Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores

Giorgos Xanthakis[1], Giorgos Saloustros, Nikos Batsaras[1], Anastasios Papagiannis[1,2], and Angelos Bilas[1]
Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH)
Heraklion, Greece
{gxanth,gesalous,nikbats,apapag,bilas}@ics.forth.gr

ABSTRACT

Key-value (KV) separation is a technique that introduces randomness in the I/O access patterns to reduce I/O amplification in LSM-based key-value stores. KV separation has a significant drawback that makes it less attractive: Delete and update operations in modern workloads result in frequent and expensive garbage collection (GC) in the value log.

In this paper, we design and implement *Parallax*, which proposes hybrid KV placement to reduce GC overhead significantly and increases the benefits of using a log. We first model the benefits of KV separation for different KV pair sizes. We use this model to classify KV pairs in three categories *small, medium, and large*. Then, *Parallax* uses different approaches for each KV category: It always places large values in a log and small values in place. For medium values it uses a mixed strategy that combines the benefits of using a log and eliminates GC overhead as follows: It places medium values in a log for all but the last few (typically one or two) levels in the LSM structure, where it performs a full compaction, merges values in place, and reclaims log space without the need for GC.

We evaluate *Parallax* against RocksDB that places all values in place and BlobDB that always performs KV separation. We find that *Parallax* increases throughput by up to 12.4x and 17.83x, decreases I/O amplification by up to 27.1x and 26x, and increases CPU efficiency by up to 18.7x and 28x, respectively, for all but scan-based YCSB workloads.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3487012

CCS CONCEPTS

• **Information systems** → **Key-value stores**; **B-trees**; **Flash memory**; • **Software and its engineering** → **Garbage collection**.

ACM Reference Format:
Giorgos Xanthakis[1], Giorgos Saloustros, Nikos Batsaras[1], Anastasios Papagiannis[1,2], and Angelos Bilas[1]. 2021. Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3487012

1 INTRODUCTION

Key-value (KV) stores typically use at their core the write-optimized LSM-tree [34] index structure to handle bursty inserts and amortize write I/O cost. LSM-tree organizes data in multiple levels of increasing size. LSM-based designs have two important characteristics: 1) They always generate large I/Os and 2) they incur high I/O amplification [17]. This is still the right tradeoff for hard disk drives (HDDs): Under small, random I/O requests, HDD performance degrades by more than two orders of magnitude, from 100s of MB/s to 100s of KB/s. With the emergence of fast block-based storage devices, such as NAND-Flash solid state drives (SSDs) and block-based non-volatile memory devices (NVMe), behavior is radically different under small, random I/Os: At relatively high concurrency, these devices achieve a significant percentage of their maximum throughput.

Previous work has used a new technique, KV separation [1, 10, 20, 29, 33, 36, 37] to introduce some degree of randomness in I/Os generated by KV stores and reduce I/O amplification. KV separation appends KV pairs in a value log as they are inserted (in unsorted order) and essentially converts the KV store to a multistage index over the log. Therefore, compaction operations across LSM levels involve only keys and metadata, without moving values. This reduces I/O amplification dramatically for large KV pairs.

[1]Also with the Department of Computer Science, Univ. of Crete, Greece.
[2]Currently with Facebook, UK.

However, KV separation results in frequent garbage collection (GC) for the log [10, 42]: Delete and especially update operations that are common [9, 43] generate old (garbage) values in the value log that need to be garbage collected frequently to avoid excessive space amplification. Similar to past experience [7, 40], garbage collection in the value log is an expensive process. Typically, GC requires two main and expensive operations: identify and relocate valid values.

To identify valid values, we need to scan each log segment *and* lookup each key in the LSM index. Reads in the multi-stage index are already expensive in LSM-based KV stores. This becomes exceedingly expensive as the number of keys in each log segment increases, e.g. when there is pressure to free space eagerly or when the KV pair size is small. To relocate valid values we need to re-append them to the log and update pointers in the index that point to them, generating additional I/Os and high amplification. Then, the old segment can be reclaimed in full for later use.

Table 1 shows the effects of GC in I/O amplification using RocksDB (no KV separation) and BlobDB (with KV separation) for small KV pairs that dominate in Facebook production workloads [9]. BlobDBnoGC is a configuration of BlobDB with GC disabled. Even though it is not a realistic setup, we use it to quantify the benefits of KV separation without GC. BlobDBnoGC has 8.7x less I/O amplification compared to RocksDB. When using GC, I/O amplification in BlobDB is 1.57x higher than RocksDB (27.4 vs. 17.4). These numbers include only the identification cost of valid values since there are no deletes or updates and no relocation occurs. Identifying valid keys consumes valuable read throughput from client get and scan operations. Delete and update operations increase I/O amplification further because the GC mechanism needs to relocate valid KV pairs by re-appending them to the log and consumes valuable write throughput.

In this paper, we propose *Parallax*, an LSM-based KV store design for fast storage devices that uses hybrid KV placement to address these issues. *Parallax* provides the benefits of using a log without excessive GC overhead, as follows. First, we use the observation that workloads typically use KV pairs of different sizes [9], including small, medium, and large KV pairs. In particular, small KV pairs constitute a large percentage (60%) of the workload [9], although medium and large KV pairs may dominate in terms of cumulative size. We model the benefits of KV separation and we identify three, size-based categories with different behavior and benefits (Figure 1): small KV pairs (less than 100 B) that do not benefit significantly in I/O amplification from using a log (up to 3x), large KV pairs (more than 1024 B) that exhibit order-of-magnitude benefits (6x-12x), and medium KV pairs in between that have smaller but significant benefits (3x-6x). We note that large values incur low GC overhead, small and medium values incur high GC overhead, with small values introducing excessive GC costs. This large variance in the benefits of KV separation (Table 1), combined with the significant overhead of garbage collection in the value log makes KV separation less attractive for workloads with mixed KV-pair sizes.

	BlobDBnoGC	RocksDB	BlobDBGC
I/O amplification	2.0	17.4	27.4

Table 1: I/O amplification in BlobDB (with/without GC) and RocksDB for inserts of small (33 B) KV pairs.

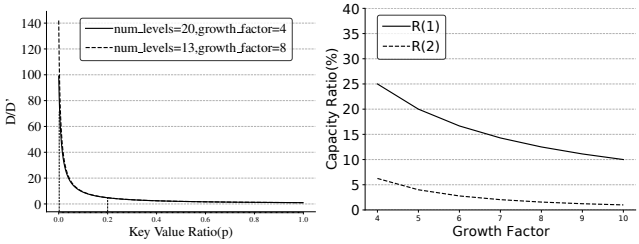
Parallax uses different KV placement strategies for different KV pair sizes. It always places large KV pairs in a log with a clear benefit in I/O amplification at low GC cost. *Parallax* stores small KV pairs in place, within each LSM level. We use a B+-tree index for each LSM level and store small KV pairs in its index leaves, while it performs transfers from level to level as in LSM-type approaches [19, 34, 41].

For medium KV pairs, *Parallax* uses a new technique: We place medium KV pairs in a log up to the last level and then we compact the medium log in the last level, freeing the medium log. Given that the medium log is freed when KV pairs are re-placed in the LSM structure, there is no GC overhead associated with the medium log. Therefore, medium KV pairs, combine most of the I/O amplification benefits with almost no GC overhead. *Parallax* essentially trades space amplification for the medium log for a significant reduction in I/O amplification. However, since all levels grow with a factor f, typically 8 for space efficiency purposes [17], space amplification in *Parallax* is limited.

Using hybrid KV placement in multiple logs and in place introduces challenges with ordering and recovery. *Parallax* uses log sequence numbers to maintain ordering of keys within each region. In addition, *Parallax* offers crash consistency and can recover to a previous (but not necessarily the last) write, discarding all subsequent writes, as is typical in modern KV stores [21].

