Building an Efficient Key-Value Store in a Flexible Address Space

Chen Chen
University of Illinois at Chicago
cchen262@uic.edu

Wenshao Zhong
University of Illinois at Chicago
wzhong20@uic.edu

Xingbo Wu
Microsoft Research
xingbowu@microsoft.com

Abstract

Data management applications store their data using structured files in which data are usually sorted to serve indexing and queries. However, in-place insertions and removals of data are not naturally supported in a file's address space. To avoid repeatedly rewriting existing data in a sorted file to admit changes in place, applications usually employ extra layers of indirections, such as mapping tables and logs, to admit changes out of place. However, this approach leads to increased access cost and excessive complexity.

This paper presents a novel storage abstraction that provides a *flexible address space*, where in-place updates of arbitrary-sized data, such as insertions and removals, can be performed efficiently. With these mechanisms, applications can manage sorted data in a linear address space with minimal complexity. Extensive evaluations show that a key-value store built on top of it can achieve high performance and efficiency with a simple implementation.

CCS Concepts: • **Information systems → Data structures**; **Database management system engines**.

Keywords: Address Space, Storage, Key-Value Store

ACM Reference Format:
Chen Chen, Wenshao Zhong, and Xingbo Wu. 2022. Building an Efficient Key-Value Store in a Flexible Address Space. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519555

1 Introduction

Data management applications store data in files for persistent storage. The data are usually sorted in a specific order so that they can be correctly and efficiently retrieved in the future. However, it is not trivial to make updates such as insertions and deletions in these files. To commit in-place updates in a sorted file, existing data may need to be rewritten to maintain the file's layout. For example, key-value (KV) stores such as LevelDB [23] and RocksDB [21] need to merge and sort KV pairs in their data files periodically, causing repeated rewriting of existing KV data [26, 38, 46].

It has been conventional wisdom to rewrite data to keep data sorted and gain a better access locality. By co-locating logically adjacent data in the storage device, the data can be quickly accessed in the future with a minimal number of I/O requests, which is crucial for traditional storage technologies such as HDDs. However, when managing data with new storage technologies that provide more balanced random and sequential I/O performance (e.g., Intel's Optane SSDs [29]), the access locality is less of a dominant performance factor [60]. In this scenario, data rewriting becomes less beneficial for future accesses but still consumes enormous CPU and I/O resources [36, 44]. Therefore, it may not be cost-effective to rewrite data on these devices in exchange for a better locality. Despite this, data management applications still need to keep their data logically sorted for efficient access. An intuitive solution is to relocate data in the address space logically without physically rewriting them. However, this is barely feasible because of the lack of support for logically relocating data in a file's address space.

In practice, applications pay a high cost to keep data sorted by using extra indirections. For example, using a B^+-Tree to index data needs to rewrite tree nodes on updates. LSM-Trees rewrite data less aggressively by using a multi-level layout, which slows down reads due to sort-merging data on the fly [69]. Additionally, committing changes to these structures requires extra mechanisms such as barriers and flushes, which inflates the cost of maintaining crash consistency, leading to problems like redundant journaling [55, 63]. If the storage layer can provide support for keeping data logically sorted, applications can delegate the data organizing jobs to the storage layer, instead of employing extra persistent indirections at the application level. To achieve this goal, the storage layer can provide a *flexible address space* that supports in-place data insertions and removals, so that the data can be easily sorted.

Much effort has been made toward this direction. For example, a few popular file systems—Ext4, XFS, and F2FS—have provided *insert-range* and *collapse-range* features for

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519555

inserting or removing a range in a file's address space to support various types of applications [22, 28]. However, these mechanisms have not been able to help applications because of a few fundamental limitations. First of all, they have rigid block-alignment requirements. For example, inserting a record of a few bytes into a sorted data file using the *insert-range* operation is not allowed. Second, *shifting* a range of address mappings is very inefficient with conventional address space indexes. Inserting an aligned data segment to a file needs to shift all the existing address mappings after the insertion point to make room for the new data. The shift operation has $O(N)$ cost (N is the number of extents or blocks in the file), which can be very costly due to intensive metadata updates and journaling. Third, commonly used data indexing mechanisms cannot keep track of shifted contents in an address space. For example, indexes using offsets to locate data are no longer usable because the offsets can be easily changed by a shift operation. Therefore, a co-design of applications and the storage layer is necessary to realize the benefits of managing data in a flexible address space.

This paper introduces FlexSpace, a storage engine that provides a *persistent flexible address space* for data management applications. The core of FlexSpace is an address space indexing structure, named FlexTree, that is derived from the B$^+$-Tree structure. In a FlexTree, it takes $O(\log N)$ time to perform a shift operation in the address space, which is asymptotically faster than that of existing index data structures with $O(N)$ cost. We implement FlexSpace as a user-space library. It adopts log-structured space management for write efficiency and performs defragmentation based on data access locality for cost-effectiveness. It also employs logical logging [50, 67] to commit metadata updates at low cost.

We build FlexDB, a KV store that demonstrates how to implement efficient data management applications in a flexible address space. Based on the advanced features provided by FlexSpace, FlexDB is able to maintain a fully sorted order of all KV pairs in a persistent flexible address space without employing complex indirections or intensively rewriting existing data. In the meantime, it has a simple structure and a small codebase. That said, FlexDB is a fully functional KV store that supports regular point and range query operations. FlexDB also integrates efficient mechanisms to support caching, concurrent access, and crash consistency. Evaluation results show that FlexDB has substantially reduced the data rewriting overheads. It achieves up to 16× and 3.3× speed-ups for read and write operations, respectively, compared to two I/O-optimized KV stores, RocksDB and KVell.

This paper makes three major contributions. First, we introduce an address space indexing structure, namely FlexTree, that enables efficient shift operations (§3). Second, we build FlexSpace to realize a persistent flexible address space, in which data management applications can perform high-speed in-place data insertions and removals (§4). Third, we use FlexDB to demonstrate a performant KV store that can be easily built based on a flexible address space (§5). Furthermore, we thoroughly evaluate the efficacy of a flexible address space and its usage for data management (§6).

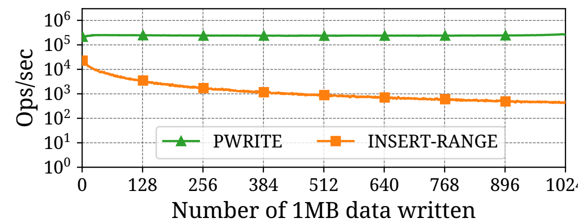


Figure 1. Performance of random write/insert on Ext4

2 Limitations of File Address Spaces

Modern file systems use extents to manage file address mappings. An extent is a group of contiguous blocks. Its metadata consists of three essential elements—file offset, length, and block number. Real-world file systems employ index structures to manage extents. For example, Ext4 uses an HTree [19]. Btrfs and XFS use a B$^+$-Tree [51, 59]. F2FS uses a multi-level mapping table [34].

Regular file operations such as overwrite do not modify existing mappings. An append-write to a file needs to expand the last extent in place or add new extents to the end of the mapping index, which is of low cost. However, the *insert-range* and *collapse-range* operations in the aforementioned data structures can be very expensive due to the shifting of extents. To be specific, an *insert-range* or *collapse-range* operation needs to update the offset value of every extent after the insertion or removal point. Therefore, the shift operation has $O(N)$ cost, where N is the total number of extents after the insertion or removal point.

We benchmark the file editing performance of an Ext4 file system on an Intel Optane 905P SSD. There are two write patterns, namely, PWRITE and INSERT-RANGE. PWRITE starts with an empty file and uses the `pwrite` system call to fill a 1 GB space with 4 KB blocks in random order without overwrites. INSERT-RANGE starts with an empty file and inserts 4KB data blocks to random 4K-aligned offsets by shifting existing file data forward until the file size reaches 1 GB. Accordingly, each insertion shifts the data after the insertion point forward. The experimental results are shown in Figure 1. The throughput of INSERT-RANGE dropped quickly and was eventually nearly 1000× lower than that of PWRITE. Although INSERT-RANGE does not rewrite any user data, it updates the metadata intensively and caused 25% more writes to the SSD compared to PWRITE. This number can be further increased if the application frequently calls `fsync` to enforce write ordering. XFS and F2FS also support the shift operations, but they exhibit much worse performance than Ext4, so their results are not included.

Extents are simple and flexible for managing variable-length address mappings. However, the alignment requirements and the inefficient extent index structures in today's file address spaces hinder the adoption of in-place data insertions and removals. To make a flexible address space generally usable and affordable for data management applications, an efficient mechanism that supports data shifting without rigid alignment requirements is indispensable.

3 FlexTree

Inserting or removing data in a file needs to shift all the existing data beyond the insertion or removal point, which causes intensive updates to the metadata of the affected extents. With regard to the number of extents in a file, the cost of shift operations can be prohibitively high due to the $O(N)$ complexity in existing extent index structures.

The following introduces FlexTree, an augmented B$^+$-Tree that supports efficient shift operations. The design of FlexTree is based on the observation that a shift operation alters a contiguous range of extents. FlexTree treats the shifted extents as a whole and applies the updates to them collectively. To facilitate this, it employs a new metadata representation scheme that stores the address information of an extent on its search path. As an extent index, it costs $O(\log N)$ time to perform a shift operation in FlexTree, and a shift operation only needs to update a few tree nodes.

3.1 The Structure of FlexTree

Before demonstrating the design of FlexTree, we first start with an example of B$^+$-Tree [12] that manages an address space in byte granularity (Figure 2a). Each extent corresponds to a leaf-node entry consisting of three elements—*offset*, *length*, and (physical) *address*. Each internal node contains *pivot* entries that separate the pointers to the child nodes. When inserting a new extent at the head of an address space, every existing extent's offset and every pivot's offset must be updated because of the shift operation on the entire address space.

