Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Modern Storage Stack with Key-Value Store Interface and Snapshots Based on Copy-On-Write B^{ε}-Trees

vorgelegt von

Felix Wiedemann

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	M. Sc. Informatik
Matrikelnummer:	6323654
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Dr. Michael Kuhn

Hamburg, den 11.01.2018

There are two hard problems in computer science: cache invalidation, naming things, and off-by-one errors.

– Leon Bambrick

Abstract

The ever increasing gap between computational power and storage capacity on the one side and storage throughput on the other side leads to I/O bottlenecks in many applications. To cope with huge data volumes, many storage stacks apply copy-on-write techniques. Copy-on-write enables efficient snapshots and guarantees on-disk consistency. However, a major downside of copy-on-write is potentially massive fragmentation as data is always redirected on write. As fragmentation leads to random reads, the I/O throughput of copy-on-write storage stacks suffers especially under random write workloads.

In this thesis, we will design, implement, and evaluate a copy-on-write storage stack that uses B^ε-Trees which are a generalisation of B-Trees and limit fragmentation by design due to larger node sizes. The storage stack has an underlying storage pool which consists of groups of storage devices called *vdevs*. Each vdev is either a single storage device, a mirror of devices, or a group of devices with additional parity blocks so that a storage pool improves performance and/or resilience compared to a single storage device. In the storage pool, the data is protected by checksums. On top of the storage pool, we use B^ε-Trees to save all user data and metadata. The user interface of the storage stack provides data sets which have a simple key-value store interface and save their data in dedicated B^ε-Trees. Each data set can be individually snapshotted as we simply use the path-copying technique for the corresponding B^ε-Tree.

In the performance evaluation, our storage stack shows its advantage over ZFS – a mature copy-on-write storage stack – in a database workload. Our storage stack is not only 10 times faster regarding small random overwrites (6.6 MiB/s versus 0.66 MiB/s) but it also exhibits a much smaller performance degradation in the following sequential read of data. While the sequential read throughput of ZFS drops by 82% due to the random writes, our storage stack only incurs a 23% slowdown. Hence, limiting fragmentation by design can be very useful for copy-on-write storage stacks so that the read performance is higher and more consistent regardless of write access patterns.

Contents

1. Introduction

Nowadays, the gap between computational power and storage capacity on the one hand and storage throughput on the other hand increases constantly (see Figure 1.1). I/O performance became the major issue in the recent decades as it is often the bottleneck. Sequentially reading a large HDD takes many hours in 2017, as opposed to multiple minutes 20 years ago. Regarding random access patterns, this ratio becomes worse as the seek time of an HDD dominates the access time, and the seek time has only seen a 2-fold improvement in the last 20 years [Con17e].

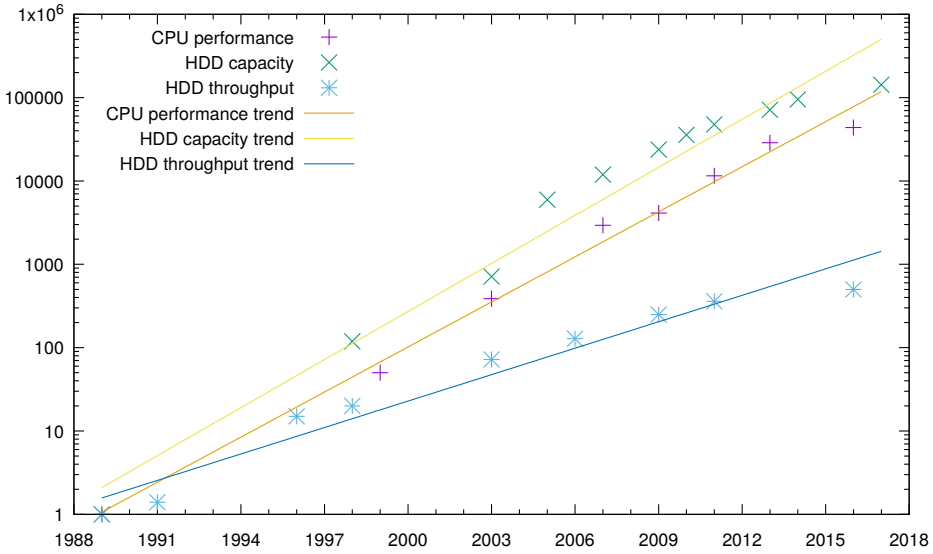

Figure 1.1.: Historical development of CPU performance, HDD capacity, and HDD throughput relative to 1989 [Sch06; Con17d; Con17c; Con17f].

Data reduction techniques such as compression are becoming more popular since less data results in better space utilization and this often yields higher I/O performance, too. In particular, compression provides a trade-off between computational power and the amount of I/O and, hence, can offset the mentioned gap. Nonetheless, we need new concepts for saving data efficiently as the gap will likely increase in the foreseeable future.

Copy-on-write is such a new concept. With copy-on-write, we never overwrite active data but redirect every write to a new location.[1] When always used, the on-disk state of this software is consistent at all times. Hence, copy-on-write is a solution for file

[1] This technique also known as *redirect-on-write* as copy-on-write actually means that the old data is copied before overwriting it. We will use the two terms synonymously.

systems checks which become infeasible due to the huge volume of data. In addition, copy-on-write enables us to implement lightweight snapshots that can be used for efficient incremental backups which also become infeasible otherwise. A downside of copy-on-write is that it causes fragmentation as each write is relocated. Hence, the use of copy-on-write leads to more random reads so that we have in turn lower read throughput, which we want to avoid because of the mentioned gap.