We implement *Parallax* and evaluate a full-fledged prototype with YCSB and different workloads. We compare *Parallax* to RocksDB [19] that places all values in place and with BlobDB [20] that uses KV separation. Our evaluation shows that, compared to RocksDB and BlobDB, *Parallax* increases throughput by up to 12.4x and 17.83x, decreases I/O amplification by up to 27.1x and 26x, and increases CPU efficiency by up to 18.7x and 28x, for YCSB workloads Load A through Run D with KV pairs of mixed sizes. For range queries (scans) in workload Run E with mixed size KV pair sizes, *Parallax* results in 7.95x higher throughput than BlobDB and 1.48x lower throughput than RocksDB, closing the gap compared to previous systems that perform KV separation.

Overall, the main contributions of our work are:



(a) KV separation benefit **(b) R(i)**

Figure 1: (a) Amplification ratio with and without KV separation ($\frac{D}{D'}$) as a function of p. (b) Percentage R(i) of space used by the first n-i levels compared to the total space of the KV store, as a function of f.

(1) We propose hybrid KV placement that achieves most of the benefits of using a log for KV separation without excessive GC overhead.
(2) We present an asymptotic analysis that describes I/O amplification in leveled LSM-based KV stores with and without KV separation, as a guide for our design.
(3) We design *Parallax* that provides hybrid KV placement, addressing issues of ordering and recovery, handling variable size keys and variable size updates for all logs and in-place values.

2 MODELING I/O AMPLIFICATION

In this Section, we use an analytical model [4] that calculates I/O amplification in LSM-based KV stores with leveled compaction [34]. We estimate the benefits of KV separation for different KV sizes and guide our design. Our model uses as a basis the original LSM model [34], which does not consider update and delete operations. We extend the original model to estimate I/O traffic during compactions. We then calculate I/O amplification for KV stores that perform leveled compaction with and without a value log.

Amplification has two significant components: First, assuming the level size grows by f times across consecutive levels, the system reads and writes an excess of f times more bytes, compared when merging L_i to L_{i+1}. Second, the cost of data reorganization across multiple levels as data travel towards the lowest (largest) level: In a system with l levels, each unique KV pair moves through all levels, resulting in l times excess traffic. We refer to these quantities of excess traffic as *merge* and *level* amplification, respectively.

$$\begin{aligned} D &= \frac{S_l}{S_0}(S_0) + 2\sum_{j=1}^{S_l/S_0}((j-1) \bmod f)\cdot S_0 \\ &+ \frac{S_l}{S_1}(2S_1) + 2\sum_{j=1}^{S_l/S_1}((j-1) \bmod f)\cdot S_1 \\ &+ \ldots \\ &+ \frac{S_l}{S_{l-1}}(2S_{l-1}) + 2\sum_{j=1}^{S_l/S_{l-1}}((j-1) \bmod f)\cdot S_{l-1} \quad (1) \end{aligned}$$

$$\begin{aligned} Eq.\ 1 \Rightarrow \quad D &= (2l-1)S_l + 2S_{l-1}\sum_{j=1}^{f^1}(j-1) \bmod f \\ &+ \ldots + 2S_0\sum_{j=1}^{f^l}(j-1) \bmod f \Rightarrow \\ D &= (2l-1)S_l + 2S_{l-1}\Big(\frac{f^1}{f}\cdot\frac{(f-1)(f-1+1)}{2}\Big) \\ &+ \ldots + 2S_0\Big(\frac{f^l}{f}\cdot\frac{(f-1)(f-1+1)}{2}\Big) \Rightarrow \\ D &= S_l(l-1+fl) \quad (2) \end{aligned}$$

$$\begin{aligned} D' &= \frac{K_l}{K_0}(K_0) + 2\sum_{j=1}^{K_l/K_0}((j-1) \bmod f)\cdot K_0 \\ &+ \ldots \\ &+ \frac{K_l}{K_{l-1}}(2K_{l-1}) + 2\sum_{j=1}^{K_l/K_{l-1}}((j-1) \bmod f)\cdot K_{l-1} \\ &+ S_l \Rightarrow D' = K_l\Big(l-1+fl\Big) + S_l \quad (3) \end{aligned}$$

$$\frac{D}{D'} = \frac{D/S_l}{D'/S_l} = \frac{(l-1+fl)}{p*(l-1+fl)+1} \quad (4)$$

Equation 1 captures I/O amplification in the insert path under the assumption that during a merge operation, the lower level is fully read and written [10, 19, 33, 34, 37, 41]. D is the amount of I/O traffic produced until all S_l data reach L_l. If S_0 is the size of the in-memory L_0 and S_l is the size of the last level, then we can assume that the entire dataset is equal to S_l and that all data will eventually move to the last level S_l. Then, S_l/S_i is the total number of merge operations from L_i to L_{i+1}, until all data reach L_l.

Equation 1 consists of multiple subexpressions (rows), one per level, to capture level amplification. Each subexpression captures merge amplification between two consecutive levels, using two terms. In each subexpression, the first term represents the data of the upper (smaller) level that have to be read and written during the merge operation. For each

level L_i, $0 \leq i \leq l-1$, each time one of the S_l/S_i merge operations occurs, all data stored in L_i are read and written, thus causing I/O traffic of size $2S_i$ (first term). Note that, in the first subexpression for L_0 that resides in memory, the factor of 2x is missing in the first term, indicating that we do not perform I/O to read data that are already in memory.

The second term captures the total amount of data that are read and written from L_{i+1} in order to merge the overlapping ranges of L_i and L_{i+1}. The $\sum$ operator expresses the fact that batches of size S_i require S_l/S_i merge operations at the corresponding level. The mod operator captures the fact that the size of the lower (larger) level grows *incrementally* up to f: in the first merge operation the lower level has no data (i.e., $j-1=0$); in subsequent merge operations it contains data 1x, 2x to (f-1)x the data in the upper level. Merging requires both reading and writing data, hence the factor of 2x before the sum.

We can re-write Equation 1 as Equation 2 to express the amount of data read and written, until all data reach the lowest level.

2.1 KV separation benefits

Similarly to Equations 1 and 2, Equation 3 calculates traffic for KV stores that use KV separation. Each SST now stores only keys and thus its size is equal to K_i, with $S_i = K_i + V_i$. The value log contains all KV pairs stored in the system, so its size is S_l. Consequently, we can write Equation 3 that expresses the amount of data read and written when using a KV log, until all data reach the lowest level. The last term S_l in Equation 3 represents the fact that KV pairs are appended to the log.

Finally, the ratio $\frac{D}{D'}$ from Equations 2 and 3 expresses the benefit of KV separation over in place values. If we assume that p is the key to value size ratio $p = K_l/(K_l + V_l) = K_l/S_l$, then we can introduce p in this ratio by dividing both numerator and denominator with S_l, which is the total size of values, as shown in Equation 4.

2.2 Discussion

Figure 1(a) plots this ratio as a function of p. Based on this figure we can use two thresholds for p, T_{SM} and T_{ML}, to divide KV pairs in three categories, based on their benefits in I/O amplification from KV separation:

(1) Large KV pairs with $0 < p \leq T_{ML}$, where the benefit can be more than an order of magnitude and where GC does not introduce significant overhead.
(2) Small KV pairs with $T_{SM} < p \leq 1$, where the benefit is small and the GC overhead becomes excessive.
(3) Medium KV pairs, with $T_{ML} < p \leq T_{SM}$, where the benefit is smaller, but still substantial and will manifest only if we reduce the corresponding GC overhead.