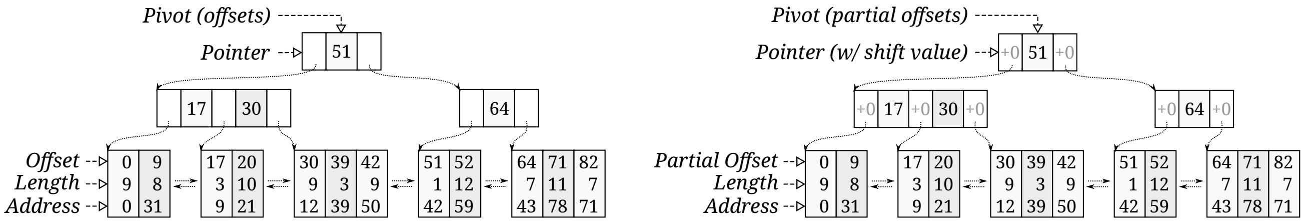
FlexTree employs an address metadata representation scheme that allows for shifting extents with substantially reduced changes. Figure 2b shows a FlexTree that encodes the same address mappings in the B$^+$-tree. In FlexTree, the offset fields in extent entries and pivot entries are replaced by *partial offset* fields. Besides, the only structural difference is that in a FlexTree, every pointer to a child node is associated with a *shift* value. These shift values are used for encoding address information in cooperation with the partial offsets. The effective offset of an extent or pivot entry is determined by the sum of the entry's partial offset and the shift values of the pointers found on the search path from the root node to the entry. The search path from the root node (at level 0) to an entry at level N can be represented by a sequence $\big((X_0, S_0), (X_1, S_1), \ldots, (X_{N-1}, S_{N-1})\big)$, where X_i represents the index of the pointer at level i, and S_i represents the shift value associated with that pointer. The partial offset of an entry is P. Its effective offset E can be calculated by $E = \big(\sum_{i=0}^{N-1} S_i\big) + P$.

3.2 FlexTree Operations

FlexTree supports basic extent operations such as appending extents at the end of an address space and remapping existing extents, as well as advanced operations, including inserting or removing extents in the middle of an address space (*insert-range* and *collapse-range*). The following explains how the address range operations execute in a FlexTree. In this section, a leaf node entry in FlexTree is denoted by a triple: *(partial_offset, length, address).*

The insert-range Operation Inserting a new extent of length L to a leaf node z in a FlexTree takes three steps. First, the operation searches for the leaf node and inserts a new entry with a partial offset $P = E - \big(\sum_{i=0}^{N-1} S_i\big)$, assuming the leaf node is not full. When inserting to the middle of an existing extent, the extent must be split before the insertion. The insertion requires a shift operation on all the extents after the new extent. In the second step, for each extent within node z that needs shifting, its partial offset is incremented by L. The remaining extents that need shifting span all the leaf nodes after node z. We observe that, if every extent within a subtree needs to be shifted, the shift value can be recorded in the pointer that points to the root of the subtree. Therefore, in the third step, the remaining extents are shifted as a whole by updating a minimum number of pointers to a few subtrees that cover the entire range. To this end, for each ancestor node of z at level i, the shift values of the pointers and the partial offsets of the pivots after the pointer at X_i are all added by L. In this process, the updated pointers cover all the remaining extents, and the path of each remaining extent contains exactly one updated pointer. When the update is finished, every shifted extent has its effective offset added by L. The number of updated nodes of a shift operation is bounded by the tree's height, so the operation's cost is $O(\log N)$.

Figure 3 shows the process of inserting a new extent with length 3 and physical address 89 to offset 0 in the FlexTree shown in Figure 2b. The first step is to search for the target leaf node for insertion. Because all the shift values of the pointers are 0, the effective offset of every entry is equal to its partial offset. Therefore, the target leaf node is the leftmost one, and the new extent should be inserted at the beginning of that leaf node. Then, there are three changes to be made to the FlexTree. First, a new entry $(0, 3, 89)$ is inserted at the beginning of the target leaf node. Second, the other two extents in the same leaf node are updated from $(0, 9, 0)$ and $(9, 8, 31)$ to $(3, 9, 0)$ and $(12, 8, 31)$, respectively. Third, following the target leaf node's path upward, the pointers to the three subtrees covering the remaining leaf nodes and the



(a) B+-Tree **(b)** FlexTree

Figure 2. Examples of B+-Tree and FlexTree that manage the same address space

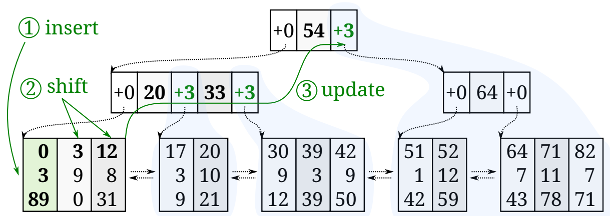


Figure 3. Inserting a new extent in FlexTree

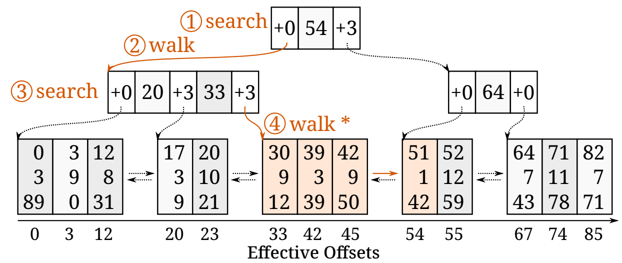


Figure 5. Looking up mappings from 36 to 55 in FlexTree

corresponding pivots are updated, as shown in the shaded areas in Figure 3. Now, the effective offset of every existing leaf entry is increased by 3.

FlexTree splits every full node when a search travels down the tree for insertion. The split threshold is set to one entry smaller than the node's capacity because an insertion may cause an extent to be split, which leads to two entries being added to the node for the insertion. To split a node, half the entries in the node are moved to a new node. Meanwhile, a pointer to the new node and a new pivot entry is created at the parent node. The new pointer inherits the shift value of the pointer to the old node so that the effective offsets of the moved entries remain unchanged. The new pivot entry inherits the effective offset of the median key in the old full node. The partial offset of the new pivot is calculated as the sum of the old median key's partial offset and the new pointer's inherited shift value. Figure 4 shows an example of a split operation. The new pivot's partial offset is 38 (which is $5 + 33$).

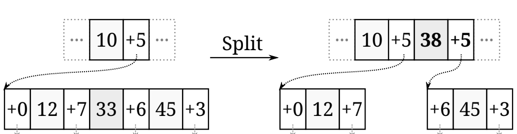


Figure 4. An example of node splitting in FlexTree

Querying Mappings of an Address Range To retrieve the mappings of an address range in FlexTree, the operation first searches for the starting point of the range, which is a byte address within an extent. Then, it scans forward on the leaf level from the starting point to retrieve all the mappings in the requested range. The correctness of the forward scanning is guaranteed by the assumption that all extents on the leaf level are contiguous in the logical address space. Apparently, a hole (an unmapped address range) in the logical address space can break the continuity and lead to incorrect range size calculation and wrong search results. To address this issue, FlexTree explicitly records holes as unmapped ranges using extents with a special physical address value that has all the bits set to one. When an unmapped range is split due to an insertion or write in it, the resulting unmapped extents, if any, still hold the special physical address value.

To query the address mappings from 36 to 55 in the FlexTree shown in Figure 5, a search of logical offset 36 first identifies the third leaf node. The partial offset values of the pivots in the internal nodes on the path are equal to their effective offsets (54 and 33), and the target leaf node has the path $\big((0, +0), (2, +3)\big)$. The starting point (logical offset 36) is the fourth byte within the first extent in the leaf node. Then the address mappings of the 19-byte range can be retrieved by scanning the leaf nodes from that point. The result is $\big((15, 6), (39, 3), (50, 9), (42, 1)\big)$, an array of four tuples, each containing a physical address and a length.

The collapse-range Operation To collapse (remove without leaving a hole) an address range in FlexTree, the operation first searches for the starting point of the removal. If the range starts in the middle of an extent, the extent is split so that the removal will start from the beginning of an extent. Similarly, a split is also used when the range ends in the middle of an extent. The address range being removed will cover one or multiple extents. For each extent in the range, the extents after it are shifted backward using a process similar to the forward shifting in the insertion operation. The only difference is that a negative shift value is used.

Figure 6 shows the process of removing a 9-byte address range (33 to 42) from the FlexTree in Figure 5 without leaving a hole in the address space. First, a search identifies the starting point, which is the beginning of the first extent (30, 9, 12) in the third leaf node. Then the extent is removed, and the remaining extents in the leaf node are shifted backward. Finally, in the root node, the pointer to the subtree that covers the last two leaf nodes is updated with a negative shift value of −9, as shown in the shaded area in Figure 6.

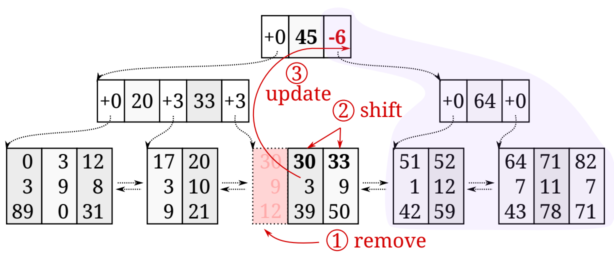


Figure 6. Removing address mapping from offset 33 to 42

FlexTree merges a node to a sibling if their total size is under a threshold after a removal. Since two nodes being merged can have different shift values in their parents' pointers, we need to adjust the partial offsets in the merged node to maintain correct effective offsets for all the entries. When merging two internal nodes, the shift values are also adjusted accordingly. Figure 7 shows an example of merging two internal nodes.

