage accesses. Wayfinder stores the keys of large entries without their corresponding values in the LSM-tree to facilitate quick identification and retrieval (using the map) during point queries. It also allows storing small entries fully within the LSM-tree to optimize range query cost. Moreover, Wayfinder reduces GC overheads by avoiding I/Os for validity checks and reinsertions, as both operations are executed against the in-memory map. We show that the size of the map is small compared to the data it represents.

Contributions. This paper makes the following contributions:

(1) We design Wayfinder, a KV store based on KV separation that addresses the aforementioned shortcomings by using an in-memory map to locate values on the value log.
(2) We show that Wayfinder outperforms BlobDB by 2.14x for point queries while also outperforming both RocksDB and BlobDB for writes by 3.1x and 1.78x, respectively. Wayfinder also reduces write amplification by 82% compared to RocksDB.

2 WAYFINDER

Design Principles. Wayfinder, depicted in Figure 2, is a KV store based on KV separation. It addresses the indirection overhead for point queries while solving the validity check and reinsertion overheads associated with compaction. It achieves this by utilizing an *in-memory map* that associates each entry with its corresponding address on the value log. The map ensures efficient access during queries and updates, bypassing the need to access secondary storage for retrieving or updating a KV pair's value address. To maintain correctness, Wayfinder updates the map with each operation, ensuring it contains mappings only for keys whose latest versions exist in the value log.

Data Ingestion. KV pairs are initially inserted into the memtable. Upon flushing the memtable to storage, Wayfinder differentiates between entries intended for the value log and those intended for the LSM-tree, utilizing a predetermined size threshold. If a KV pair is intended for storage in the LSM-tree, its entry is first removed from the in-memory map, if present, before flushing. Conversely, if the KV pair is meant for the value log, it is appended to the value log, and its address in the map is updated (potentially overriding any existing mapping). Subsequently, the key is flushed to the LSM *without* an associated value. This process is illustrated using orange dashed arrows in Figure 2. For example, consider key $k1$ in Figure 1, which is stored in blob file 2. Its address is stored in the map while the LSM stores the key without the value.

Selective KV Separation. Wayfinder employs selective KV separation [3], utilizing a size threshold to differentiate between small and large KV pairs. Large KV pairs are stored in the value log and managed through a dedicated garbage collection mechanism, while small KV pairs reside in the LSM-tree and benefit from the compaction process. This approach aims to minimize the overhead of accessing both the LSM-tree and the in-memory map for range queries on small keys. The granularity of block reads may result in high read amplification for small KV pairs if they are placed in the value log, as they are no longer sorted. In case of an update to the KV size, where the key may transition between the value log and the LSM, the map is updated accordingly to ensure it contains an entry for the key only if its latest version is in the value log.

Point Queries. Wayfinder processes point queries by first checking the memtable. If the KV pair is not found, it falls back to the in-memory map and queries it for the value address. If an address is found, it reads the value from the value log and returns. Otherwise, it attempts to read the KV pair from the LSM-tree. In Figure 1, for example, the query for $k1$, depicted in blue, discovers its address in the value log by querying the in-memory map (after failing to find it in the memtables). On the other hand, a query for $k3$ would instead fall back to the LSM, as no mapping for it exists in the map.

Range Queries. For range queries, we initially identify keys within the specified range using standard LSM-tree procedures by querying both the memtable and LSM-tree. Small keys with values are found in the LSM due to selective key-value separation. Keys without associated values are stored in the value log. Retrieving such keys involves obtaining their address from the in-memory map and then reading from the value log. As the map is stored in memory, the overhead involves just one additional memory I/O, regardless of whether the KV pair is stored in the LSM-tree or the value log.

Deletes. Wayfinder treats deletes as tombstones and flush them to the LSM. During flushing, any corresponding mappings in the map are removed. This process maintains correctness for subsequent queries by ensuring that a key has a mapping in the map only if its most recent version is in the value log.

Value Log GC. Wayfinder's GC operates independently of the LSM-tree compactions, performing validity checks and reinsertions solely using the in-memory map. This eliminates the overhead previously associated with these operations. Leveraging this capability, Wayfinder enables the adoption of *any* GC policy, effectively eliminating the tradeoff between write amplification due to compaction and space amplification caused by infrequent GC, a limitation found in all existing designs coupling GC with compaction. Figure 2 illustrates in green a GC process which selects a subset of the blob files and performs GC on them by interacting with the in-memory map for validity checks and reinsertion.