We should note that there are no models available for the cost of GC in systems that use KV separation. Experimental evidence from our and related work is that these overheads are high. Delete and especially update operations, which are common in modern workloads [9], cause fragmentation in the value log. To avoid high space amplification [17], there is a need for frequent garbage collection (GC) in the value log, which incurs high overhead [7, 10, 20, 42]. Typically, the system initiates GC periodically and after a configurable amount of update (delete) operations. The log is usually organized as a list of contiguous chunks of space (*log segments*). GC scans KV pairs in a configurable number of log segments to identify and relocate valid KV pairs, using and updating the multilevel KV index. Identifying valid KV pairs incurs *lookup cost*. Lookup cost depends significantly on the number of KV pairs in a segment. Especially for small KV pairs, this cost is high. In addition, lookup cost is independent of the workload; Even with a small percentage of delete and update operations GC must perform a lookup for each KV pair in a log segment. Relocating valid KV pairs at the end of the log incurs *cleanup cost* for transferring KV pairs and updating index pointers to the new KV locations. Cleanup cost depends mostly on the percentage of updates and deletes.

In our work we approximately set $T_{ML} = 0.02$. KV pairs beyond this boundary are considered *large* and benefit significantly from placement in the log. In addition, the GC overhead for large KV pairs is mediocre and is not likely to offset these benefits. On the other extreme, we use $T_{SM} = 0.2$. Beyond this point, for small KV pairs, I/O amplification benefits are small, whereas GC overheads are high and likely to dominate. We observe, based on previous work for the characteristics of real-life workloads [9], that KV pair sizes span all three categories defined by the model.

This classification leaves a relatively large range for medium KV pairs. For instance, if we assume that keys have a size of roughly 20 B, then the small, large, and medium categories will have values with sizes of up to 80 B, above 1000 B, and in-between, respectively.

We note that this model can be simplified for systems, such as *Parallax*, where the index stores only fixed-size prefixes instead of variable-size keys. To keep the model more general and applicable to other systems as well, we use the above formulation. In practice, keys are typically smaller than values and similar in size to prefixes.

We believe that future work should examine more detailed models and the implications of the specific thresholds. Today, there are no available models that take into account update and delete operations, or the percentage and distribution of each operation type for designs that support KV separation. Extending existing models to consider these aspects is an interesting and challenging problem.

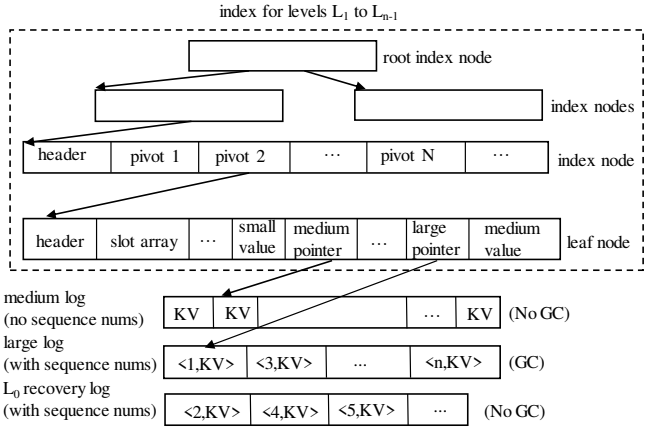


Figure 2: Overview of index and log design for levels L_1 to L_{n-1} in *Parallax*. Levels L_0 and L_n always store medium values in place.

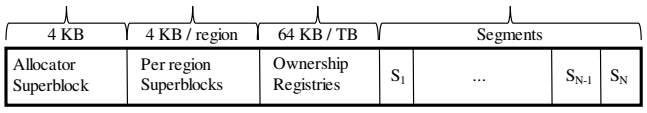


Figure 3: Device layout used by *Parallax* and its allocator. A segment may belong to the large, medium, L_0 recovery, or region allocation log, or to a level index.

3 *PARALLAX* DESIGN

3.1 Data organization

Parallax is a leveled, LSM KV store that offers a dictionary API (insert, delete, update, get, scan) for variable size KV pairs. KV pairs are organized in non-overlapping ranges, called *regions*. Each level in a region uses a concurrent B+-tree index [5, 6, 37, 41] and the region as a whole uses three logs: L_0 recovery log, medium log, and large log. Figure 2 shows an overview of *Parallax*. All KV logical structures consist of fixed size segments on the device (Figure 3). Currently, we use 2 MB segments. We discuss the *Parallax* allocator and how it recovers from failures later in this section.

The B+-tree index for each level consists of two types of nodes: *index* and *leaf* nodes. Index nodes store pivots (full keys), whereas leaf nodes store either (a) the pair <key_prefix, pointer> to the KV location or (b) the actual KV pair. Index and leaf nodes have a configurable size. We determine experimentally that 12 KB and 8 KB respectively lead to better performance.

When a KV pair is placed in a log, *Parallax* stores in the index leaf a key prefix and a pointer to the KV pair. The key prefix size is configurable, currently set to 12 B. *Parallax* uses prefixes for key comparisons inside leaves because they significantly reduce I/O to the logs [8, 30, 36, 37].

To store a variable number of KV pairs with variable size in leaves, we use two dynamically growing segments, the *slot array* and the *data segment* [11, 24]. The slot array is a small array with 4 B cells that grows from left to right. We reserve the three high-order bits of each cell to store the KV category. The rest of the bits contain an offset inside the leaf with the location of the actual data. The data segment is an append only buffer that contains either pointers to the log or the in place KV pairs themselves, and grows from right to left. When the slot array and the data segment meet, the leaf becomes full. Update operations append the new value to the data segment and update the slot array.

Get operations examine hierarchically all levels from L_0 to L_n and return the first occurrence. *Scan* operations create one scanner per-level and use the index to fetch keys in sorted order. They combine the results of each level to provide a global sorted view of the keys. *Delete* operations mark keys with a tombstone and defer the delete operation similar to RocksDB [19], freeing up space at the next compaction. *Update* operations are similar to a combined *insert* and *delete*.

Finally, *insert* operations write each KV pair in L_0, similar to all LSM KV stores. At each *insert* operation, a KV pair is categorized as small, medium, or large, by comparing the ratio p of prefix over KV-pair size to T_{SM} and T_{ML}. Based on the category, *Parallax* uses its hybrid strategy for placement, as we discuss next.

3.2 Handling small KV pairs

Parallax always stores small KV pairs in place, in the B+-tree leaves, and moves them from level to level with regular compactions. Similar to bLSM [41], *Parallax* compacts level L_i to L_{i+1} in a bottom-up fashion: It merges the sorted leaves of levels L_i and L_{i+1} and builds the B+-tree index for L'_{i+1}. In this manner, leaves are always full and compactions require fewer CPU cycles because they avoid index traversals.

3.3 Handling large KV pairs via GC

For large KV pairs, *Parallax* always performs KV separation [10, 20, 33, 37]: It places large KV pairs in the large log, maintains and compacts pointers in the index from level to level, and uses GC to reclaim space, as follows.

First, *Parallax* avoids full scans in the large log. For this purpose, we store in a private *GC region* information about the space used by invalid KV pairs in each segment, as a KV pair <segment, invalid-byte-count>. We represent segments with their 8-byte device offset and we use an 8-byte invalid-bytes counter per segment. When a compaction thread discovers a deleted or updated large KV pair, it performs a read-modify-insert operation on the corresponding <segment, invalid-byte-count> pair and updates the number of invalid bytes in the segment. Although this operation incurs overhead, it is

only required for large KV pairs and eliminates the need for full log scans. In addition, reads to the GC region are always served in memory because the GC region is small and fits in L_0, e.g. 8 MB for a 1 TB device and 2 MB segments. The GC region contains only small KV pairs, uses only regular compactions without GC, and therefore, is recoverable.