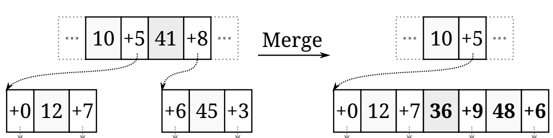


Figure 7. An example of node merging in FlexTree

3.3 Implementation

FlexTree manages extent address mappings in byte granularity. To be specific, the size of an extent can be an arbitrary number of bytes. In the implementation of FlexTree, the internal nodes have 64-bit shift values and pivots. For leaf nodes, we use 32-bit lengths, 48-bit partial offsets, and 48-bit physical addresses for extents. The largest physical address value ($2^{48} - 1$) is reserved for unmapped address ranges.

An effective offset can address a 64-bit space using the sum of 64-bit shift values and a 48-bit partial offset. When a leaf node's maximum partial offset becomes too large, to avoid overflow, FlexTree subtracts a value M, which is the minimum partial offset in the node, from every partial offset of the node, and adds M to the node's corresponding shift value in the parent node. Within a leaf node, the extents can cover up to 256 TB, which is sufficiently large in practice.

In the next section, we will introduce how we use FlexTree to manage data in a persistent address space correctly and efficiently

4 FlexSpace

FlexSpace is a storage engine that provides persistent data storage in a flexible address space. With FlexSpace, applications can make a better tradeoff by leveraging the lightweight in-place insertion/removal operations to manage sorted data without using extra indirections or repeated data rewriting.

We implement FlexSpace as a user-space library. It supports common file operations such as `read`, `write`, `pread`, and `pwrite`. It also provides advanced `insert_range` and `collapse_range` APIs for in-place data insertions and removals. The library enables concurrent access to individual address spaces using reader-writer locks. It does not employ automated readahead since I/O efficiency is often better exploited from the application level [32, 33, 36].

Internally, a FlexSpace's data and metadata are stored in regular files in a traditional file system. Each FlexSpace consists of three files—a data file, a FlexTree file, and a logical log file. The user-space library implementation gives FlexSpace the flexibility to perform byte-granularity space management without any block alignment limitations. In the meantime, the FlexSpace library delegates the job of cache management to the operating system.

4.1 Space Management

A FlexSpace stores its data in a data file. The data file's space is divided into fixed-size segments, which is similar to the structures in log-structured storage systems [34, 52, 53]. Each segment is 4 MB in our implementation because this size provides a good balance between space allocation complexity and garbage collection cost (discussed later). Each new extent is allocated within a segment. Specifically, a large write operation may create multiple logically contiguous extents residing in different segments. To avoid small writes, an in-memory segment buffer is maintained, where consecutive extents are automatically merged if they are logically contiguous. Only one segment is buffered in memory at any time.

The FlexSpace library performs garbage collection (GC) to reclaim space from underutilized segments. It maintains an in-memory array to record the valid data size of each segment. A GC process scans the array to identify a set of most underutilized segments and relocates all the valid extents from these segments to new segments. Then, the FlexTree extent index is updated accordingly. Since the extents in a FlexSpace can have arbitrary sizes, the GC process may produce less free space than expected because of the internal fragmentation in each segment. To address this issue, we adopt an approach used by a log-structured memory allocator [53] to guarantee that a GC process can always make forward progress.

By limiting the maximum extent size to $\frac{1}{K}$ of the segment size, relocating extents in one segment whose utilization ratio is not higher than $\frac{K-1}{K}$ can reclaim free space for at least one new extent. Therefore, if the space utilization ratio of the data file is capped at $\frac{K-1}{K}$, the GC can always reclaim space from the most underutilized segment for writing new extents. In the implementation, we set the maximum extent size to be $\frac{1}{32}$ (128 KB) of the segment size and conservatively limit the space utilization ratio of the data file to $\frac{30}{32}$ (93.75%). In addition, we reserve at least 64 free segments for relocating extents in batches. The FlexSpace library also provides a `flexspace_defrag` interface for manually relocating a range of data in the file into new segments. We will evaluate the efficiency of the GC policy in §6.

4.2 Metadata Management and Consistency

FlexSpace accesses the FlexTree file by having it memory-mapped into the user space (using `mmap`). Therefore, tree nodes can be loaded and written back on demand. FlexSpace must ensure atomicity and crash consistency when it synchronizes updates to the FlexTree file. An insertion or removal operation often updates multiple tree nodes along the search path in the FlexTree. If we use a per-node journaling mechanism to commit updates in the FlexTree file, every dirtied node in the FlexTree will be written twice. To address the potential performance issue, we use a combination of Copy-on-Write (CoW) [50] and logical logging [50, 67] to minimize the I/O cost.

CoW Before updating a tree node, FlexSpace creates a copy of the node in a new location in the file, and performs updates in the new copy. The FlexTree file has a header at the beginning that contains a version number and a root node position. A commit to the FlexTree file creates a new version of the FlexTree in the file. In the commit process, updated nodes are written to their new locations (via `msync`) in the FlexTree file without rewriting existing nodes. Once all the updated nodes have been written, the file's header is updated atomically to make the new version persist. When the new version has been committed, the file space used by the updated nodes in the old version can be safely reused. If a system crash happens during the commit process, the file header still points to the old version that is intact. Meanwhile, all updates are abandoned.

Logical Logging Updates to the FlexTree extent index can be intensive with small insertions and removals. If every metadata update directly commits to the FlexTree file and creates a new version of the FlexTree, the I/O cost can be high because every commit can update multiple tree nodes in the FlexTree file. The FlexSpace library adopts the logical logging mechanism [50, 67] to further reduce the metadata I/O cost. For every update in the FlexTree, the FlexSpace library performs the CoW updates to the tree in the mapped memory, but does not create a new header or synchronize the updated pages to the backing file. Meanwhile, a record of the index operation is written to a logical log file. Only when the log has accumulated a sufficient amount of updates, FlexSpace synchronizes all the (dirty) mapped pages to the backing FlexTree file, followed by creating a new FlexTree header to commit a new version.

A log entry for an insertion or removal operation contains the logical offset, length, and physical address of the operation. A log entry for a GC relocation contains the old and new physical addresses and the length of the relocated extent. Each log entry takes 24 bytes of space (including 2 bits for the operation type), which is much smaller than the FlexTree node size. The version number of the persistent FlexTree is recorded at the head of the log. Upon a crash-restart, uncommitted updates to the persistent FlexTree can be recovered by replaying the log on the persistent FlexTree. Note that the logical log will be read only when recovering from a system crash during FlexTree synchronization. On regular metadata retrievals, FlexSpace only accesses the memory-mapped FlexTree file.

Write Ordering When writing data to a FlexSpace, the data are first written to free segments in the data file. Then, the metadata updates are applied to the memory-mapped FlexTree file using CoW and recorded in an in-memory buffer of the logical log. The buffered log entries are committed to the log file periodically or on-demand for persistence. Particularly, the buffered log entries are committed after every execution of the GC process to make sure that the new positions of the relocated extents are persistently recorded. Then, the reclaimed space can be safely reused. Upon a commit to the log file, the data file must be first synchronized so that the logged operations will refer to correct file data. When the logical log file size reaches a pre-defined threshold, or the FlexSpace is being closed, the updated FlexTree nodes are committed to the FlexTree file, and a new version of the FlexTree is created. Afterward, the log file can be truncated and reinitialized using the FlexTree's new version number.

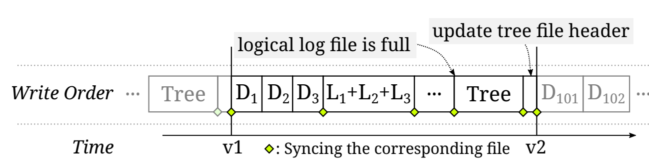


Figure 8. An example of write ordering in FlexSpace

Figure 8 demonstrates the write ordering of a FlexSpace with an example. D_i and L_i represent the data write and the logical log write for the i-th file operation, respectively. At the time of "v1", there are no uncommitted updates to the FlexTree file. Meanwhile, the log file is almost empty, containing only the FlexTree version (version 1). Then, for each write operation, the data is written to the data file (or buffered if the data is small), and its corresponding metadata updates are logged in the logical log buffer. When the logical log buffer is full, all the file data (D_1, D_2, and D_3) are synchronized to the data file. Then the buffered log entries ($L_1 + L_2 + L_3$) are written to the logical log file. When the log file is full, the updated FlexTree nodes are synchronized to the FlexTree file ($Tree$). Afterward, a new FlexTree file header is written atomically to create the new version (version 2). The logical log is then cleared for recording future operations based on the new version. I/O barriers (`fsync` or `msync`) are used before and after each logical log file commit and each FlexTree file header update to enforce write ordering, as shown in Figure 8.

Crash Consistency Following the write ordering introduced above, FlexSpace can always maintain a consistent state when a system crash occurs. We justify this by taking Figure 8 as an example. If the system crashes during a data file I/O (e.g., D_1, D_2 or D_3), no metadata has been flushed from the in-memory log buffer, so these data will be abandoned and their space will be reclaimed after a crash recovery. If the system crashes during a logical log write (e.g., $L_1 + L_2 + L_3$), the log entries can be partially written. However, since the logs are flushed in order and all their corresponding data have been written (D_1, D_2 and D_3), redoing the consecutively valid log entries can recover FlexSpace to a consistent state. If the system crashes during committing the updated FlexTree nodes to the on-disk FlexTree file, this process can always restart from the beginning because all the logical logs are written and the old version of the FlexTree is accessible after restarting. This is also the case when the system crashes during updating the tree file header.

5 FlexDB

We build FlexDB, a KV store powered by the advanced features of FlexSpace. Just like the popular LSM-tree KV stores, LevelDB [23] and RocksDB [21], FlexDB buffers updates in a MemTable and writes to a write-ahead log (WAL) for immediate data persistence. When committing updates to the persistent storage, however, FlexDB adopts a greatly simplified data model. FlexDB stores all the KV pairs in sorted order in a FlexSpace, without using any persistent indirections. Instead of performing repeated compactions across a multi-level store hierarchy that causes high write amplification, FlexDB directly commits updates from the MemTable to the FlexSpace in place at low cost. FlexDB employs a space-efficient volatile sparse index to track positions of persistent KV data in the FlexSpace and implements user-space caching for fast reads.