Therefore, in this thesis, we will combine the copy-on-write technique with a fragmentation limiting data structure – the B^ε-Tree. We will build a general purpose storage stack with support for multiple data sets and snapshots that provides data integrity and resilience against disk failures. Additionally, we analyze and evaluate the implementation against popular competitors used in production.

1.1. Outline

The thesis continues with the following chapters:

Background This chapter introduces technical and theoretical background to build data structures for persistent storage. This includes the primary data structure for this thesis, the B^ε-Tree, and a list of key features for storage stacks.

Design This chapter presents the layered design approach of the storage stack of this thesis and describes each layer in detail.

Comparison In this chapter, we compare the design against two popular storage stacks. First, the Linux storage stack based on ext4, LVM, and RAID and second, ZFS.

Related Work We compare related work with our approach regarding modernizing storage stacks and increasing the efficiency of storage stack layers.

Implementation In this chapter, we will focus on the Rust programming language in which the storage stack is implemented and highlight some implementation details and optimizations.

Evaluation Finally, we will evaluate the performance of this storage stack with different workloads and compare the results against the Linux storage stack and ZFS.

Conclusion This chapter will summarize the results of this thesis.

Future Work In this chapter, we present possible extensions for the storage stack and show possible directions for future work.

Usage Last but not least, this appendix is a quick start for using Rust and the storage stack implementation.

2. Background

In this chapter, we will introduce storage concepts and data structures. First, we will present the "Disk Access Model" for analyzing data structures and algorithms that save data on persistent storage. Afterwards, we introduce the B-Tree which is the most commonly used data structure for persisting data. Then, we will introduce a generalization of the B-Tree called B^ϵ-Tree which will be the primary data structure for this thesis. Finally, we define the term "storage stack" and list key features of modern storage stacks.

2.1. Disk Access Model

Modern computers tend to have a complex memory hierarchy with many levels which exhibit vastly different performance characteristics (see Table 2.1). For analyzing the theoretical performance of persistent data structures like B-Trees, we need a suitable but much simpler model of a computer.

The *Disk Access Model* (*DAM*) – also known as *external memory model* – introduced by Aggarwal and Vitter in 1988 [AV88] is the standard model for analyzing persistent data structures. It relies on the assumptions that the persistent memory levels are multiple orders of magnitude slower than the volatile ones and that the data structure is actually I/O bound. The modeled computer has a two level memory hierarchy: A CPU with an attached cache of size M and a disk of unbounded size. The data transfer between the cache and the disk operates with a block size B:

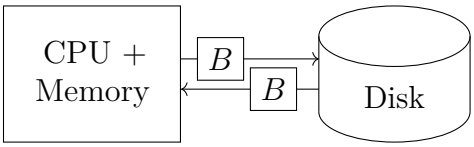

Figure 2.1.: *DAM* machine

The running time of an algorithm or data structure operation is defined as the number of block transfers between cache and disk. Hence, this model emphasizes reducing the number of disk accesses.

2.1.1. Shortcomings

Due to its simplicity, the *Disk Access Model* does not account for specific performance characteristics of HDDs and SSDs: Most importantly, the block transfer cost is uniform and does not depend on the access pattern.

Memory Level	Latency	Throughput	Size
L1 Cache	1 ns	900 GiB/s	4x 64 KiB
L2 Cache	4 ns	350 GiB/s	4x 256 KiB
L3 Cache	14 ns	190 GiB/s	8 MiB
RAM	65 ns	20 GiB/s	16 GiB
SSD (SATA)	60 µs	500 MiB/s	512 GiB
HDD	15 ms	200 MiB/s	4 TiB

Table 2.1.: Performance characteristics of various memory levels on a modern desktop computer with an Intel E3-1225 v3 CPU, a Samsung SSD 850 EVO 500 GB MZ-75E500B and a Seagate Desktop HDD 4 TB ST4000DM003 [Spe13; Ins13; Was13].

Another aspect is choosing the optimal block size for an algorithm in practice. Computers with different cache sizes and number of cache levels may require specific block sizes for the algorithm to perform optimally. Hence, the algorithm needs to be optimized for each computer individually. Also, each cache level might need a different block size to perform optimally.

In the following we will analyze and discuss these aspects.

Access Patterns

In practice, HDDs and SSDs benefit from spatial locality. The total time needed for a block transfer is the access time plus the transfer time. The access times of multiple consecutive requests depend on the actual access pattern and can vary a lot – especially on HDDs as an HDD may have to move its head to the correct position. Figure 2.2 shows the read and write throughput of a consumer HDD and SSD using sequential and random access patterns with block sizes from 4 KiB to 4 MiB.

At 4 KiB block size – which is the native sector size of the disk – the HDD has a very low random read and write throughput with about 500 KiB/s and 1 MiB/s. Sequential throughput at the same block size is about two orders of magnitude higher than random. As the block size increases the gap closes. With 4 MiB block size, the difference is only about 40%.

Regarding the read performance of the SSD, the situation is similar but less pronounced. At 4 KiB, sequential access is only about three times faster than random access. At 4 MiB block size, both random read and sequential read have a throughput of 510 MiB/s and nearly max out the SATA interface.

The write throughput of the SSD is the same for both access patterns at 4 KiB and 8 KiB. At larger block sizes, the sequential write performance steadily increases up to 450 MiB/s, but the random write performance of the SSD seems to be artificially capped at 200 MiB/s by the drive firmware – maybe to reduce the stress on the internal garbage collection and wear leveling algorithms or to distinguish this consumer device from more expensive SSDs.