Then, *Parallax* reclaims segments with invalid KV pairs. It places segments that exceed a preconfigured threshold (10%) in an eligibility list. If there are no such segments and there is capacity pressure, then it considers all segments in the GC region as eligible. *Parallax* uses a dedicated GC thread to scan eligible segments. For each segment, it iterates over all KV pairs and uses lookup operations to identify valid KV pairs and append them to the large log. After all valid KV pairs are appended, it reclaims the segment.

3.4 Handling medium KV pairs

For medium KV pairs, *Parallax* uses a hybrid technique: It performs KV separation for all levels except the last one or two levels, by placing medium KV pairs in the medium log. As a result, it defers compaction of medium values up to the last few levels. At the last level(s), e.g. L_n, it moves medium KV pairs in place, similar to small KV pairs. Once medium values are moved in place, *Parallax* can reclaim the medium log segments, without the need for expensive GC. Placing medium KV pairs in a log and deferring their compaction, raises two questions: (a) What is the size of the medium log and the associated space amplification? (b) How can we merge efficiently the medium log back in the LSM structure?

Medium log size: With respect to the size of the medium log, we observe that a the cumulative capacity of the first levels in an LSM-based KV store is a small percentage of the last one or two levels. Given a growth factor f, the total capacity S_k of the first k levels is $S(k) = S_0 * \frac{(1-f^k)}{(1-f)}$. Therefore, the capacity ratio $R(i) = \frac{S(N-i)}{S(N)}$ of the first N-i over all N levels is $R(i) = \frac{1-f^{N-i}}{1-f^N}$.

Figure 1(b) shows R(1) and R(2) for growth factors f between 4 and 10. The value f=4 is optimal for LSM-tree and results in minimum I/O amplification [4]. On the other hand, a larger f reduces space amplification at the expense of increased I/O amplification for high update ratios: A larger f results in fewer intermediate levels and therefore, less wasted space, if we assume that intermediate levels contain only update operations. For this reason, production systems prefer to use f around 8-10.

R(1) shows that in the worst-case scenario, where all KV pairs are medium, deferring compaction up to level N-1 (excluding only the last level), wastes between 10% ($f = 8$) and 25% ($f = 4$) of the device capacity. R(2) shows that if we merge medium KV pairs at level N-2, then we waste at most 6% of the device capacity. At the same time, we avoid reorganizing medium KV pairs for all but the last one or two levels. We evaluate further this trade-off between space and I/O amplification in our experimental analysis.

Efficient merging of the medium log: The compaction process in LSM-tree [34] requires inserting keys from L_i to L_{i+1} in sorted order to amortize I/O costs. Otherwise, this process will result in excessive data transfers.

When we merge medium KV pairs from the log in place, e.g. in level N, the index of level N-1 already contains sorted pointers to the KV pairs of the medium log. However, a full scan of an unsorted medium log will cause a significant penalty in traffic: medium KV pairs in the order of hundreds of bytes will result in 4 KB I/O operations, i.e. up to 40x traffic for 100 B KV pairs.

To overcome this cost, *Parallax* maintains sorted segments in the medium log. To achieve this, we use L_0 and its recovery log as follows. We insert medium KV pairs *in place* in L_0 (in memory) and we also append them to the L_0 recovery log, along with small KVs for recovery purposes. During compaction from L_0 to L_1, we use the L_0 index to store medium KV pairs in the medium log as a sorted run of segments and we store pointers in the L_1 index. Merging the medium log in place, requires at most one segment from each sorted run of segments in the medium log.

One concern for merging the medium KV pair log in place is the amount of buffering required in memory for the sorted runs, during the merge operation. Assuming only inserts of distinct keys, medium KV pairs, and device capacity of about 10 TB, the medium log at L_{n-1} is about 1 TB with a growth factor of 10. If L_0 is 64 MB, there are about 16K sorted runs for medium KV pairs in a 1 TB L_{n-1} medium log. Given that fast storage devices, such as SSDs, operate at peak throughput with I/Os of tens or hundreds of KB, we need only to fetch in memory, e.g. a 64 KB chunk for each sorted run of the medium log. As a result, we need at most 1 GB memory for buffering purposes, when merging at L_{n-1}. If medium KV pairs are merged at L_{n-2}, then we only require about 100 MB of buffering.

In the presence of updates, the maximum size for the medium log can exceed the above calculation. In case updates occur to different keys, e.g. each key is updated once, then the medium log size will remain manageable. In case updates occur to a small set of keys, then the size of the medium log can become excessive. In this case, when the medium log exceeds a threshold, *Parallax* merges medium KV pairs in place earlier and reclaims the medium log. Essentially, the percentage and type of updates affects how late or early the medium log will be merged in place.

Finally, during the compaction that merges medium KV pairs in place, e.g. at L_n, the size of L_{n-1} and L_n needs to be

calculated based on their number of bytes rather than the number of keys to satisfy the growth factor and maintain the properties of LSM-tree.

3.5 Data recovery

Recovery in *Parallax* relies on value logs. In addition to the medium and large logs, for recovery purposes, *Parallax* uses a temporary log for small KV pairs (Figure 2), called L_0 recovery log. Conceptually, *Parallax* places each KV pair in the respective log. After a failure, *Parallax* can recover all KV pairs by replaying the logs. The actual implementation of the recovery scheme is more involved, as we discuss next.

We always place large KV pairs directly in the large log, upon insert, where they remain for their lifetime. We place small KV pairs in the L_0 recovery log *and* write them in place in L_0. When we compact L_0 in L_1, we can reclaim the L_0 recovery log in full. Therefore, the size of the L_0 recovery log is bounded by the size of L_0 and is typically a few MB.

We could place medium KV pairs directly in the medium log, however, this would require more I/O traffic to sort medium log runs (discussed above). For this reason, we initially insert medium KV pairs in place in L_0. When we compact L_0, we also sort medium KV pairs and place them in the medium log as a sorted run. To ensure medium KV pairs are recoverable while they live in L_0, we also write them to the L_0 recovery log. Medium KV pairs do not affect reclaiming the L_0 recovery log, which still occurs after L_0 compactions.

Parallax needs to maintain ordering across the large and L_0 recovery logs (but not the medium log) and handle cases with KV pairs that change category after an update. For this reason, it uses per region 8-byte *Log Sequence Numbers (LSNs)*. Each appended KV pair has an LSN, which we increment atomically before appending it. During region recovery, *Parallax* replays each log entry of the two logs with the correct order, as indicated by their LSN number.

Finally, similar to other KV stores [21], *Parallax* acknowledges writes as soon as they are written in memory. KV pairs are recoverable after a group commit operation that flushes the log tail asynchronously to the device, currently in chunks of 256 KB for I/O purposes. Therefore, *Parallax* can recover to a previous consistent point, which may not include the last (acknowledged) write(s). Most KV stores (and *Parallax*) can be configured to acknowledge writes after they are written to the device or to perform more frequent flush operations, but these are not commonly used as they increase acknowledgment delay or I/O overhead.

3.6 Space allocation and recovery

Parallax uses a memory bitmap for the device to allocate segments to all regions and for different purposes. For 2 MB segments, this bitmap is about 512 KBits or about 64 KB per 1 TB device capacity. Regions use a global lock to allocate and free segments from the in-memory bitmap, during insert and compaction operations.

Parallax needs to keep track of space allocation in the presence of failures. This is done cooperatively by the different regions, without introducing any additional synchronization. For this purpose, each region maintains (a) an allocation log and (b) an ownership registry, that are both private and persistent (Figure 3). All region operations that allocate or free a segment, record these actions in the region allocation log. To limit the size of the allocation log and reduce recovery time, *Parallax* periodically, trims the allocation log for each region using the region ownership registry. The ownership registry is a private bitmap, similar to the in-memory allocator bitmap which records only the segments used by the region. The region trims its allocation log, by scanning the log and updating its ownership registry with the respective allocate and free operations, essentially performing a checkpoint.