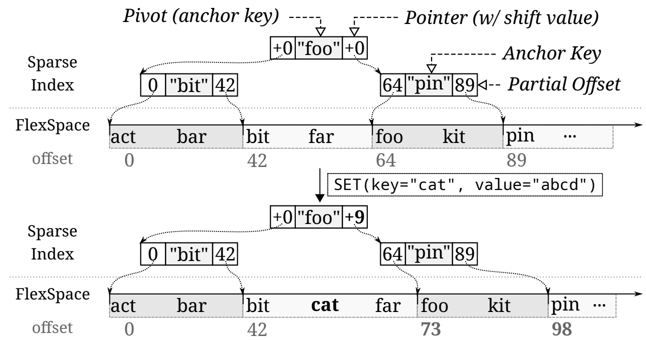


Figure 9. An example of the sparse KV index in FlexDB

5.1 Managing KV Data in a FlexSpace

FlexDB stores persistent KV pairs in a FlexSpace and keeps them always sorted (in lexical order by default) with in-place updates. Each KV pair in the FlexSpace starts with the key and value lengths encoded with Base-128 Varint [8], followed by the key and value's raw data. A sparse KV index, whose structure is similar to a B$^+$-Tree, is maintained in the memory to enable fast search in the FlexSpace.

KV pairs in the FlexSpace are grouped into *intervals*, each covering a number of consecutive KV pairs. The sparse index stores an entry for each interval using the smallest key in it as the index key. The entry also records the size of the interval. As with FlexTree, the sparse index encodes the offset of an interval using the partial offset and the shift values on its search path. Specifically, each leaf node entry contains a *partial offset*, and each child pointer in internal nodes records a *shift* value. The effective offset of an interval is the sum of its partial offset and the shift values on its search path. A search of a key performs a binary search on the sparse index and calculates the effective offset of the interval. Then, the search scans the interval to find the KV pair. Figure 9 shows an example of the sparse KV index with four intervals. The first interval does not need an index key. The index keys of the other three intervals are "bit", "foo" and "pin", respectively. A search of "kit" reaches the third interval ("foo" < "kit" < "pin") at offset 64 (0+64).

When inserting (or removing) a KV pair in an interval, the offsets of all the intervals after it need to be shifted so that the index can stay in sync with the FlexSpace. The shift operation is similar to that in a FlexTree. First, the operation updates the partial offsets of the intervals in the same leaf node. Then, the shift values on the path to the target leaf node are updated. Unlike in FlexTree, the partial offsets in the sparse KV index are not the search keys but the values in leaf node entries. Therefore, the shift operation does not modify any index keys or pivots. An update operation that resizes a KV pair is performed by removing the old KV pair and inserting the new one at the same offset.

To insert a new KV pair ("cat", "abcd") in the FlexDB shown in Figure 9, a search first identifies the interval at offset 42 whose index key is "bit". Assuming the new KV item's size is 9 bytes, we insert it to the FlexSpace between keys "bit" and "far" and shift the intervals after it forward by 9. As shown at the bottom of Figure 9, the effective offsets of the last two intervals are incremented by 9.

The sparse index needs to split a large interval or merge two small intervals when their sizes reach specific thresholds. The thresholds are specified by the total data size in bytes and the number of KV pairs. In the implementation, the split threshold is defined as 16 KB and 16 KV items, whichever is exceeded first. Two intervals can be merged if the total size is less than 16 KB and they contain less than 16 KV items. These threshold values are chosen because they show the most balanced performance on our testbed. They can be adjusted based on the actual system configuration to gain better performance when FlexDB is deployed.

5.2 Interval Caching

Real-world workloads often exhibit skewed access patterns [1, 5, 64]. Many popular KV stores employ user-space caching to exploit the access locality for improved search efficiency [9, 21, 24]. FlexDB adopts a similar approach by implementing an *interval cache* to store frequently used intervals in the main memory. Every interval's entry in the sparse index contains a cache pointer that is initialized as `NULL` to represent an uncached interval. Upon a cache miss, a new cache entry is allocated by creating an array of KV pairs based on the interval's data loaded from the FlexSpace. Interval cache is a read cache that adopts a write-through policy and uses the CLOCK replacement algorithm [11]. A data write first persists in the underlying data file, then updates its corresponding cache entry. On a cache eviction, the entry to be evicted can be directly removed and freed without writing back its cached KV pairs.

When an interval is being loaded into the cache, FlexDB marks it as *fragmented* if the number of extents is more than half the number of KV pairs. When a marked interval is updated, FlexDB uses `flexspace_defrag` (§4.1) to perform defragmentation on it. In a cached interval, each KV pair is associated with a 16-bit hash fingerprint of the key for fast point queries with a minimal number of key comparisons. In range queries, a `SEEK` performs a binary search on the array.

5.3 Supporting Concurrent Access

Updates in FlexDB are buffered in a MemTable. The MemTable is a thread-safe skip list that supports concurrent access of one writer and multiple readers. Updates in the MemTable are periodically (or immediately when the MemTable is full) committed to the FlexSpace and the sparse index by a background committer thread. During this process, the MemTable becomes immutable and a new MemTable is created to receive updates. The committer can rewrite a highly fragmented interval for defragmentation if the thread is not fully loaded. A lookup in FlexDB first searches the MemTables. If the key is not found, it queries the sparse KV index to find the key in the FlexSpace.

When the committer thread is active, it requires exclusive access to the sparse index and the FlexSpace to prevent inconsistent data or metadata from being reached by readers. To this end, a reader-writer lock is used. A thread performing point or range queries must acquire the reader lock to access the sparse index or the FlexSpace. The committer thread must hold the writer lock while committing updates to the sparse index and the FlexSpace to block other threads that need to read them. Note that the lock does not block access to the MemTable. Particularly, a thread writing to the MemTable can only be blocked if the MemTable is full. For balanced performance and responsiveness, the committer thread releases and reacquires the lock every time it has committed 1000 KV pairs. Therefore, readers can be served quickly without waiting for the completion of the committing process. We will measure and discuss the wait time in §6.3.

5.4 Crash Recovery and Index Rebuilding

Upon a restart, FlexDB first recovers the uncommitted KV data from the write-ahead log. Then, it constructs the volatile sparse KV index. Intuitively the sparse index can be built by sequentially scanning the KV pairs in the FlexSpace, but the cost can be significant in a large store. In fact, the rebuilding only requires an index key for each interval. Therefore, a sparse index could be quickly constructed by skipping a certain amount of data every time an index key is determined.

In FlexDB, the FlexSpace's extents are created by inserting or removing KV pairs, which guarantees that every extent always begins with a KV pair. To identify a KV pair in the middle of the FlexSpace without knowing its exact offset, we add a `read_extent(off, buf, maxlen)` function to the FlexSpace library. The function searches for the extent at the designated offset (`off`) and reads up to `maxlen` bytes of data from the beginning of the extent. The extent's size, logical offset (≤ `off`), and the number of bytes read are returned. To build a sparse index, `read_extent` is used to retrieve a key at each approximate interval offset (8 KB, 16 KB, ...) and these keys are used as index keys of the new intervals. FlexDB can immediately start processing requests once the sparse index is built. A recovered interval whose size exceeds the split threshold will be split when it is accessed.

6 Evaluation

In this section, we experimentally evaluate FlexTree, the FlexSpace library, and FlexDB. All the experiments are run on a server with an Intel 10-core Xeon Silver 4210 CPU and 64 GB RAM. The persistent storage device of all tests is an Intel Optane 905P SSD with 960 GB capacity. The workstation runs a 64-bit Linux OS with kernel version 5.10.32 LTS.

Table 1. Throughput of the extent metadata operations

	Experiment	Insert		Append		Lookup		Range	
	# of Extents	10^5	10^6	10^8	10^9	10^8	10^9	10^8	10^9
Mops/sec	FlexTree	3.46	2.45	13.62	11.93	1.06	0.71	0.63	0.49
	B⁺-Tree	0.032	0.0018	14.07	12.13	1.11	0.70	0.63	0.49
	Sorted Array	0.029	0.0019	20.63	19.48	1.13	0.76	0.80	0.61

6.1 FlexTree as an Address Space Index

First of all, we evaluate the performance of the FlexTree index structure and compare it with a regular B⁺-Tree and a sorted array. In the evaluation of FlexTree, we want to answer the following questions: (1) What is the practical performance advantage of the asymptotic $O(\log N)$ shift operations in FlexTree compared to data structures that have $O(N)$ cost? (2) Can FlexTree efficiently handle range queries, which are frequently used for retrieving the address mapping information of a range of data? (3) How much overhead does FlexTree introduce to common address space operations such as lookup and append, compared to a regular B⁺-Tree?

The B⁺-Tree has the structure shown in Figure 2a, which is identical to FlexTree except that the shift values are removed from the internal nodes. In a shift operation, the B⁺-Tree and the array must update all the shifted extents.

We benchmark four index operations—*insert*, *append*, *lookup* and *range-query*. An insert experiment starts with an empty index. Each operation inserts a new extent at a random offset within the existing space. An append experiment starts with an empty index. Each operation appends a new extent after the existing extents. A lookup experiment randomly queries extents, and every operation must search the index. A range-query experiment randomly queries ranges consisting of 50 extents, where each operation searches for the first extent, then walks on the leaf nodes or the array to read the next 50 extents. These extent index structures are memory-resident and there are no persistent data.

Table 1 shows the throughput of each data structure in the experiments. Since FlexTree's address metadata representation scheme allows for much faster extent insertions, it shows high throughput in the insert experiments. However, the B⁺-Tree and the sorted array show extremely high overheads due to the intensive memory writes and movements. To be specific, every time an extent is inserted at the beginning, the entire mapping index is rewritten. FlexTree maintains a consistent $O(\log N)$ cost for insertions, which is asymptotically and practically faster.