As a conclusion, we see that the block size matters but also the access pattern plays

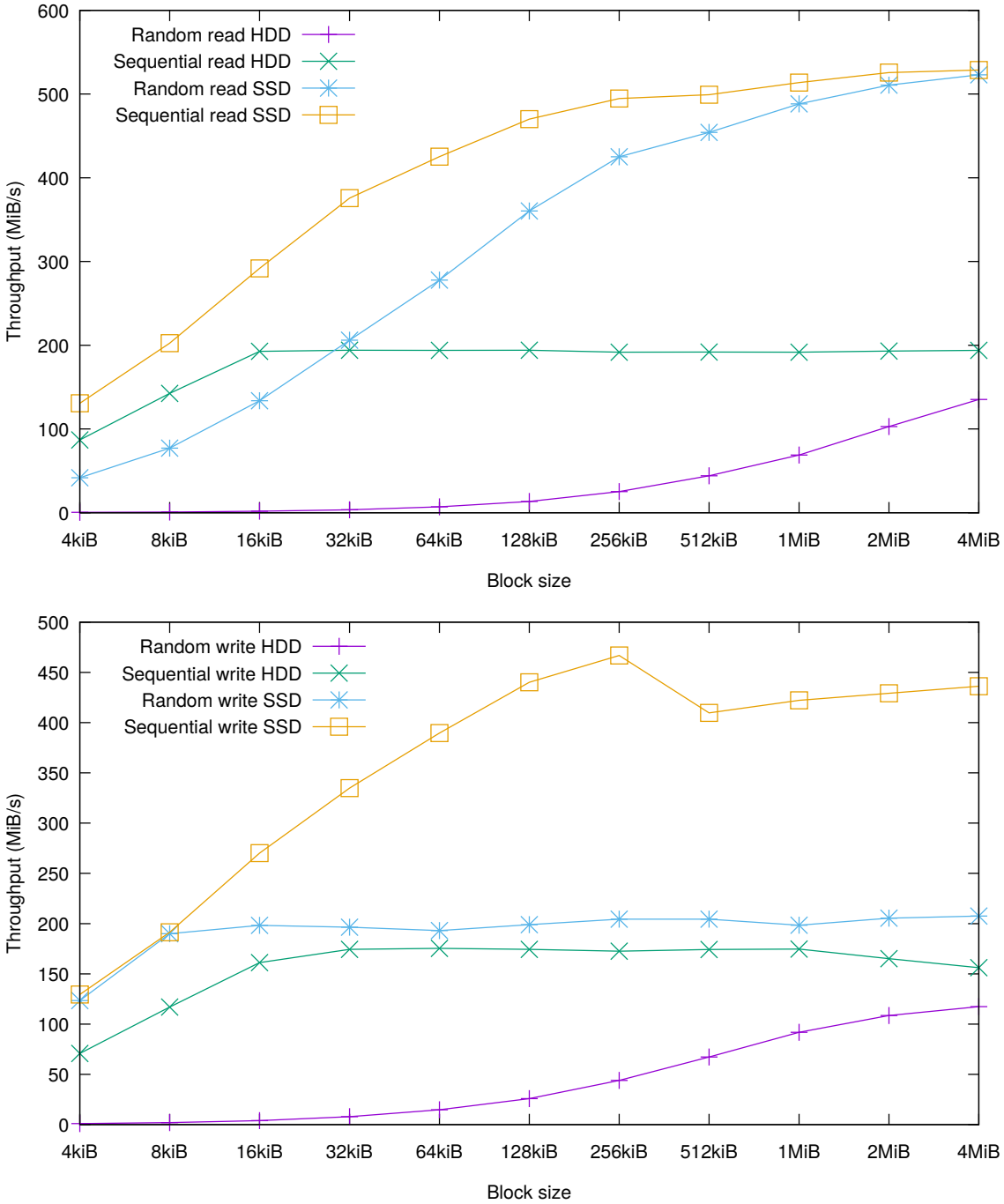

Figure 2.2.: Read and write throughput of a 3.5“ 1 TB 7200rpm HDD[a] and a SATA3 SSD[b] with varying block sizes.

[a]Model name: Seagate Desktop HDD 1 TB ST1000DM003.
[b]Model name: Samsung SSD 850 EVO 500 GB MZ-75E500B.

a major role – even for SSDs. The latter part is not taken into account by the *Disk Access Model*. Consequently, algorithms with the same runtime under the *DAM* but with different access patterns may perform vastly different in practice. To cope with this deficit, we will analyze the access patterns of the algorithms in this thesis in addition to the runtime performance in the model.

Optimal Block Sizes

Choosing an optimal block size is a hard problem as the performance of algorithms depends on many factors so that different computer systems require different block sizes. The upside is that most factors do not have a strong influence on performance. According to the Disk Access Model, only the last level of all levels in the memory hierarchy matters as it is the slowest by a large margin. This last level is actually quite similar among most computer systems as it consists either of HDDs or SSDs today.

Hence, an algorithm is often optimized for HDDs and SSDs separately, i.e. the algorithm chooses a different block size. Another aspect is choosing different block sizes for different workloads as the workload can be a major factor. Often, the actual user can manually change the block size to tune the algorithm for their workload.

A general solution to this problem is the more advanced *Cache-Oblivious Model* (*CO Model*) where – in contrast to the *DAM* – cache size and block size are not known to the algorithm [Fri+99]. Hence, an algorithm which is optimal in the *CO Model* performs good on every memory hierarchy and does not need any manual tuning.

2.2. B-Tree

The *B-Tree* is one of the most common index data structures for external memory today. It was introduced by Bayer and McCreight in 1972 [BM70]. The following definition of a *B-Tree* is based on [Rod07] and [Cor+09]. For a more in-depth introduction, see *Introduction to Algorithms, 3rd Edition* [Cor+09].