Each region uses a superblock (4 KB) for configuration information (Figure 3). The superblock is updated after each compaction or after a number of log flush operations, e.g. every few chunks. Both of these are coarse grained operations and do not affect common path performance. *Parallax* uses a system superblock (4 KB) for global configuration information (Figure 3), which however, is not updated frequently.

After a failure, *Parallax* reads the superblock and the registry of each region, replays all region allocation logs, and rebuilds the in-memory allocation bitmap.

3.7 Direct and memory-mapped I/O

Parallax carefully uses two I/O paths, direct I/O via system calls and memory-mapped I/O [12]. *Parallax* performs device I/O in the following occasions:

- Large, full-segment (2 MB) reads, writes at compactions.
- Large writes of log chunks (256 KB) at log flushes.
- Large reads of full log segments (2 MB) for GC.
- Small, random reads of index and leaf nodes (B-KB) and log pages (B-KB) during get, scan operations.
- Small (B-KB) reads, writes to region metadata.

Parallax uses direct I/O for all cases, except get and scan operations that use memory-mapped I/O. We use direct I/O because: (a) it is more efficient for large I/Os, (b) it allows explicit control of allocator state in the device for recovery purposes, and (c) avoids pollution of the read cache with data that need not reside in memory. We use memory-mapped I/O for small, random reads during get and scan operations because it eliminates system calls and data copies [25, 35, 37].

4 METHODOLOGY

Our testbed consists of a single server which runs the key-value store and the YCSB [14] client. Our server is equipped

Workload	KV Size (B)	KV Mix S%-M%-L%	# pairs (Millions)	Cache (GB)	Dataset (GB)
Small (S)	33	100-0-0	500	2	10
Medium (M)	130	0-100-0	200	4	26
Large (L)	1024	0-0-100	100	16	100
Small Dom. (SD)	-	60-20-20	100	4	22.5
Medium Dom. (MD)	-	20-60-20	100	4	25.5
Large Dom. (LD)	-	20-20-60	100	4	62.5

Table 2: Workloads we use in our evaluation.

with two Intel(R) Xeon(R) CPU E5-2630 running at 2.4 GHz, with 16 physical cores for a total of 32 hyper-threads and with 256 GB of DDR4 DRAM. It runs CentOS 7.3 with Linux kernel 4.4.159. The server has one Intel Optane P4800X device model with total capacity of 375 GB [27].

We compare *Parallax* to RocksDB that places all KV pairs in place and to BlobDB that always performs KV separation. We configure BlobDB and RocksDB v6.11.4 with 128MB for the WAL and 4 threads for background I/O operations (log flushing and compactions), on top of *XFS* with disabled compression and jemalloc [18], as recommended. We configure RocksDB to use direct I/O, as recommended by RocksDB [22], and we verify that this results in better performance in our testbed. Furthermore, we use RocksDB's user-space LRU cache, by varying the size of the cache based on the workload which we similarly use for *Parallax* as shown in Table 2. To have an equal comparison we disable bloom filters for RocksDB and *Parallax* as BlobDB does not yet support them. Additionally, we run a configuration of RocksDB with bloom filters (10 bits per key) enabled. We use this configuration to estimate the reduction in I/O amplification from the use of bloom filters. For BlobDB we set GC to scan 30% of the log when GC threads wake up after a compaction. For *Parallax* we set GC to reclaim a log segment when 10% of the segment is invalid. We choose relatively aggressive values for these thresholds because preventing space waste is important in production systems [26].

We use a C++ version of YCSB [39] and we modify it to generate the required KV pair sizes. In all systems we use a total of 8 databases and 16 threads. We show results for all YCSB workloads. For Load A and Load E we use a uniform distribution, while for Run A-Run D, Run E we use a Zipfian distribution. We use Load A and Run A for large parts of our analysis as the mixes of these two operations exhibit the lookup and cleanup costs of our analysis of garbage collection in the log: Load A includes 100% insert operations and exhibits only the lookup cost of GC, whereas Run A includes update (50%) and read (50%) operations and exhibits both lookup and cleanup costs. Furthermore, in the case of mixed workloads, update operations of Run A change the sizes of KV pairs and thus their category. We also pay attention to Run E that includes scan operations which are important KV separation.

In terms of KV pair sizes, we evaluate six workloads, three with a single KV pair size (Small, Medium, Large) and three with mixed KV pair sizes (Small-Dominated, Medium-Dominated, and Large-Dominated), as suggested by Facebook and Twitter workloads [2, 9, 43]. Table 2 summarizes our workloads. We describe each workload as the percentage of KV pair sizes it uses, e.g. 100% small KV pairs. We also show the dataset size and the size of the cache we use in the KV store(s) to ensure realistic I/O traffic to the devices.

We use the following key and value sizes for each category. All categories have a key size of 24 B on average. The value size is 9 B for small KV pairs, 104 B for medium KV pairs, and 1004 B for large KV pairs. These result in $p = 0.02$ for large KV pairs, $p = 0.72 > 0.2$ for small KV pairs, and $p = 0.19$ (between 0.02 and 0.2) for medium KV pairs, closer to the small rather than the large category as this is more representative of actual workloads. We use $T_{SM} = 0.2$ and $T_{ML} = 0.02$. We leave a more detailed exploration of T_{SM} and T_{ML} based on the workload in terms of KV size and operation distributions, for future work.

In all cases, we examine throughput in Kops/s, I/O amplification as the ratio of device over application I/O traffic (reads, writes), and efficiency in Kcycles/op, calculated as:

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}}$$

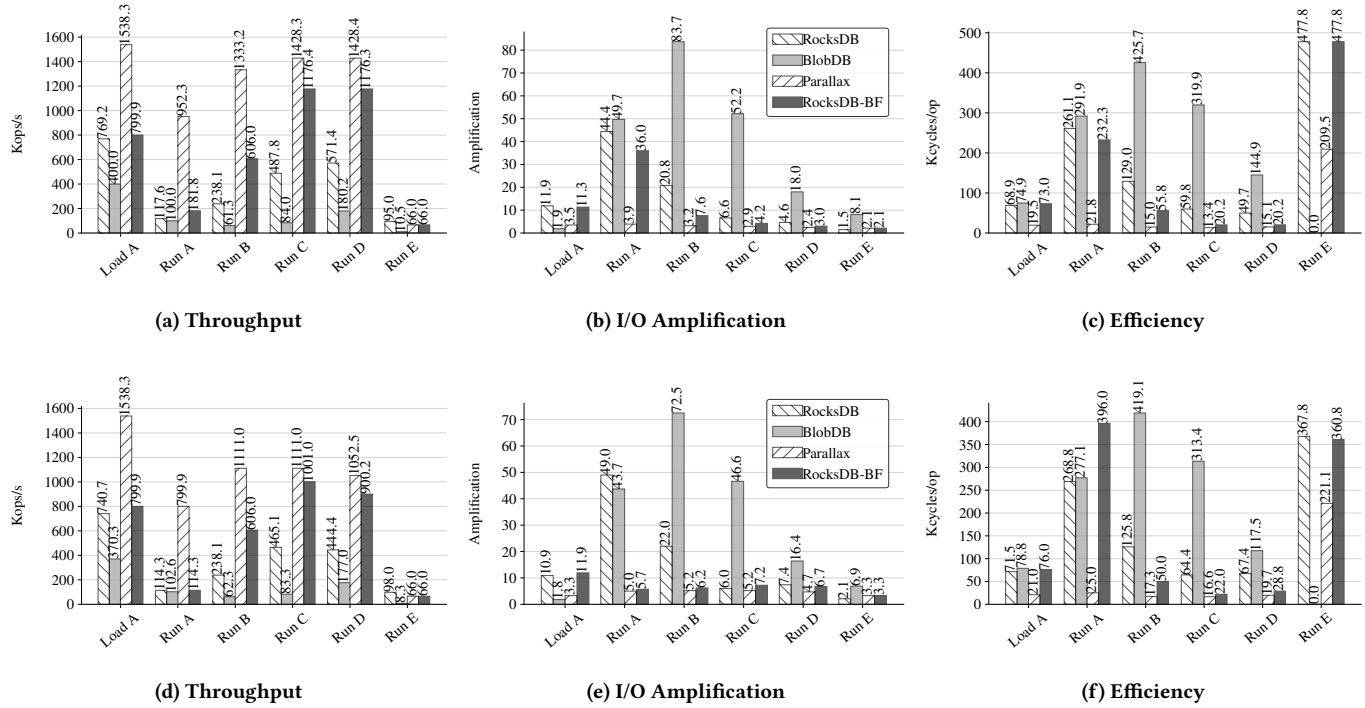
$CPU\_utilization$ is the average CPU utilization for all CPUs, excluding idle and I/O wait time, as given by *mpstat*. As $cycles/s$ we use the per-core clock frequency, $average\_ops/s$ is the throughput reported by YCSB, and $cores$ is the number of system cores including hyperthreads.