For appends, the sorted array outperforms FlexTree and the B⁺-Tree because appending new extents at the end of an array does not need node splits or memory allocations. Meanwhile, FlexTree is only 3% slower than the B⁺-Tree. In the lookup and range-query experiments, the sorted array also outperforms the others because a binary search in a large sorted array with fixed-sized extents can be done efficiently without moving between a multi-level tree structure. In the three experiments, the throughput of FlexTree and B⁺-Tree are close, which suggests that the calculation of effective offsets in FlexTree is of low cost. FlexTree also inherits the good range query efficiency from B⁺-Tree.

6.2 The FlexSpace Library

In this section, we evaluate the efficiency of data I/O operations in the FlexSpace library. Note that FlexSpace is a storage engine that provides a persistent flexible address space for data management applications. Although there are overlaps between FlexSpace and file system functionalities, FlexSpace does not replace file systems on managing traditional files and directories. Therefore, file system benchmarks that require hierarchy directory structures do not apply to FlexSpace. In this section, we focus on data I/O and shifting operations within a persistent address space. Since FlexSpace adopts FlexTree as its extent index structure, we expect it to be highly efficient in shifting operations. In the meantime, FlexSpace is not specifically optimized for regular file I/O operations. Therefore, we do not expect FlexSpace to outperform traditional file systems on these operations.

We compare FlexSpace with file address spaces provided by four representative file systems, Ext4 [20], XFS [58], F2FS [34], and Btrfs [51]. Among them, Ext4, XFS, and F2FS support block-aligned shift operations. The four file systems are formatted using `mkfs` with their default arguments. FlexSpace stores its internal files on an XFS file system.

In the evaluation of FlexSpace, we want to answer the following questions: (1) What is the performance benefit of FlexSpace's *insert-range* and *collpase-range* operations? (2) How do different access patterns affect the performance of FlexSpace? (3) What are the performance implications of implementing a storage engine in the user space?

Each experiment consists of a write phase and a read phase with one thread. There are three write patterns for the write phase—random insert (using *insert-range*), random write, and sequential write. The first two patterns are the same as the INSERT and PWRITE in Section 2, respectively. The sequential write pattern writes data blocks sequentially. A write phase starts with an empty address space and writes or inserts data blocks using the respective pattern. Finally, an I/O barrier (`fsync` in file systems) is issued to enforce I/Os. Note that an I/O barrier in FlexSpace consists of flushing all buffered segment writes and synchronizing its internal files appropriately. After the write phase, we measure the read performance with two patterns—sequential and random. Each read operation reads a block of data from the address space. The random pattern uses randomly shuffled offsets so that it reads each data block in the address space exactly once. For each read pattern, the kernel page cache is first cleared. Then the program reads the entire address space twice, once with a cold cache and once with the cache

Table 2. Single-threaded I/O performance of FlexSpace and regular files in XFS, Ext4, F2FS, and BtrFS

I/O Size	4 KB (File Size = 1 GB)													64 KB (File Size = 16 GB)												
Write Pattern	Rand. Insert			Rand. Write					Seq. Write					Rand. Insert			Rand. Write					Seq. Write				
System	Flex	XFS	Ext	Flex	XFS	Ext	F2	Btr	Flex	XFS	Ext	F2	Btr	Flex	XFS	Ext	Flex	XFS	Ext	F2	Btr	Flex	XFS	Ext	F2	Btr
Write (GB/s)	0.62	ε	ε	0.61	0.57	0.50	0.61	0.62	0.62	0.63	0.60	0.55	0.62	0.75	ε	0.05	0.76	0.79	0.77	0.85	0.64	0.77	0.82	0.77	0.72	0.82
W. A. Ratio	1.03	5.53	2.23	1.03	1.02	1.06	1.02	1.10	1.03	1.02	1.05	1.02	1.03	1.02	1.83	1.10	1.02	1.02	1.05	1.03	1.03	1.02	1.02	1.05	1.03	1.03
Read (GB/s) Seq. Cold	0.39	1.11	0.95	0.41	1.97	1.82	1.81	1.15	1.92	1.97	1.83	1.81	1.66	0.95	1.93	2.07	0.95	1.92	2.05	2.02	1.63	1.93	2.05	2.05	2.02	1.71
Read (GB/s) Rand. Cold	0.38	0.38	0.36	0.38	0.41	0.37	0.40	0.24	0.40	0.41	0.36	0.40	0.25	0.94	0.97	0.99	0.94	0.95	1.00	0.98	0.79	0.95	1.01	0.98	1.00	0.81
Read (GB/s) Seq. Warm	3.44	4.26	4.48	3.44	4.39	4.44	4.43	4.44	4.33	4.42	4.44	4.42	4.38	5.45	5.76	5.70	5.43	5.70	5.71	5.73	5.74	5.58	5.71	5.74	5.76	5.74
Read (GB/s) Rand. Warm	2.42	3.23	2.75	2.40	3.28	3.39	3.41	3.38	2.77	3.28	3.36	3.42	3.38	5.21	5.46	4.43	5.17	5.43	5.44	5.40	5.49	5.19	5.42	5.47	5.42	5.46

warmed up. In the experiments, we adopt two I/O sizes—4 KB and 64 KB. With each I/O size, we use the same number of blocks (2^{18}) to construct the address space. Therefore, the address space sizes are 1 GB and 16 GB, respectively. Table 2 shows the experimental results (ε represents a value <0.01). We also include the write amplification (WA) ratios of each experiment, derived from the SMART data of the SSD. The following discusses the key observations from the experiment results.

Insert FlexSpace's random insert throughput can be up to 180× higher than Ext4 (620 MB/s vs. 3.36 MB/s) and four orders of magnitude higher than XFS. F2FS exhibits lower throughput than XFS so its results are omitted. Throughout the insertion process, FlexSpace can maintain high throughput while Ext4 and XFS suffer extreme throughput degradations because of the growing extent index sizes that lead to increasingly intensive metadata updates.

Write The random and sequential write throughput of FlexSpace is on par with the other systems. FlexSpace commits writes to the data file (stored in XFS) in the unit of segments, which enables batching and buffering in the user space. Meanwhile, FlexSpace adopts the log-structured write in the data file, which transforms random writes on the FlexSpace into sequential writes on the SSD. As a result, random writes in FlexSpace can outperform XFS with the 4 KB I/O size.

Write Amplification In the random and sequential write experiments, all the systems show low WA ratios because the metadata updates are not intensive. However, in the random insert experiments, Ext4 and XFS show very high WA ratios (up to 5.53) because each insert operation updates half of the existing extents' metadata on average, which leads to intensive computation and metadata I/O. XFS and Ext4's WA ratios are lower with the I/O size increased (64 KB) since the amount of metadata updates remains the same. That said, they still show low throughput because of the high computation cost. In FlexSpace, the insert operations have a very low cost ($O(\log N)$ time) and the logical logging can further reduce metadata write. As a result, FlexSpace achieves fast inserts (≥ 620 MB/s) with constantly low WA ratios (≤ 1.03).

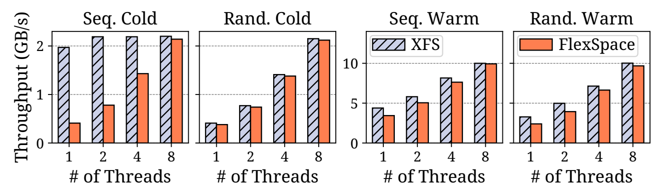


Figure 10. Read throughput after random write (4 KB)

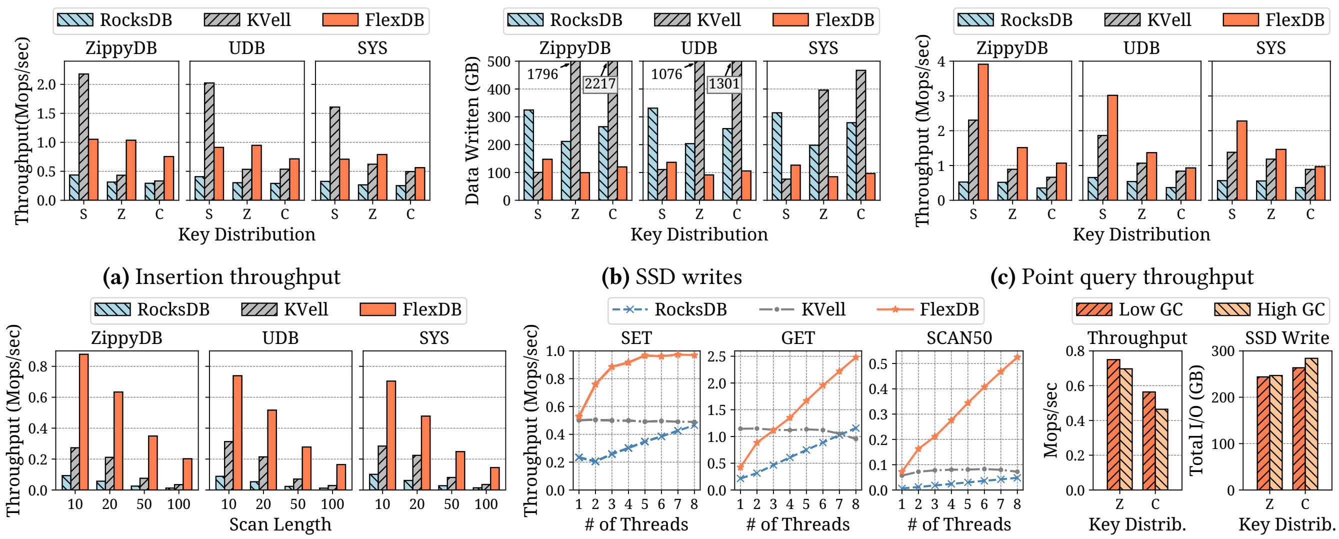
Read All the systems show similar read speed on address spaces constructed with sequential writes. However, with random writes/inserts, FlexSpace generates a fragmented data file layout which causes random read in the data file. As a result, when reading sequentially with a cold cache, FlexSpace shows 2.8× to 4.8× lower throughput than the file systems. That said, all the systems show slow random read with a cold cache since there is hardly any readahead in the kernel.