From a high-level perspective, a *B-Tree* is a *map* from a totally ordered set of keys to a set of values and provides the following operations which run in logarithmic time:

query(k) Returns the *value* of a *key* if present.

insert(k, v) Inserts a new *key value* pair.

delete(k) Deletes a *key*.

range-query(k_{low}, k_{high}) Returns a sequence of *key-value pairs* that lie in the given *key range*.

As the name suggests, a *B-Tree* is a tree and it has two node types: internal nodes and leaves. Every internal node contains an ordered sequence of keys (k_i) and a sequence of child pointers (c_i). There is exactly one more child pointer than keys. Every leaf node contains an ordered sequence of key-value pairs ((k_i, v_i)).

Let N be the number of key-value pairs in the *B-Tree* and let $t \geq 2$ be the minimum *degree* of the internal nodes. The following structural invariants hold:

1. Every non-root node contains at least $t - 1$ keys and, hence, at least t children for internal nodes. The internal root node contains at least 1 key and, thus, at least 2 children.

2. Every node contains at most $2t + 1$ keys (and internal nodes have at most $2t + 2$ children).

3. All leaf nodes have the same depth.

Figure 2.3 shows a conceptual overview. This variant of a *B-Tree* – no data in the internal nodes – is also known as a *B^+-Tree* and is the most common one in actual implementations. Having no data in the internal nodes has two main benefits: First, the range query algorithm simply needs to walk along the leaf nodes as the leaves contain all key-value pairs. Second, the internal nodes can have a higher fanout so that caching the internal nodes becomes more effective.

In the following we will describe how to implement the four mentioned operations, give brief argumentation on correctness, and analyze their resource requirements and runtime performance. We will use a *top-down* approach for fixing the structural invariants of the tree due to an insertion or deletion – similar to [Rod07].

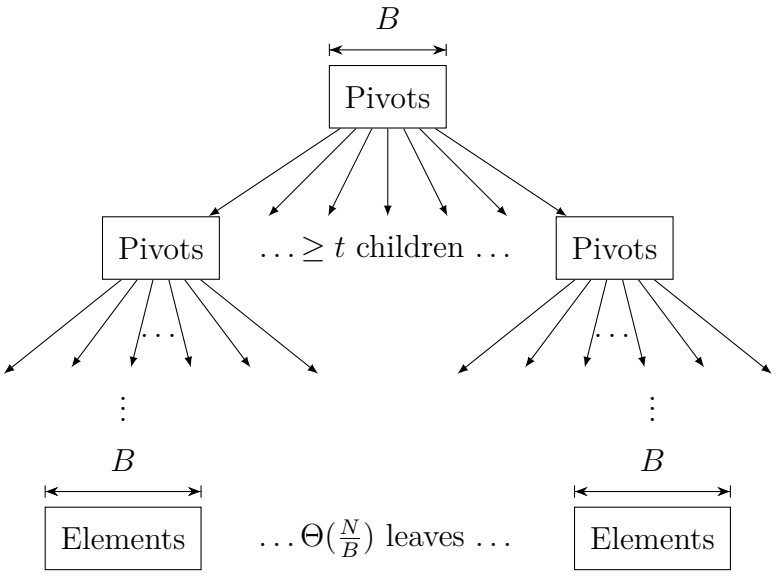

Figure 2.3.: B-Tree. Figure adapted from [Jan+15b]

2.2.1. Operations

Query

Let k be the queried key. Starting at the root, apply the following recursive algorithm:

1. If the current node is an internal node, check for the largest i so that $k \leq k_i$. If there exists a satisfying i, query c_i for the key k. Otherwise, query the right-most child.

2. If the current node is a leaf node, look for the corresponding key value pair and return it if it exists. Otherwise, return nothing.

Correctness Because this algorithm walks down the tree w.r.t. the search order, it finds the correct leaf and, hence, returns the correct result. As it does not modify the tree, it cannot violate the invariants.

Insert

Let (k, v) be the key-value pair to be inserted. Starting at the root, apply the following recursive algorithm:

1. If the root node has $2t + 1$ keys, execute the operation *Split* and add a new node as root node which is an internal node with this node and its new sibling as children and the split pivot key (see Figure 2.4).

2. If this node is a leaf node, insert the key-value pair in this node so that the order is maintained if the key does not exist yet. If the key exists, just replace its value.

3. Otherwise, it's an internal node. Find the largest i so that $k \leq k_i$. If i exists, use c_i as child node. Otherwise, use the right-most children as child node.

4. If this child node has $2t + 1$ keys, execute the *Split* operation on the child and insert its right sibling as c_{i+1}, shifting all children after it to the right. Insert the pivot key as k_{i+1}, shifting all pivot keys after it to the right. Go to step 3 again.

5. Go to step 2 with the child node as current node.

Correctness This algorithm walks down the tree w.r.t. the search order so that the inserted key pair is in the correct leaf. It ensures that each node on this path has fewer than $2t + 1$ keys before visiting it – except the root node. As the parent nodes are not full, their child nodes can be split if they are full and the key pair can be inserted into the leaf without overflowing. Note that step 4 is executed at most once per level.

Hence, the algorithm does not violate the invariants 1 and 2. Only the first step in the algorithm may violate invariant 3, but as it adds one tree layer above the previous node all leaf nodes are still at the same depth, so the third invariant is not violated.