5 EXPERIMENTAL EVALUATION

In our evaluation we examine the following questions:

(1) What is the impact of hybrid KV placement in *Parallax*, compared to full in place and full KV separation?
(2) What is the benefit of using the medium category?
(3) What is the impact of merging medium KV pairs in place earlier than the last level?
(4) What is the impact of sorting log segments for medium KV pairs in L_0?

Impact of Hybrid KV Placement: First, Figure 4 shows results for all YCSB workloads for SD (top row) and MD (bottom row). We use these two workloads because they are more typical in modern applications [9, 43]. In addition to Load A and Run A, *Parallax* increases throughput, decreases I/O amplification, and increases efficiency for all workloads except for Run E, compared to both RocksDB and BlobDB.



(a) Throughput (b) I/O Amplification (c) Efficiency

(d) Throughput (e) I/O Amplification (f) Efficiency

Figure 4: Throughput (left), I/O amplification (middle), and efficiency (right) for all YCSB workloads for SD (top row) and MD (bottom row).

In both SD and MD, for Run B - Run D *Parallax* increases throughput by up to 5.6x and 61x, reduces I/O amplification by up to 6.5x and 26x, and increases efficiency by up to 8.6x and 28x. For calibration purposes we run a configuration of RocksDB with bloom filters enabled (RocksDB-BF, Figure 4). Bloom filters in RocksDB reduce I/O amplification from 1.5x to 11.2x across workloads. Although it is not meaningful to compare *Parallax* and BlobDB (which do not use bloom filters) directly to RockDB-BF, we believe that using bloom filters in these systems can result in similar benefits.

Run E consists mostly of scan (95% scans 5% inserts) operations and we discuss it separately. We run Run E both for SD and MD workloads. We run Load E with 100M operations and for Run E we run 20M operations. In SD, MD *Parallax* moves at least 50% the medium KVs from logs to in place in the index. Please note, we do not report in the Figures 5 efficiency for BlobDB as it is too high (3165 Kcycles/op) and about 8x worse than both *Parallax* and RocksDB. Comparing *Parallax* to RocksDB, RocksDB exhibits 1.43x (SD) and 1.48x (MD) higher throughput. It is important to note here that having all keys in place is the best organization for scan operations. However, this comes at a high cost (I/O amplification and CPU efficiency) for most other workloads. BlobDB has a throughput for SD, MD of 6.2, and 7.6 Kops/s, which makes KV separation impractical for such workloads. However, *Parallax* reduces this gap dramatically and within 36% (SD) and 39% (MD) of in place. Therefore, hybrid KV placement is a more practical approach compared to both in place organization and full KV separation, as it is able to handle well workloads with different mixes of KV pair sizes.

Next, Figure 5 shows in more detail the impact of hybrid KV placement compared to RocksDB and BlobDB using Load A and Run A for all six mixes of KV-pair sizes (Table 2). Note, that the GC cost for *Parallax* in Load A is almost zero due to its GC mechanism (no updates take place), while *Parallax* exhibits the full GC cost for Run A.

For Load A (Figure 5, top row), we see that in terms of operation throughput, *Parallax* exhibits up to 3.57x and 4.15x higher throughput compared to RocksDB and BlobDB, respectively. For all workloads except S (small KV pairs), *Parallax* reduces I/O amplification by up to 4.38x compared to RocksDB. Compared to BlobDB, *Parallax* exhibits up to 1.95x higher I/O amplification because BlobDB places all values in a log and never includes values in compactions. However, GC cost in BlobDB is high and makes BlobDB throughput significantly worse compared to *Parallax*. In terms of CPU efficiency we observe that *Parallax* is by up to 3.92x better than both systems, with RocksDB and BlobDB generally

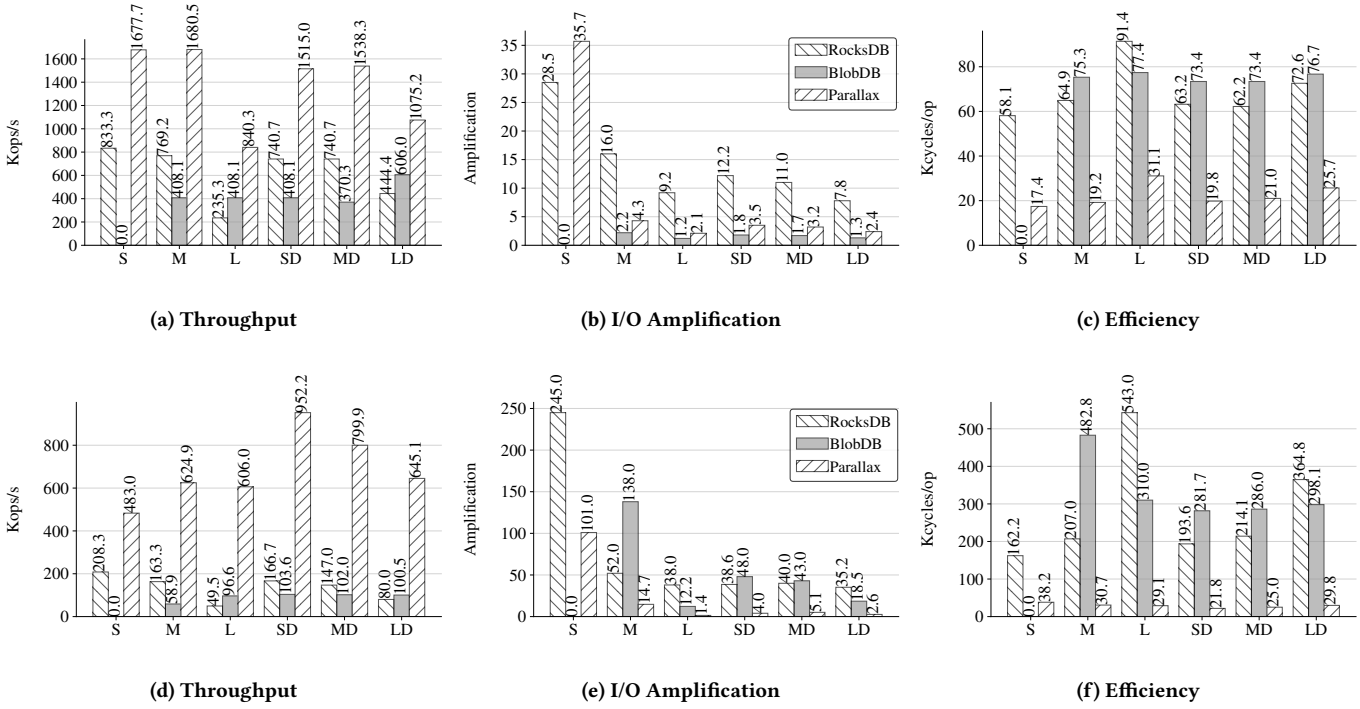


Figure 5: Throughput (left), I/O amplification (middle), and efficiency (right) for Load A (top row) and Run A (bottom row), for *Parallax* (hybrid), RocksDB (always in place), and BlobDB (always in log).