Compaction. LSM-tree compaction operates independently of the value log and allows for the adoption of *any* policy. During the compaction process, if a tombstone associated with a value stored in the value log is encountered, its corresponding map entry (if any) was removed when the deletion was flushed from the memtable. Moreover, keys without associated values, indicating they are stored in the value log, are handled conventionally. Figure 1 depicts in red a standard leveled subcompaction in BlobDB.

3 PRELIMINARY EVALUATION

Implementation. We integrated Wayfinder into BlobDB [1] to evaluate its performance within a production-grade KV store. Built on top of RocksDB, BlobDB implements KV separation by managing the value log as a collection of blob files written during flushing. BlobDB couples GC with compaction to avoid the validity checks and reinsertion overheads as discussed in Section 1. During compaction, BlobDB identifies and transfers valid KV pairs from old blob files to new blob files based on an age mechanism, subsequently marking obsolete files for deletion. Our implementation builds on top of BlobDB by incorporating an in-memory map, which is updated during flushing, as detailed in Section 2. For GC, we implemented a separate mechanism that monitors the space amplification of the blob files by tracking the total size of garbage bytes in each blob file. These statistics are updated during flushing as records are updated in the in-memory map. Once the space amplification surpasses a set threshold, garbage collection is triggered by transferring valid key-value pairs from the blob files with the most garbage to new ones. For the in-memory map, we utilize an open-source implementation of a parallel hashmap [6]. This is needed due to the map being accessed and modified simultaneously by various threads, including user workload threads, flush threads, and compaction threads.

Testbed. We conduct our experiments on AWS using a c5d.4xlarge EC2 instance. The instance has 16 vCPU cores with a clock speed of 3GHz, 32GB of memory and 400GB NVMe SSD. The operating system is Ubuntu 22.04.4 LTS, and the file system used is ext4. We evaluated the SSD's performance by running fio, measuring 4KB random reads, and achieving speeds of 723MB/s with 176k IOPS.

Setup. We compare Wayfinder against RocksDB and BlobDB. Our experiment first loads 100 million keys into each database, where keys are 16 bytes and values are 1KB. It then randomly updates 35 million keys. Following that, we run 5 million random point queries on existing keys. We configure the KV stores with recommended defaults, setting the memtable size to 64 MB and bloom filters with 10 bits per key. In addition, selective KV separation is disabled for the experiment, meaning all KV pairs are stored in blob files in the case of BlobDB and Wayfinder. Given our emphasis on minimizing indirection overhead, we have chosen to disable the built-in cache in RocksDB for all KV stores. In addition, we enable direct IO to make sure all accesses are done directly against the disk bypassing the system page cache. For BlobDB GC, we use the default 25% age cutoff, relocating KV pairs in the oldest 25% of blob files during compaction. For Wayfinder, we set the space amplification threshold to 1.2. To achieve this for RocksDB and BlobDB, we set the size ratio to 5 with dynamic leveling enabled. For consistency, we do the same for Wayfinder. For all systems, we set the maximum number of concurrent background jobs (compactions, flushes, and GC) to 4.

Results. Figure 3 on the top left shows the throughput for the insert and update phase. The throughput of Wayfinder is 3.1x higher than RocksDB and 1.78x higher than BlobDB. RocksDB shows the lowest throughput because it is affected by compactions, which repeatedly rewrite large key-value pairs across different levels. BlobDB is also impacted due to frequent triggering of GC, which is tied to compaction rather than space amplification. In contrast, Wayfinder only initiates GC when the space amplification exceeds 1.2, resulting in less contention with the user workload. Moreover, Wayfinder undergoes fewer compactions because of its smaller LSM tree. This is a consequence of not storing the addresses of KV pairs stored in blob files. This is also illustrated in the bottom right figure which depicts the write amplification of the workload. Wayfinder and BlobDB reduce write amplification by about 82% compared to RocksDB.