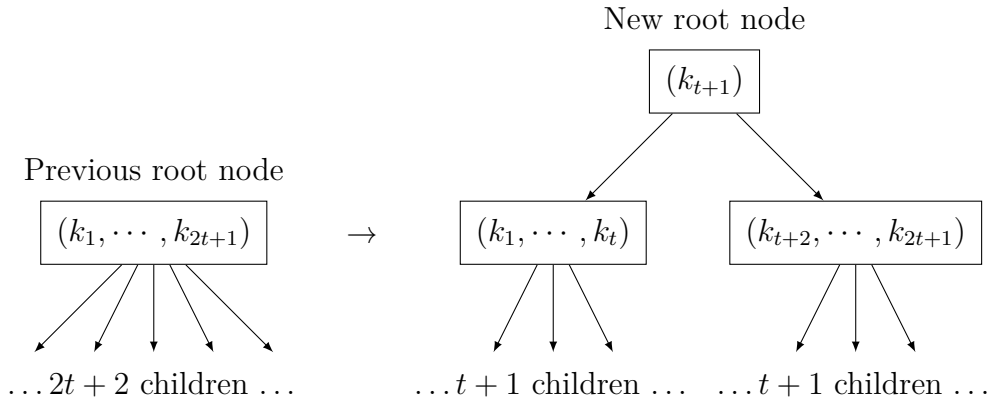

Figure 2.4.: Splitting the internal root node of a B-Tree

Split

Given: A node.
Returns: A new sibling node and a pivot key.
Precondition: Node has $2t + 1$ keys.
Postcondition: Both nodes have at most $2t$ keys.

1. If this node is an internal node, remove the keys $k_{t+1}, \cdots, k_{2t+1}$ and the children $c_{t+2}, \cdots, c_{2t+2}$ from the node. Add a right sibling internal node to the node with $(k_{t+2}, \cdots, k_{2t+1})$ as pivot keys and $(c_{t+2}, \cdots, c_{2t+2})$ as children. Use k_{t+1} as pivot key for these two nodes in the parent.

2. If this node is a leaf node, remove the key-value pairs $(k_{t+2}, v_{t+2}), \cdots, (k_{2t+2}, v_{2t+2})$ and make a new right sibling leaf node which contains these (in the original order). Use k_{t+1} as pivot key for these two nodes in the parent.

Correctness This algorithm obeys the search order because the new pivot key of the two nodes divides the key space in two parts, where the left nodes belongs to the lower part ($\leq$) and the right node to the upper part ($>$). Splitting a node results in two nodes at the same depth with at least t and at most $t + 1$ keys. Hence, it does not violate the invariants.

Delete

Let k be the key to be deleted. Starting at the root, apply the following recursive algorithm:

1. If this node is a leaf node, delete the corresponding key-value pair if present and return.

2. Otherwise, it is an internal node. Find the largest i so that $k \leq k_i$. If i exists, use c_i as child node. Otherwise, use the right-most children as child node.

3. If this child has $t-1$ keys, execute the *Rebalance* operation on child this node and one of its direct siblings. If this is the root node and it has only one child, set the child as root. Go to step 2 again.

4. Go back to Step 1 with the child node as current node.

Correctness This argumentation is very similar to the argumentation for the *Insert* operation. The algorithm walks down the tree w.r.t. the search order so that it finds the leaf to which the key belongs. It ensures that each node on this path has at least t keys before visiting it – except the root node. As the parent nodes have one more key than they must have, their child nodes can be merged by *rebalancing* and the key-value pair can be deleted without underflowing. Note that step 3 is executed at most once per level.

Hence, the algorithm does not violate the first two invariants. In the second step, we may change the depth of nodes by removing the root node, but all leaf nodes will still be at the same depth.

Rebalance

Given: Two nodes which are direct siblings and their pivot key in the parent.
Returns: Either a new pivot key for the parent or a merged node.
Precondition: One of the nodes has $t-1$ keys.
Postcondition: The two nodes (or the merged node) have at least t keys.

1. If they are internal nodes: Append the parent's pivot key and the pivot keys of the right node to the pivot keys of the left node. Append the right node's children to the left node's children.

2. If they are leaf nodes: Append the key-value pairs of the right node to the key-value pairs of the left node.

3. If the right node has had at most t keys, return the left node as the merged node.

4. If they are internal nodes: Let x be the number of keys in the left node. Move the upper $\lfloor \frac{x}{2} \rfloor$ keys and the upper $\lfloor \frac{x}{2} \rfloor + 1$ children from the left node to the right node. Remove the upper most key from the left node and return it as pivot key.

5. If they are leaf nodes: Move the upper half of the key-value pair sequence of the left node to the right node. Return the largest key in the left node as pivot key for the parent.

Correctness This algorithm obeys the search order because keys and children are not reordered during the operation but only moved between the two nodes. If the two nodes have been merged, they had at least $2t-2$ and at most $2t-1$ keys due to the condition in step 3. Hence, the number of keys in the merged node is in bounds. Otherwise, the nodes will be rebalanced and both nodes have in total at least $2t$ and at most $3t$ keys. By splitting the number of keys evenly between both nodes, they have at least t keys

and clearly fewer than $2t$ keys. So the number of keys is in bounds for the rebalance case, too.

Range Query

Let (k_{low}, k_{high}) be the queried key range. Starting at the root, apply the following recursive algorithm:

1. If the current node is a leaf node, return all key-value pairs that are in the queried range.
2. Otherwise, the current node is an internal node.
3. Find the largest i so that $k_i < k_{low}$ or set i to 0 if no such i exists.
4. Find the smallest j so that $k_j \geq k_{high}$ or set j to the number of pivot keys.
5. Call this algorithm for each child in $(c_i, \cdots, c_j)$ and aggregate the results.

Correctness This algorithm recursively descends to all children whose key range determined by the pivot keys overlaps with the requested range. Hence, this algorithm visits all leaves that may contain key-value pairs in the given key range.