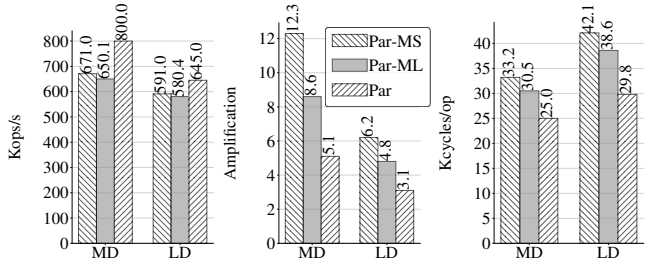
being relatively close and within 17% of each other. Please note that in the case of large only *Parallax* exhibits slightly higher I/O amplification compared to BlobDB (2.1 vs 1.2) due to its B+-tree index per level. For workload S, *Parallax* has 1.25x worse amplification because of the slot array in the B+-tree leaves. In particular, when KVs are small, the slot array accounts for 8% of the total leaf's capacity. Regarding throughput *Parallax* has 2x more throughput than RocksDB. We suspect this difference comes from 1) The difference of the in-memory component [3, 44] and 2) RocksDB has resilience mechanisms such as CRCs (which cannot be disabled). We leave this investigation for future work. However, in our evaluation, we focus mostly on amplification which mainly represents *Parallax* techniques.

For Run A (Figure 5), *Parallax* improves throughput, I/O amplification, and efficiency even further. Run A exhibits both lookup and cleanup costs for GC in the log for BlobDB, therefore, I/O amplification in BlobDB becomes even worse compared to *Parallax*. *Parallax* compared to RocksDB and BlobDB, increases throughput by up to 12.24x and 10.75x, reduces I/O amplification by up to 27.1x and 9.38x, and increases efficiency by up to 18.7x and 16x.

Benefits of introducing the medium category: Next, we examine if the complexity of introducing the medium category results in substantial benefits. Figure 6 shows throughput and I/O amplification for Run A for two *Parallax* configurations: moving medium KV pairs to the small category (*Parallax*-MS) and moving them to the large category (*Parallax*-ML). Essentially, these two configurations for *Parallax* correspond to setting the KV pair thresholds to $T_{SM} = T_{ML} = 0.02$, for *Parallax*-MS and to $T_{SM} = T_{ML} = 0.2$ for *Parallax*-ML.

We configure GC as we describe in Section 4. Also, the duration of the experiment is such that in all cases the GC threads process at least 97% of the system logs by the time the workload finishes. Figure 6 shows that for MD and LD using *Parallax* compared to *Parallax*-MS and *Parallax*-ML improves throughput by up to 1.23x and 1.11x respectively. Also, it reduces I/O amplification by up to 2.43x and 2x and increases efficiency by up to 1.32x and 1.41x. As expected, the difference is higher for MD, since in LD the large percentage of large KV pairs results in all three *Parallax* configurations placing most of the dataset in the log for large KV pairs.

Merging medium KV pairs in place earlier: *Parallax* can merge medium KV pairs in place at different levels of the LSM structure. Merging KV pairs later in the LSM structure results in lower I/O amplification but higher space amplification due to larger logs for medium KV pairs. Therefore, merging



(a) Throughput **(b) I/O Amplification** **(c) Efficiency**

Figure 6: Run A throughput, I/O amplification, and efficiency for three *Parallax* configurations: default (Par), MS, and ML. MS and ML use only two key categories, small and large. MS classifies medium keys as small, whereas ML classifies them as large.

medium KV pairs earlier limits the size of the medium log and thus reduces space amplification.

In terms of space amplification, our model (Figure 1(b)) shows that merging medium values in place in L_{n-2} vs L_{n-1}, reduces space amplification for a growth factor of 8 from about 13% to less than 3%. For a growth factor of 4, space amplification is reduced from 25% to 6%. In terms of I/O amplification, we observe that each level contributes equally to level amplification (one level less for compactions of medium KV pairs), therefore, moving medium values in place one level earlier increases level amplification by 1/N (N is the max number of levels based on storage capacity). However, total I/O amplification includes also merge amplification.

In Figure 7 we configure Parallax with growth factor 4 and set L_0 size to 128MB. We run Load A with 150M keys with workload M to create enough levels in the LSM tree and stress the system. In the runs with labels (N-1)/(N-2), Unsort (N-1)/(N-2), *Parallax* has transferred at least 90% of medium keys in place. Figure 7 quantifies the impact of merging medium KV pairs earlier. If we examine merging medium KV pairs (M workload - (N-1)/(N-2)) at levels L_{n-1} vs L_{n-2}, I/O amplification is 6.8 vs. 9.6 and throughput is 1579 Kops/s vs. 1339 Kops/s (Figure 7(a,b)). Therefore, a 16% improvement in throughput and 34% improvement in I/O amplification come at a cost of about 4x increase in space amplification (from 6% to 25% for growth factor 4 or from 13% to 3% for growth factor 8). Depending on the tradeoffs in specific setups, we believe that merging values at either of the last two levels can be a good approach. Especially, if scan performance is also important, then merging at L_{n-2} is a good tradeoff. As a reference point, we also include numbers for RocksDB and a non-achievable (ideal) baseline version of *Parallax*, NoMerge, that keeps medium KV pairs always in the log, never merges them in place, and performs no GC in the log.

Impact of sorting log segments in L_0: Figure 7 shows that in workload M, sorting segments (runs) in L_0 (when merging medium KV pairs at L_{n-1}), improves throughput by up to 2.63x, from 600 to 1578 Kops/s, and reduces I/O amplification by up to 4x (from 25.8 to 6.8). As a secondary observation, we note that if we choose to use unsorted segments, it is preferable to merge medium KV pairs at L_{n-2} instead of L_{n-1}. For workload MD, using sorted segments results also in higher throughput by up to 1.92x (merging at L_{n-1}), in higher efficiency by up to 1.2x, and 2.17x higher I/O amplification.

6 RELATED WORK

We group related work in the following categories: (a) hybrid KV placement techniques, (b) I/O amplification analysis in LSM-tree KV stores, and (c) GC for KV separation.

Parallax performs leveling for small KV pairs, KV separation for large pairs, and a hybrid technique for medium KV pairs. The purpose of handling KV pairs in this hybrid manner is to reduce I/O amplification, both for reorganizing data and GC purposes. In the spectrum of techniques between levelling and KV separation a main approach today is tiering [1, 13, 23, 28, 38], which performs less reorganization of KV pairs compared to levelling and more reorganization compared to KV separation. *Parallax* uses for medium KV pairs a technique that produces initial sorted runs but does not reorganize medium KV pairs further, therefore can be placed somewhere between pure tiering and pure KV separation. However, other design points may exist using different combinations of techniques.

DiffKV [31], similar to *Parallax*, uses three KV pair categories, based on their size. Similar to *Parallax*, they place small KV pairs in place and large values in a log. Unlike *Parallax* that uses a log for medium KV pairs, they perform tiering for medium KV pairs to avoid expensive compactions and they merge medium KV pairs lazily in the last two LSM levels. Their approach results in a higher degree of reorganization for medium KV pairs during tiering operations, whereas in *Parallax* medium KV pairs are written only once in the medium log. On the other hand, the DiffKV approach reduces memory requirements in the worst case during merging of medium KV pairs in the last LSM levels.