Data management systems often rely on asynchronous I/O or multi-threading to exploit I/O bandwidth [32, 33, 36]. To evaluate the I/O efficiency in this context, we run the read experiments with different numbers of threads. As shown on the left of Figure 10, XFS's throughput is already near its peak with one thread because of the automated readahead in the kernel. FlexSpace's throughput continues to increase with more threads and eventually reaches 98% of XFS's throughput.

As shown in Figure 10, FlexSpace's throughput is close to XFS when the cache is warmed up. The difference is larger with fewer threads because of the constant costs of accessing the FlexTree. Like the previous experiment, multi-threading can hide these costs and also increase access throughput. With eight threads, FlexSpace's throughput increased by up to 4× and is at least 96% of XFS's throughput.

6.3 FlexDB Performance

The goal of FlexDB is to demonstrate that a simple persistent KV store built based on a persistent flexible address space (FlexSpace) can match or outperform the state-of-the-arts that are built based on traditional files. We expect FlexDB to show significantly lower write amplification ratios because



(a) Insertion throughput **(b)** SSD writes **(c)** Point query throughput

(d) Scan throughput (Zipfian) **(e)** Scalability (UDB+Zipfian) **(f)** GC overhead

Figure 11. Microbenchmark results of FlexDB. Key distributions: *S* – Sequential; *Z* – Zipfian; *C* – Zipfian-Composite.

FlexSpace enables lightweight in-place update functionalities. Meanwhile, a fully sorted data layout in FlexDB shall greatly reduce its query cost.

We evaluate the performance of FlexDB through various experiments and compare it with Facebook's RocksDB [21], a representative LSM-Tree KV store, and KVell [36], an NVMe-optimized B$^+$-Tree-based KV store that exploits I/O bandwidth with asynchronous I/O and uses a full index in memory for fast search. We also evaluated LMDB (B$^+$-Tree based) [57] and TokuDB (B$^\epsilon$-Tree based) [45]. However, they exhibit consistently low performance compared with RocksDB. Similar observations are also reported in recent studies [17, 24, 44]. Therefore, their results are omitted.

For a fair comparison, both FlexDB and RocksDB are configured with 1 GB MemTables and 16 GB user-space cache. RocksDB is tuned as suggested by its official tuning guide [49][1]. FlexDB has its automatic defragmentation and the FlexSpace GC always enabled. KVell maintains its own page cache in the user space and uses direct I/O to bypass the kernel's cache. We adjust its cache size (≥16 GB) based on the actual memory footprint in each experiment to make sure it can fully utilize the available memory on the machine. Compression is disabled in all the stores.

All the experiments in this section run with 4 concurrent client threads unless otherwise noted. FlexDB uses only one background thread (the committer thread described in §5.3). RocksDB has up to 4 background compaction threads. KVell is configured with 4 worker threads, each with an I/O depth of 64. Therefore, the numbers of CPU cores that can be utilized by FlexDB, RocksDB, and KVell are 5, 8, and 8, respectively. For read and YCSB experiments, each data point is measured by running the respective workload for 60 seconds.

Table 3. Synthetic KV datasets with real-world KV sizes

Dataset	ZippyDB [5]	UDB [5]	SYS [1]
Avg. Key+Value Sizes (B)	48+43	27+127	28+396
Number of KV pairs	720 M	420 M	150 M
FlexDB Index Size	3.51 GB	1.50 GB	534 MB

We generate synthetic KV datasets using the representative KV sizes of Facebook's production workloads [1, 5]. Table 3 shows the details of the datasets, as well as the size of FlexDB's sparse index when storing each dataset. Note that the sparse index memory footprint can be tuned by adjusting the interval splitting threshold. The size of each dataset is about 64 GB, approximately 4× the size of the user-level cache in FlexDB and RocksDB. The workloads are generated using three key distributions—sequential, Zipfian ($\alpha = 0.99$), and Zipfian-Composite [24]. With Zipfian-Composite, the prefix (the first three decimal digits) of a key follows the default Zipfian distribution, and the remaining bits are drawn uniformly at random.

Write (PUT) Each write experiment starts from an empty store. Each client thread inserts 25% (approximately 16 GB) of the dataset to the store following the key distribution. For sequential load, the dataset is partitioned into four contiguous ranges, and each thread inserts one range of KV pairs. For the Zipfian and Zipfian-Composite distributions, existing keys can be overwritten, which leads to reduced write I/O if MemTables are used.

Figures 11a and 11b show the measured throughput and amount of disk I/O of the systems. KVell outperforms FlexDB

[1] Following the configurations for "Total ordered database, flash storage".

and RocksDB by more than 2× with sequential load, which is because KVell fully utilizes the I/O bandwidth without writing to a WAL. In comparison, FlexDB has only one committer thread and needs to record KV pairs in the WAL. Meanwhile, RocksDB must pay extra costs for compactions.

However, when facing workloads that regularly update existing keys (with Zipfian and Zipfian-Composite distributions), KVell shows significantly degraded throughput and up to 8.5× more data written to the SSD compared with FlexDB. The reason is that KVell uses slab allocators to manage space in the SSD and must perform block-sized in-place updates, which leads to high WA when the average KV size is smaller than the block size. FlexDB shows higher throughput than RocksDB by 2.2–3.3× across the experiments, The advantage mainly comes from FlexDB's capability of directly committing updates to the FlexSpace at low cost. In contrast, RocksDB requires repeated compactions to sort-merge KV pairs across the multi-level structure, which leads to high WA and computation cost. As shown in Figure 11b, RocksDB writes 2.1–2.9× more data to the SSD than FlexDB.

Read (GET and SCAN) We measure the point and range query throughput of the three systems. For each dataset, we populate the store with 4 threads, followed by 4 GB of random updates using the Zipfian distribution to emulate a randomized data layout in real-world KV stores.

As shown in Figure 11c, RocksDB shows low `GET` throughput because each operation requires a number of key comparisons to identify candidate tables at each level. For each candidate table, it needs to examine the bloom filter and then search the index if the filter returns true. KVell and FlexDB achieve higher throughput by maintaining a single-level in-memory index for fast lookups. The advantage of FlexDB is particularly high because it uses a much smaller sparse index and can quickly search in an interval with few key comparisons (see §5.2). Additionally, KVell stores the block address of each KV pair in the full index. A lookup in KVell needs to retrieve the cached block with an extra lookup in the page cache, which adds a constant overhead.

As shown in Figure 11d, the advantage of FlexDB remains significant in range queries because of its low cost on accessing KV data in the interval cache. In comparison, range queries in RocksDB require expensive sort-merging of KV data from multiple overlapping tables. To avoid synchronization overhead, KVell partitions the store with hash-based sharding, where each shard is exclusively managed by a worker thread. A range query in KVell must access every shard and sort-merge all the KV pairs at the client side to generate the search results. As a result, the scans are bottlenecked by excessive data copying and sort-merging.

Multi-core Scalability To measure the multi-core scalability of FlexDB, we rerun the write and read experiments with 1 to 8 client threads using the UDB dataset and the Zipfian access pattern. The scan experiments use a scan length of 50 keys. The results are shown in Figure 11e. FlexDB and RocksDB both scale well in the read (and also write for RocksDB) experiments because the workloads are mainly CPU-bound. However, FlexDB's write throughput stops increasing with more than 5 threads. In this scenario, the committer thread in FlexDB has been fully loaded and becomes the bottleneck. KVell shows constant throughput because it has a fixed number of worker threads, each exclusively processing requests for a shard. We reconfigure KVell with different numbers of shards, and the `GET` performance reaches its peak at 1.96 Mops/sec with 8 worker threads and 2 client threads (on the 10-core machine). The `PUT` and `SCAN` throughput do not improve since the I/O bandwidth is already saturated with four workers.

Table 4. Latency and Throughput with UDB+Zipfian

Op.	PUT				GET			
Sys.	Rocks	KVell	KVell_1	Flex	Rocks	KVell	KVell_1	Flex
Avg. (μs)	13.8	1669	153	3.9	9.0	453	72.6	3.7
95 p (μs)	17	2904	271	9	21	953	143	9
99 p (μs)	19	3386	306	17	43	1360	173	33
Mops/sec	0.30	0.53	0.09	0.95	0.52	1.13	0.15	1.65

Latency We discuss the latency metrics with the UDB dataset under Zipfian workloads (shown in Table 4). Compared with RocksDB, FlexDB is able to quickly commit KV updates to the FlexSpace instead of merging data in a multi-level structure. In the meantime, a lookup in FlexDB does not need to access multiple tables and sort data on the fly. Therefore, FlexDB shows the lowest latency metrics in both `PUT` and `SET` operations. KVell relies on asynchronous I/O to gain high throughput with a deep request queue (up to 64 queued requests). The queuing causes much longer response times than in FlexDB and RocksDB. That said, a smaller queue depth can improve responsiveness and reduce the latency readings of KVell. Accordingly, we measure KVell's latency metrics with its queue depth set to 1 and show the results in the columns named "KVell_1" in Table 4. KVell's latency metrics improve by about an order of magnitude by reducing the queue depth from 64 to 1, but the absolute numbers are still worse than FlexDB and RocksDB. Furthermore, the improvement comes at a cost of mediocre throughput because of the lack of I/O parallelism, as shown in the last row in Table 4.

GC Overhead We evaluate the impact of the FlexSpace GC activities on FlexDB using an update-intensive experiment. Each run of the experiment performs in total 800 million KV updates to a store containing the UDB dataset. The total update size is approximately twice the store size, which generates a fully aged storage layout during the experiment. We first run the experiment with the FlexSpace's data file size capped at 128 GB, which represents the scenario of a modest space utilization ratio (50%) and low GC overhead.