2.2.2. Resource and Runtime Analysis

The *query* operation walks down the tree from the root to a leaf and needs access to two nodes at the same time – a parent node and its child node. Therefore, the runtime of this operation is bounded by $2h$ where h is the height of the B-Tree.

Regarding the *insert* and *delete* operation, we also have algorithms that walk down the tree, but we have a bound of $3h$ as they might need to split or rebalance nodes and, hence, need three nodes: the parent and two children.

In summary, the resource requirements have very low bounds for all operations. We will now show that the operations are also "fast" by bounding the height h of B-Trees.

Theorem 2.2.1. The height h of a B-Tree is at most $1 + log_t \frac{N}{2}$ where $N \geq 1$ is the number of key-value pairs in the B-Tree.

Proof. Due to the invariants, the root node contains at least 1 key and all other nodes contain at least $t - 1$ keys. Hence, a B-Tree with height h has at least 2 nodes on depth 1, $2t$ nodes on depth 2, $2t^2$ nodes on depth 3, and so on. Therefore, there are $2t^{h-1}$ leaf nodes. As each leaf node has at least 1 key, we have the following inequality:

$$N \geq 2t^{h-1}$$

By simple algebra, we get $log_t \frac{N}{2} \geq h - 1$. □

We assume that the keys and the child pointers have a constant size so that we can set $t \in \Theta(B)$ (if we fill the nodes up to the block size B). Hence, $h \in O(log_B N)$ holds. It follows that all presented B-Tree operations have a runtime bound of $O(log_B N)$. As a last proof, we will show that the *query* operation is asymptotically optimal.

Theorem 2.2.2. For comparison based index data structures, the lower bound for the *query* operation is $\Omega(log_B(N))$.

Proof. Essentially, we need to find the correct position for inserting our element in an ordered sequence of N elements so that the sequence remains ordered after the insertion. The position has a range of $[1, N+1]$ and has $log_2 N + 1$ bits of information. If we fetch a block from the disk, it contains $c \cdot log_2 B$ bits of information where c is a constant. Therefore, we need to fetch at least $\lceil \frac{log_2 N+1}{c \cdot log_2 B} \rceil \in \Theta(log_B N)$ blocks to find the correct position. □

2.2.3. Evaluation

As shown in the analysis, B-Trees have good performance in theory because all operations run in asymptotic logarithmic time. Additionally, we have seen that there are not any large constants hidden in the asymptotic analysis: Every node is at least half full, every operation needs at most two nodes per level. This results in a good performance in practice for a many workloads. Especially for random queries, B-Trees are very fast.

Another aspect is the complexity of implementing the shown operations which is reasonably low. Due to its simplicity and its age, the B-Tree is well understood. In [Gra11], Graefe discuss many advanced B-Tree concepts and techniques such as concurrency control, optimized representations of B-Tree nodes, variable-length data, and efficient compression of nodes.

For optimizing the B-Tree for certain workloads, we have two tuning parameters: The obvious block size B which controls the B-Tree node size and the non-obvious node size bounds. The chosen node size bounds are used for an easy analysis and can be relaxed so that nodes may contain less keys. This helps avoiding costly splits and rebalances at the cost of space utilization and node fanout. For more information, see the discussion in [Rod07].

The B-Tree node size is a trade-off parameter for range query and (random) query/insert performance. Large nodes have a high fanout and many entries in the leaves so that range queries are fast. But (random) queries and random inserts have a huge read and write amplification as each operation results in one leaf node read or write (for B-Trees that are not tiny). Small nodes have low fanout and less entries so that the tree has a higher depth which directly affects all operations. In practice, a cache is used for the tree which will hold most of the internal nodes so that most operations just need one I/O. Therefore, most implementations have quite small node sizes per default:

- Btrfs: 16 KiB[1]
- PostgreSQL: 8 KiB[2]
- Ext4/XFS: 4 KiB[3]

These node sizes result in good query performance, reasonably good insert performance, and bad range query performance as range queries suffer when the tree is aged. As already mentioned in Section 2.1, the *Disk Access Model* does not depict that the I/O performance depends on the actual access pattern in practice. When the tree is still young, many leaf siblings will be written next to each other on the disk so that a range query involves less random seeking while reading these leaves. Hence, it benefits from the good sequential read performance in this case. As the tree ages, some leaf nodes will be rewritten to other places on the disk. Now, a range query depends on random read performance which is especially low on HDDs.

Random inserts still have a high write amplification for small values. For example, if we insert many 20 byte values with random keys, we need at least one node write (the leaf node) per value which results in a write amplification factor of about 200 for a node size of 4 KiB. It is an inherent downside of B-Trees that inserts are as expensive as queries. This does not need to be the case as we will see in the next section that introduces B^ε-Trees – a refinement of B-Trees with a potentially much higher range query and insert performance while maintaining the optimal query performance.

[1] Since Linux 3.12 (see *man mkfs.btrfs*).
[2] Compile time default page size. Each B-Tree node is stored in one page.
[3] Block size. Each B-Tree node is stored in one block. Ext4 actually uses *H-Tree* for indexing which is a B-Tree derivative.

2.3. B$^\varepsilon$-Tree

Brodal and Fagerberg analyzed the trade-off between insert and query performance in [BF03]. As B-Trees already have an asymptotically optimal query performance, it is interesting to know whether the insert performance can be optimized without losing the optimal query performance. The paper shows two lower bound trade-offs and also proves that these bounds are tight by providing data structures that match these bounds.