In terms of models for I/O amplification, Hyeontaek et al. [32] propose a simulation-based model that estimates write traffic for leveled LSM KV stores, given the ratio of insert, update, and delete operations. This model does not take into account designs that use KV separation. Although the model could be extended for our needs, we use an analytical approach [4] that allows for faster exploration of alternatives. Monkey [15] and Wacky [16] develop models

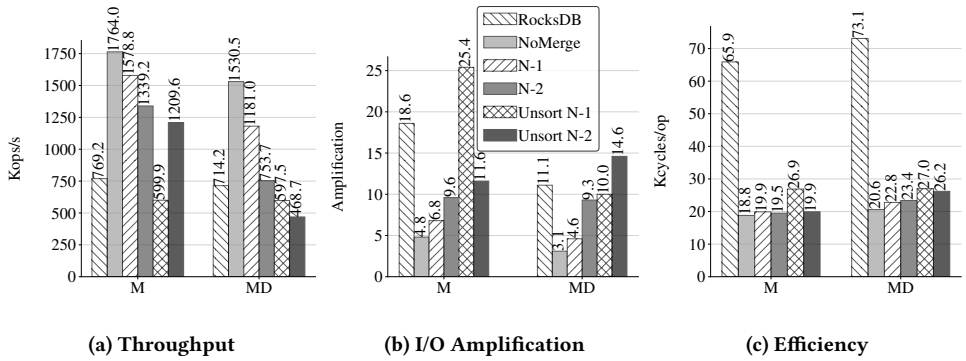


Figure 7: Impact of merging medium KV pairs earlier or using unsorted medium log runs in Load A, which exhibits more compactions.

for read and write operation cost for balancing the use of memory across different LSM functions or optimizing LSM organization by using hybrid level organization approaches. These models are not directly applicable to calculating I/O traffic for the purpose of hybrid value placement.

With respect to GC for KV separation, HashKV [10] deals with the high cost of GC in leveled LSM-type KV stores that use KV separation. HashKV reduces GC overhead for update intensive workloads with heavy zipfian distribution. It tries to identify hot keys (update-wise) and places them in a separate value log. As a result, most fragmented segments are found in this separate value log, which can then be reclaimed more efficiently by GC. *Parallax* is similar to HashKV in that both systems try to eliminate full scans over the value log. However, HashKV depends on the key distribution, whereas *Parallax* keeps additional metadata (per segment free space counters) and is agnostic to key distribution.

NovKV [42] eliminates costly lookups in the LSM index, required to identify invalid KV pairs in the log. It keeps a separate index of updated and deleted keys. During scanning of log segments, instead of looking up the regular index, it looks up each key in the separate index. If the separate index is small, then most lookups will be served in memory. NovKV still needs to scan the full log. Instead, *Parallax* avoids scanning the full log by using a separate index for the free space in each (large) log segment, which provides a list of eligible segments. During segment scan, *Parallax* performs lookups to the regular LSM index, however, only for large values. Therefore, the techniques used by *Parallax* to optimize full scans and by NovKV to optimize lookups are orthogonal.

7 CONCLUSIONS

In this paper, we design *Parallax*, a persistent LSM key value store for fast storage devices. *Parallax* first models and analyzes the benefits of using a log for KV separation. Based on this analysis it performs hybrid placement of KV pairs to reduce I/O amplification and reduce GC cost in the value log. *Parallax* always stores small KV pairs in place in each level index and large KV pairs in a value log, which uses a novel GC technique to avoid full log scans. For medium KV pairs, it uses a medium log until the last level(s) and then merges them in place, back to the LSM index to avoid performing GC in the medium log altogether. Compared to RocksDB, *Parallax* increases CPU efficiency by up to 18.7x and decreases I/O amplification by up to 27.1x. We believe that the hybrid placement approach of *Parallax* can be used in other systems as well to reduce I/O amplification and increase CPU efficiency for medium and large KV pairs.

ACKNOWLEDGMENTS

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the project EVOLVE (Grant Agreement ID: 825061). During the course of the work Anastasios Papagiannis was also supported by the Facebook Graduate Fellowship. Finally, we thank the anonymous reviewers for their insightful comments and our shepherd Asaf Cidon for his help with preparing the final version of the paper.

REFERENCES

[1] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. 2019. Jungle: Towards Dynamically Adjustable Key-Value Store by Combining LSM-Tree and Copy-on-Write B+-Tree. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems* (Renton, WA, USA) *(HotStorage '19)*. USENIX Association, USA, 9.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer*

Systems (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64.

[3] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 80–94.

[4] Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. 2020. VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. arXiv:2003.00103 [cs.DC]

[5] Rudolf Bayer and Edward McCreight. 2002. *Organization and maintenance of large ordered indexes*. Springer.

[6] R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-trees. *Acta Inf.* 9, 1 (March 1977), 1–21.

[7] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. 2001. Heuristic Cleaning Algorithms in Log-Structured File Systems. (05 2001).

[8] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. 2001. Main-memory Index Structures with Fixed-size Partial Keys. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. ACM, New York, NY, USA, 163–174.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST '16)*. USENIX Association, Santa Clara, CA, 209–223.

[10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 1007–1019.

[11] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. UTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* 13, 12 (July 2020), 2634–2648.

[12] Howard Chu. 2011. MDB: A memory-mapped database and backend for OpenLDAP. In *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*. 35.

[13] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, 49–63.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[16] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush and the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 449–466. https://doi.org/10.1145/3299869.3319903

[17] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[18] Jason Evans. 2018. jemalloc. http://jemalloc.net/.

[19] Facebook. 2011. RocksDB. http://rocksdb.org/.

[20] Facebook. 2018. BlobDB. http://rocksdb.org/. Accessed: October 3, 2021.

[21] Facebook. 2018. RocksDB WAL Performance. https://github.com/facebook/rocksdb/wiki/WAL-Performance. Accessed: October 3, 2021.

[22] Facebook. 2018. RocksDB WAL Performance. https://github.com/facebook/rocksdb/wiki/Direct-IO. Accessed: October 3, 2021.

[23] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: Optimizing Key-Value Storage for Spatial Locality. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 27, 16 pages.

[24] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[25] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981–992.

[26] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665.

[27] INTEL. 2017. OPTANE SSD DC P4800X SERIES. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html. Accessed: October 3, 2021.

[28] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 16–25.

[29] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data.. In *MSST*. IEEE Computer Society, 1–14.

[30] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461.

[31] Yongkun Li, Zhen Liu, Patrick P. C. 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)*. USENIX Association, 673–687.

[32] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST '16)*. USENIX Association, USA, 149–166.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*. USENIX Association, Santa Clara, CA, 133–148.

[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (06 1996), 351–385.

[35] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. Memory-Mapped I/O on Steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom)

(EuroSys '21). Association for Computing Machinery, New York, NY, USA, 277–293.

[36] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Denver, CO, 537–550.

[37] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 490–502. https://doi.org/10.1145/3267809.3267824

[38] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 497–514.

[39] Jinglei Ren. 2016. YCSB-C. https://github.com/basicthinker/YCSB-C.

[40] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 1–15.

[41] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 217–228.

[42] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. [n.d.]. NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores. ([n. d.]).

[43] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 191–208.

[44] Jeseong Yeon, Leeju Kim, Youil Han, Hyeon Gyu Lee, Eunji Lee, and Bryan S. Kim. 2020. JellyFish: A Fast Skip List with MVCC. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 134–148.