Figure 3 on the top right, depicts the throughput for the random reads workload. Wayfinder outperforms BlobDB by 2.14x since for every existing key BlobDB has to first access the LSM to retrieve the

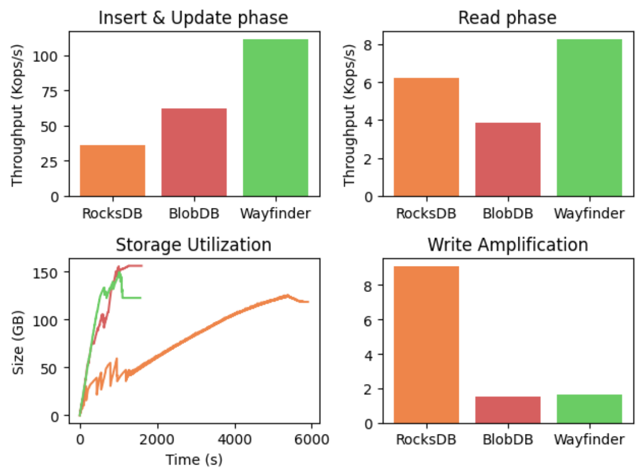

Figure 3: Results for insertion of 100M keys and update of 35M keys, followed 5M point queries for existing keys. The top left corner displays throughput for both the insert and update phases, while the top right corner depicts point query performance. Bottom left indicates the database size on storage with GC enabled for both Wayfinder and BlobDB, where Wayfinder space amplification threshold is set to 1.2. Bottom right showcases the overall write amplification.

address and only then read the value from the blob file. Wayfinder outperforms RocksDB by 1.32x times because RocksDB has to read the entire block and search for the key position within it.

Figure 3 on the bottom left illustrates the size of the database on storage. Wayfinder and BlobDB lines end earlier since they have higher throughput for insertion and update. The final disk size of Wayfinder and RocksDB is approximately the same since both are configured to maintain a space amplification of 1.2 and trigger GC or compaction whenever necessary. In contrast, BlobDB exhibits the highest space utilization because its GC mechanism is tied to compactions and is triggered only for the oldest 25% of blob files.

4 DISCUSSION & FUTURE WORK

Map Memory Footprint. A potential challenge posed by the map is its memory footprint. The address of a blob record is encoded using three varint64 values (ranging from 1 to 10 bytes) for the blob file number, offset, and size of the blob record. Additionally, two extra bytes are allocated for encoding compression and type information. Using 16-byte keys, 1KB values, and a maximum of 10K blob files, each map entry consumes 24 bytes, negligible compared to the size of the KV pair. This results in an estimated RAM usage of around 2.4GB, without considering the overhead of the hashmap structure itself. To enable our design to scale to larger number of entries we plan to adapt our design to use tabular filters [4] for future work. These filters are compact at the expense of potential false positives, as they prioritize memory efficiency over precision.
Recovery. If a crash occurs, we must reconstruct the in-memory map. To do so we query the LSM to identify keys lacking values; these keys are stored in the value log. Next, the latest value for each key is found by scanning through all blob files in the value log. This ability to retrieve the most recent version of a key from the value log is a common feature in existing value log designs, such as those employing curricular garbage collection mechanisms [9] or partition-based garbage collection [3, 11].
Value Log Layout & GC. Wayfinder enables to update the address of a KV pair located in the value log without affecting the LSM. This enables to leverage various methods to re-layout the value log in order to optimize GC and range queries. In particular we plan to explore gradually sorting the values on the value log over time in order to improve range query performance [11] and hot-cold data separation in order to improve GC efficiency [3, 8].
Snapshot Isolation. RocksDB and BlobDB provide consistent views for range scans and multi-gets. Wayfinder lacks this, as its in-memory map retains only the latest version for each key. We plan to modify the map to enable per-key version support and update it during compaction to reflect keys not visible in any snapshot.

5 CONCLUSION

Existing designs for key-value separation suffer from degraded performance for point queries and GC overheads associated with maintaining the key to address mapping in the LSM. We explored using an in-memory hash table to achieve fast point queries and mitigate garbage collection overheads while simultaneously leveraging the benefits of key-value separation.

6 ACKNOWLEDGMENTS

This research was supported NSERC grant #RGPIN-2023-03580 and NSERC grant #RGPIN-2024-05696.

REFERENCES

[1] 2021. BlobDB. https://rocksdb.org/blog/2021/05/26/integrated-blob-db.html.
[2] 2024. Pliops. https://pliops.com/.
[3] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.
[4] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[5] Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, et al. 2021. The End of Moore's Law and the Rise of the Data Processor. *VLDB* (2021).
[6] Gregory Popovitch. 2023. The Parallel Hashmap. https://github.com/greg7mdp/parallel-hashmap.
[7] Stratos Idreos and Mark Callaghan. 2020. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2667–2672.
[8] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated {Key-Value} Storage Management for Balanced {I/O} Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.
[9] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
[10] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.
[11] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. Fencekv: Enabling efficient range query for key-value separation. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3375–3386.
[12] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid key-value placement in lsm-based key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 305–318.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009