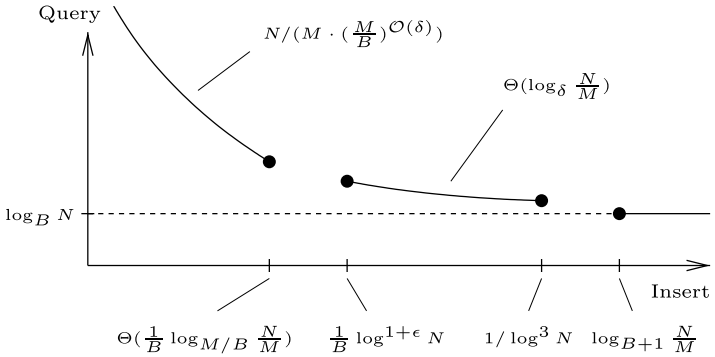

Figure 2.5.: Trade-off curve between query and insert performance for comparison based index data structures [BF03]

In Figure 2.5, we see the resulting trade-off curve. The axes depict the number of I/Os required per operation. The three solid lines denote the performance of data structures that match the trade-off curve. From left to right: truncated buffer tree (a variant of the buffer tree [Arg95]), B$^\varepsilon$-Tree, and the B-Tree. As the curve suggests, the B$^\varepsilon$-Tree has a much higher insertion performance than the B-Tree without sacrificing query performance.

We will now give an introduction to B$^\varepsilon$-Trees similar to the B-Tree as they are derived from the B-Tree and provide the same operations. For a more detailed overview, see "An introduction to B$^\varepsilon$-Trees and write-optimization" [Ben+15].

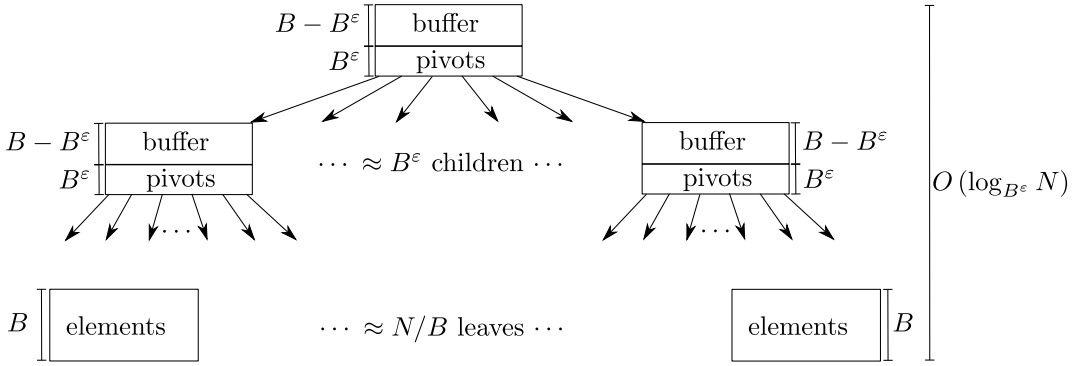

Figure 2.6.: A B$^\varepsilon$-Tree [Ben+15]

A B$^\varepsilon$-Tree supports the same operations as a B-Tree: query, insert, delete, and range-query. It also builds on a similar tree structure, but with an important refinement: In

a B$^\varepsilon$-Tree, each internal node has an associated buffer (see Figure 2.6). The tuning parameter $\varepsilon \in (0, 1)$ selects how much space the buffer and the pivot keys occupy. (For a B-Tree, we essentially have $\varepsilon = 1$.) In these buffers, a B$^\varepsilon$-Tree keeps messages that encode updates to an underlying leaf. The key to improving the insert performance is to delay the actual insertions by keeping them in buffers until they are eventually applied to the leaf. If we have a full buffer on insertion, we *flush* the buffer: We take the messages that correspond to a child and insert them into the child's buffer if it's an internal node or apply it to the leaf.

Let N be the number of key-value pairs in the *B^ϵ-Tree*, B the block size, $\varepsilon \in (0, 1)$, and $t \geq 2$ with $t \in \Theta(B^\varepsilon)$ be the minimum *degree* of the internal nodes. Let $s \in \Theta(B - B^\varepsilon)$ be maximum number of messages in internal nodes and $u \in \Theta(B)$ with $s \leq 2u$ the minimum number of entries in the leaf nodes. The B$^\varepsilon$-Tree has very similar structural invariants to the B-Tree but with some additions for the buffers:

1. Every non-root internal node contains at least $t - 1$ keys and, hence, at least t children. The root node contains at least 1 key.
2. Every internal node contains at most $2t$ keys.
3. Each buffer contains at most s messages.
4. Every non-root leaf node contains at least u entries.
5. Every leaf node contains at most $2u$ entries.
6. All leaf nodes have the same depth.
7. All messages for a key are on the single root-to-leaf path for this key w.r.t. the search order.

We will now describe the operations on the B$^\varepsilon$-Tree. We will start with the basic set of *query* and *insert* to simplify the analysis. Later on, we will show how to implement *delete* using *insert* and introduce *upserts* which enable fast updates to B$^\varepsilon$-Trees.

2.3.1. Operations

Query

Let k be the queried key. Starting at the root, apply the following recursive algorithm:

1. If the current node is an internal node, collect all messages in the buffer that belong to our search key k and select the next child to query w.r.t. the search order.
2. If the current node is a leaf node, look for the corresponding key-value pair and apply all collected messages to it in reverse order (i.e. from old to new). Return the result.

Correctness By obeying the search order we find the corresponding leaf to our search key. Due to invariant 7, we know that we have collected all messages for the search key so that after applying these messages in order, we obtain the correct result.