For comparison, we run the same experiments with the data file size capped at 75 GB. The smaller size leads to high GC activities in the FlexSpace with a higher space utilization ratio (85%). The results are shown in Figure 11f. The intensive GC shows a negligible impact on both throughput and I/O with Zipfian workloads. In this scenario, the GC process can easily find near-empty segments because the frequently updated keys are often co-located in the data file. Comparatively, the Zipfian-Composite distribution has a much weaker spatial locality, which leads to more rewrites in the GC process.

YCSB Benchmark YCSB [10] is a popular benchmark that evaluates KV store performance using realistic workload patterns. We use the UDB store populated by the corresponding four-thread load experiment, and run the YCSB workloads from A to F. The details of the YCSB workloads are shown in Table 5. A scan in workload E performs a seek and retrieves 50 KV pairs. Figure 12a shows the benchmark results.

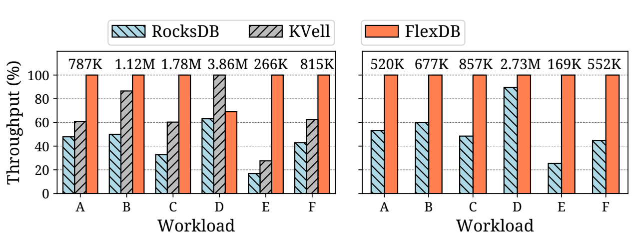
Table 5. YCSB workloads

Workload	A	B	C	D	E	F
Distribution	Zipfian			Latest	Zipfian	
Operations	50% U 50% R	5% U 95% R	100% R	5% I 95% R	5% I 95% S	50% R 50% M

* I: Insert; U: Update; R: Read; S: Scan; M: Read-Modify-Write.

In read-dominated workloads including B, C, and E, FlexDB outperforms RocksDB and KVell by 2.0–5.9× and 1.2–3.6×, respectively. This is especially the case in workload E because of FlexDB's advantage in range queries. Workload D performs sequential write while reading very recent updates with an ideal access locality. KVell achieves the highest throughput because it can evenly distribute requests across the hash-based shards without lock contention.

In write-dominated workloads, including A and F, FlexDB outperforms RocksDB and KVell by 2.1–2.3× and 1.6×, respectively. The performance advantage is not as high as that in the read-dominated workloads. In the FlexDB implementation, when the committer thread is merging updates into the FlexSpace, readers that reach the sparse index can be temporarily blocked (see §5.3). In workload A, the P99 latency is 30 μs with a maximum reader blocking time of 3.4 ms. The blocking time can be improved by partitioning the store [24], which is beyond the scope of this paper.



(a) 64 GB store **(b)** 500 GB store

Figure 12. YCSB benchmark with the UDB KV data size. Results of each workload are normalized to the highest. The throughput (ops/sec) of the highest bar in each experiment is shown on the top.

We also run the YCSB benchmark in an out-of-core scenario by increasing the UDB dataset size to about 500 GB. In this setup, KVell's full index does not fit in the available memory. When running with swap space enabled, KVell shows severe performance degradation by more than an order of magnitude compared to the in-core experiments, except for workload D that has optimal locality. Similar slowdowns are also observed in KVell's evaluation on the impact of different memory sizes [36]. Therefore, we do not turn on swap and exclude KVell from this experiment.

Figure 12b shows the out-of-core benchmark results. In this scenario, both FlexDB and RocksDB show reduced throughput in all the YCSB workloads due to the increased I/O cost. The advantage of FlexDB over RocksDB is reduced in the most I/O-intensive workloads (C and E). This is because the increased I/O time overshadowed FlexDB's search efficiency on the sparse index. That said, FlexDB still achieves 1.1–3.9× speedups over RocksDB.

Memory Size We analyze the impact of having a reduced memory size on the stores by running YCSB F benchmark with different memory configurations. In this experiment, we set the size ratio of total memory to user-space cache to 4:1, which is consistent with the default setup of all the previous experiments. Then, we adjust the available memory sizes from 64 GB to as low as 4 GB and measure the throughput of the systems. Since KVell's full index cannot be accommodated with the smaller memory sizes, we only compare FlexDB with RocksDB in this experiment.

Table 6. YCSB F throughput with different memory budgets

Size (GB)		Throughput (Mops/sec)		FlexDB
Memory	Cache	RocksDB	FlexDB	Speedup
64	16	0.350 (100%)	0.815 (100%)	2.3×
32	8	0.331 (94.6%)	0.709 (87.0%)	2.1×
16	4	0.315 (90.0%)	0.562 (69.0%)	1.8×
8	2	0.282 (80.6%)	0.443 (54.4%)	1.6×
4	1	0.133 (38.0%)	0.421 (51.7%)	3.2×

The experiment results are shown in Table 6. FlexDB outperforms RocksDB with different memory configurations. As the memory size decreases, the speed-up of FlexDB over RocksDB gets smaller. The reason is that FlexDB has an in-memory sparse index, which makes the size of the available OS page cache for FlexDB lower than that in RocksDB. RocksDB loads its block indexes on demand, which makes it more memory-efficient. That said, it still suffers from extra costs on searching its multi-level store. Additionally, when the memory size further reduces to 4 GB, RocksDB shows severe performance degradation because the available memory has become too small to make the block indexes sufficiently cached.

Recovery We evaluate FlexDB's recovery speed (described in §5.4) with a clean page cache and four concurrent recovery threads. For a store containing the 64 GB UDB dataset, the recovery process takes 7.8 seconds using a small rebuilding interval size of 16 KB. Increasing the recovery interval size to 64 KB reduces the recovery time to only 1.9 seconds. In practice, users can make trade-offs between reduced service downtime and better first-time access latency by adjusting the recovery interval size. Besides, the first-time access latency can be further reduced by promptly warming up the intervals in the background using spare bandwidth. RocksDB also achieves fast recovery by only scanning the WAL and lazily loading table files on demand. In comparison, KVell uses 64 seconds to rebuild a full index in the memory with four worker threads, and a complete scan of all the keys is inevitable in this process because of the unordered persistent storage layout of KVell.

7 Related Work

Data-management Systems Studies on improving I/O efficiency in data-management systems are abundant [14, 68]. B-tree-based KV stores [41, 43, 57] support efficient searching with minimum read I/O but have suboptimal performance under random writes because of the in-place updates [37]. LSM-Tree [42] uses out-of-place writes and delayed sorting to improve write performance, and it has been widely adopted in write-optimized KV stores [21, 23]. However, the improved write efficiency comes at a cost of slow read operations since a search may query multiple tables at different locations [39]. To compensate reads, LSM-tree based KV stores need to rewrite table files periodically using a compaction process, which in turn offsets the benefit of out-of-place write [4, 15, 16, 25, 27, 46, 48, 61, 69]. KVell and HiKV index all the keys in a volatile ordered index for fast access and leaves KV data unsorted on the persistent storage [6, 36, 62]. However, maintaining a volatile full index leads to high memory footprints and lengthy recovery processes. SplinterDB employs B^{ε}-tree for fast write by logging unsorted KV pairs in tree nodes [9]. However, the unordered node layout leads to slow reads, especially for range queries. Hashing-based KV stores gain point query efficiency but have to give up support to range queries [33, 61]. Recent studies also employ byte-addressable NVM for fast access and persistence [3, 7, 31, 32, 65]. These solutions require non-trivial implementations, including space allocation, GC, and maintaining crash consistency, which overlaps the core duties of file systems. FlexDB delegates the challenging data organizing tasks to the mechanisms behind the persistent address space, which effectively reduces application complexity. Managing persistently sorted KV data with efficient in-place updates achieves fast read and write at low cost.

Address Space Management Modern in-kernel file systems, such as Ext4, XFS, Btrfs, and F2FS, use B^{+}-Tree and its variants or multi-level mapping tables to index file extents [19, 34, 51, 58]. These file systems provide comprehensive support for general file management tasks but exhibit suboptimal performance in metadata-intensive workloads, such as massive file creation, crowded small writes, as well as *insert-range* and *collapse-range* that require data shifting. Recent studies employ write-optimized data structures in file systems to improve metadata management performance. Specifically, BetrFS [30, 66, 67], TokuFS [18], WAFL [40], TableFS [47], and KVFS [56] use write-optimized indexes, including B^{ε}-Tree [2] and LSM-Tree [42], to manage file system metadata. Their designs exploit the advantages of these indexes and successfully improved many existing file system metadata and file I/O operations. However, these systems still employ the traditional file abstraction and do not support easily moving data in the file address space. Therefore, managing sorted data in these systems still relies on rewriting existing data or employing indirections.

In-memory systems such as rewired memory [35, 54] utilize virtual memory mappings (i.e., page tables) to dynamically relocate page-aligned in-memory data blocks to sort data without copying. These mechanisms suffer from the same data shifting problems as in file extent indexes. Counted B-Tree [13] proposes storing the size of each subtree in internal nodes, which can be adapted for encoding address mappings with reduced shifting cost than B-Tree. However, querying address mappings on a Counted B-Tree requires linear scanning in each node on the search path, which is much more expensive than on a FlexTree. The design of FlexSpace removes a fundamental limitation in persistent address spaces. By leveraging the efficient shift operations for logically reorganizing data, applications built on FlexSpace can easily avoid data rewriting in the first place.

8 Conclusion

This paper presents a novel storage abstraction that provides a *flexible address space*, which enables lightweight in-place updates. It allows applications to perform efficient data management on a linear data layout with a simplified implementation. FlexDB, a KV store built on FlexSpace with a simple structure, achieves speedups of up to 16× for read and 3.3× for write, compared with highly optimized KV stores.

Acknowledgments

We are grateful to our shepherd Dr. Liuba Shrira and the anonymous reviewers for their invaluable feedback. Xingbo Wu conducted this work when he was at the University of Illinois at Chicago. This work was supported in part by the UIC startup funding.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload Analysis of a Large-Scale Key-Value Store". In: *SIGMETRICS Perform. Eval. Rev.* 40.1 (2012), pp. 53–64.

[2] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. "And introduction to Be-trees and write-optimization". In: *Login; Magazine* 40.5 (2015).

[3] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. "Viper: An Efficient Hybrid PMem-DRAM Key-Value Store". In: *Proceedings of the VLDB Endowment* 14.9 (2021), pp. 1544–1556.

[4] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. "Accordion: Better Memory Organization for LSM Key-Value Stores". In: *Proc. VLDB Endow.* 11.12 (2018), pp. 1863–1875.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook". In: *18th USENIX Conference on File and Storage Technologies (FAST'20)*. 2020, pp. 209–223.

[6] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. "FASTER: A Concurrent Key-Value Store with In-Place Updates". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 275–290.

[7] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong 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)*. 2021, pp. 17–32.

[8] DWARF Debugging Information Format Committee. *DWARF debugging information format version 5*. 2017.

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

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 2010, pp. 143–154.

[11] Fernando J Corbato. *A paging experiment with the multics system*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[13] *Counted B-Trees*. https://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html.

[14] Ali Davoudian, Liu Chen, and Mengchi Liu. "A Survey on NoSQL Stores". In: *ACM Comput. Surv.* 51.2 (2018).

[15] Niv Dayan and Stratos Idreos. "Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging". In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. 2018, pp. 505–520.

[16] Niv Dayan and Stratos Idreos. "The Log-Structured Merge-Bush & the Wacky Continuum". In: *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. 2019, pp. 449–466.

[17] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, and Tony Savor. "Optimizing Space Amplification in RocksDB." In: *The Conference on Innovative Data Systems Research (CIDR'17)*. Vol. 3. 2017, p. 3.

[18] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. "The TokuFS Streaming File System". In: *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*. 2012, p. 14.

[19] *Ext4 Disk Layout*. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[20] *Ext4 Filesystem*. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[21] Facebook. *RocksDB*. https://rocksdb.org.

[22] *fallocate(2) — Linux manual page*. https://www.man7.org/linux/man-pages/man2/fallocate.2.html.

[23] Sanjay Ghemawat and Jeff Dean. *LevelDB*. https://github.com/google/leveldb.

[24] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. "EvenDB: Optimizing Key-Value Storage for Spatial Locality". In: *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. 2020.

[25] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. "X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing". In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 651–665.

[26] Stratos Idreos and Mark Callaghan. "Key-Value Storage Engines". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2667–2672.

[27] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. "PinK: High-speed In-storage Key-value Store with Bounded Tails". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 173–187.

[28] *Inserting a hole into a file*. https://lwn.net/Articles/629965/.

[29] *Intel® Optane™ Technology*. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[30] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. "BetrFS: Write-Optimization in a Kernel File System". In: *ACM Trans. Storage* 11.4 (2015).

[31] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. "SLM-DB: Single-Level Key-Value Store with Persistent Memory". In: *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 2019, pp. 191–204.

[32] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "Redesigning LSMs for Nonvolatile Memory with NoveLSM". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'18)*. 2018, pp. 993–1005.

[33] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. "Reaping the Performance of Fast NVM Storage with Udepot". In: *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 2019, pp. 1–15.

[34] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. "F2FS: A New File System for Flash Storage". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 2015, pp. 273–286.

[35] D. De Leo and P. Boncz. "Packed Memory Arrays - Rewired". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 830–841.

[36] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. "KVell: The Design and Implementation of a Fast Persistent Key-Value Store". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. 2019, pp. 447–461.

[37] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. "Tree Indexing on Solid State Drives". In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 1195–1206.

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

[39] Chen Luo and Michael J Carey. "LSM-based storage techniques: a survey". In: *The VLDB Journal* 29.1 (2020), pp. 393–418.

[40] Peter Macko, Margo Seltzer, and Keith A. Smith. "Tracking Back References in a Write-Anywhere File System". In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. 2010, p. 2.

[41] *MongoDB*. https://www.mongodb.com/.

[42] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Inf.* 33.4 (1996), pp. 351–385.

[43] Michael A Olson, Keith Bostic, and Margo I Seltzer. "Berkeley DB." In: *USENIX Annual Technical Conference, FREENIX Track*. 1999, pp. 183–191.

[44] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. "Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-Value Store". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'16)*. 2016, pp. 537–550.

[45] *PerconaFT (TokuDB)*. https://github.com/percona/PerconaFT.

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

[47] Kai Ren and Garth Gibson. "TABLEFS: Enhancing Metadata Efficiency in the Local File System". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. 2013, pp. 145–156.

[48] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. "SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data". In: *Proc. VLDB Endow.* 10.13 (2017), pp. 2037–2048.

[49] *RocksDB Tuning Guide*. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[50] Ohad Rodeh. "B-Trees, Shadowing, and Clones". In: *ACM Trans. Storage* 3.4 (2008).

[51] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-Tree Filesystem". In: *ACM Trans. Storage* 9.3 (2013).

[52] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System". In: *ACM Trans. Comput. Syst.* 10.1 (1992), pp. 26–52.

[53] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-Structured Memory for DRAM-Based Storage". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 2014, pp. 1–16.

[54] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. "RUMA Has It: Rewired User-Space Memory Access is Possible!" In: *Proc. VLDB Endow.* 9.10 (2016), pp. 768–779.

[55] Kai Shen, Stan Park, and Meng Zhu. "Journaling of Journal is (Almost) Free". In: *Proceedings of the 12th*

USENIX Conference on File and Storage Technologies (FAST'14). 2014, pp. 287–293.

[56] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. "Building Workload-Independent Storage with VT-Trees". In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13).* 2013, pp. 17–30.

[57] *Symas Lightning Memory-mapped Database.* https://symas.com/lmdb/.

[58] *The SGI XFS Filesystem.* https://www.kernel.org/doc/Documentation/filesystems/xfs.txt.

[59] Darrick Wong, Dave Chinner, Eric Sandeen, Ryan Lerch, and Sillion Graphics Inc. *XFS Algorithms & Data Structures, 3rd Edition.* 2018.

[60] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "Towards an Unwritten Contract of Intel Optane SSD". In: *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'19).* 2019, p. 3.

[61] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. "LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items". In: *2015 USENIX Annual Technical Conference (USENIX ATC'15).* 2015, pp. 71–82.

[62] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference.* 2017, pp. 349–362.

[63] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. "Don't Stack Your Log On My Log". In: *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14).* 2014.

[64] Juncheng Yang, Yao Yue, and K. V. Rashmi. "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).* 2020, pp. 191–208.

[65] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. "MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM". In: *2020 USENIX Annual Technical Conference (USENIX ATC'20).* 2020, pp. 17–31.

[66] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. "Optimizing Every Operation in a Write-Optimized File System". In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies.* 2016, pp. 1–14.

[67] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. "The Full Path to Full-Path Indexing". In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18).* 2018, pp. 123–138.

[68] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. "In-Memory Big Data Management and Processing: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 27.7 (2015), pp. 1920–1948.

[69] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. "REMIX: Efficient Range Query for LSM-trees". In: *19th USENIX Conference on File and Storage Technologies (FAST 21).* 2021, pp. 51–64.

A Artifact Appendix

A.1 Abstract

The artifact contains our implementation of FlexTree, FlexSpace, FlexDB, and the scripts we used to produce the reported evaluation results. It demonstrates a bottom-up design of a data management application based on the abstraction of a flexible address space. The goal of this artifact is to allow readers to reproduce the paper's key results, and build new research on top of our proposed work.

A.2 Description & Requirements

A.2.1 How to access The source code of FlexTree, FlexSpace, FlexDB and their future updates can be found at: https://github.com/flexible-address-space/flexspace. The materials used in this paper's artifact evaluation are archived at: https://github.com/flexible-address-space/eurosys22-artifact. The instruction for artifact evaluation is documented in a separate file, `README.md`, in the artifact repository. These materials are indexed by https://doi.org/10.5281/zenodo.6345713.

A.2.2 Hardware dependencies The system functionality does not require special hardware. To reproduce the experiment results, we suggest using similar hardware as we used in our evaluation (Intel 10-core Xeon Silver 4210 CPU, 64 GB RAM and Intel Optane 905P SSD with 960 GB capacity).

A.2.3 Software dependencies This artifact now only supports operating systems using the Linux kernel. To build our systems, it is required to use a Linux kernel with `io_uring` support (version 5.1 and up). The user space dependencies are `liburing`, `jemalloc` and `clang`.

A.2.4 Benchmarks The code for micro-benchmarks described in the paper has been included in the artifact materials. Specifically, the benchmark code for FlexDB contains an implementation of the YCSB [10] benchmark. The file systems we used to compare against FlexSpace are all in the Linux kernel source tree. The RocksDB we used in the evaluation is unmodified and can be fetched from its public source code. We used a patched version of KVell to support variable-sized keys. Its code is archived at: https://github.com/flexible-address-space/eurosys22-artifact-kvell.

A.3 Set-up

The artifact is verified to compile on Arch Linux with kernel version 5.10.32 LTS, and all user-level dependency packages (`clang` 12.0.1, `jemalloc` 5.2.1 and `liburing` 2.0).

A.4 Evaluation workflow

A.4.1 Major Claims First, FlexTree achieves significantly lower asymptotic (and practical) time complexity on shift operations compared to B$^+$-Tree, and it introduces a negligible extra cost on regular index operations. Second, FlexSpace realizes faster *insert-range* and *collapse-range* speed compared to address spaces provided by major file systems. Third, FlexDB shows high performance and low write amplification under various micro-benchmarks and real-world workloads with a simple design.

A.4.2 Experiments We provide fully automated scripts to run the experiments and interpret the results with minimal effort. You can use them to run the experiments following the documentation in https://github.com/flexible-address-space/eurosys22-artifact. We expect that the results are similar to those reported in the paper, or showing a similar trend (i.e., do not affect the major claims).

A.5 General Notes

This appendix only applies to the artifact submitted for evaluation. Future updates of the implementation will be made available through the main repository.