Insert

1. If the node is an internal node with $2t$ keys or a leaf with $2u$ entries, execute the *Split* operation on the node and add a new internal node as root node with an empty buffer, the pivot key as single key, and the old root node and its sibling as children.

2. If the root node is an internal node and its buffer contains s messages, execute the *Flush* operation on it. If the root node has only one child, set the child as root node, execute the *Insert* operation for each of the messages in the buffer, and go back to step 1.

3. Insert the message if the root node is an internal node or apply the message to the leaf node.

Correctness The algorithm inserts the message at the root node so that this message is the "newest" message on the root-to-leaf path for this key. Before the insertion the algorithm ensures that there is enough space in the root node by splitting in the leaf node case or flushing in the internal node case.

Flush

Given: An internal node with t' keys and s' messages.
Precondition: $t' < 2t$ and $s < s' \leq \frac{2t+1}{2t} s$

1. Select a child node that has the most messages in the buffer in its key range.

2. If the child is a leaf, apply all these messages to the leaf, split if necessary, and return.

3. If the child has $2t$ keys, execute the *Split* operation on the child node, insert the sibling and the split key into this node as in the B-Tree case, and continue with the child node that has the larger buffer in this node.

4. Move as many messages as possible into the corresponding child's buffer without violating the precondition for flushing the child node.

5. If the child node has more than s messages in its buffer, execute the *Flush* operation on the child node.

Correctness The algorithm flushes messages along the corresponding root-to-leaf paths. By selecting a child with the most buffered messages and the pigeon hole principle, we can guarantee that at least $\lfloor \frac{s}{2t} \rfloor$ messages have been selected and will be flushed so that afterwards this node buffer contains at most s messages.

In the child leaf node case, we apply up to $\frac{2t+1}{2t}s$ messages to the leaf. The resulting leaf can have 0 to $4u$ entries so that one execution of rebalance or split fixes this anomaly. Therefore, this node may gain or lose at most one child. Due to the precondition, the node fanout invariants cannot be violated.

In the child internal node case, we flush the messages to the corresponding child's buffer. Because we split the child beforehand, it cannot overflow during the flush process.

Split

Given: A node.
Returns: A new sibling node and a pivot key.
Precondition: Either an internal node with $2t$ keys or a leaf with at least $2u$ and at most $4u$ entries.

1. Split the nodes as described in the B-Tree case.
2. If these are internal nodes, add all messages of the previous node into the buffers so that messages with keys lower or equal to the pivot key belong to the left node's buffer and all other messages to the right one's.

2.3.2. Resource and Runtime Analysis

As for B-Trees, the *query* operation walks along the root-to-leaf path and needs access to a child node when examining the parent node – hence, at most two nodes at the same time. Additionally, this operation collects all messages for the search key on this path which can be up to $s \cdot h$ messages where h is the height of the B$^\varepsilon$-Tree. Again, the runtime of this operation is bounded by $2h$.

The worst case runtime performance for the *insert* operation is also bounded by the height: If all buffers overflow after flushing on the root-to-leaf path for this key, we have to modify all nodes on this path and each of those nodes might split. Resource wise, the *insert* operation needs at most three nodes at any time – in the case of a split.

Much more interesting is the amortized insertion performance of the B$^\varepsilon$-Tree which is significantly better compared to B-Trees. Due to the batched flushes of at least $\lfloor \frac{s}{2t} \rfloor \in \Theta(B^{1-\varepsilon})$ messages, L insertions result in $O(\frac{L}{B^{1-\varepsilon}})$ flushes per level. In total, we have a cost of $O(h\frac{L}{B^{1-\varepsilon}})$ and respectively an amortized insertion cost of $O(\frac{h}{B^{1-\varepsilon}})$ which is much lower than the worst case insertion cost of $O(h)$.

The B-tree height theorem (and proof) (see Theorem 2.2.1) can be applied to B$^\varepsilon$-Tree without any modifications. This implies the following bound:

$$h \leq 1 + log_t \frac{N}{2} \quad \text{for } N \geq 1$$

Before evaluating these results, we will first describe how to implement deletion for B$^\varepsilon$-Trees and introduce the *upsert* operation which benefits from B$^\varepsilon$-Trees.

2.3.3. Deletion

As we use generic messages for insertions, we can also implement the *delete* operation by message insertion. For the deletion of an element, we simply insert a tombstone message. On query, this message will be applied to the previous value and just delete it. Note that we do not actually delete the value in the leaf as this would be too expensive. The actual deletion occurs eventually when the tombstone message is flushed to a leaf. As the leaves can underflow when applying messages that delete data, we may have to rebalance nodes when flushing. Rebalance can be implemented just like for B-Trees expect that we need to rebalance the buffers of internal nodes as well.

Deleting elements by insertion implies that the delete operation has the same runtime bounds as insertion for B$^\varepsilon$-Trees.

2.3.4. Upsert

Many workloads involve a read-modify-write pattern. Mapping this pattern to B-Trees, we have a query followed by an insert. In total, we have a cost of $\Theta(h) = \Theta(log_B N)$ for this pattern. For B$^\varepsilon$-Trees, we could use the same operations to implement this pattern but this would result in the same performance as for B-Trees. In particular, we cannot take advantage of the improved amortized insertion performance because of the (dominating) query performance.

A solution to this problem is the *upsert* operation which encodes an update or insert for a specific key. If the value of this key is already present, it will be updated, otherwise it will be inserted. If the upsert does not depend on any state, we can encode this operation as message and insert it into the tree. On query, the upsert message is applied on-the-fly just like for the tombstone messages. Eventually, the messages will be flushed to the leaf and the actual operation will be applied on the value.

With this concept, we can support the read-modify-write pattern on B$^\varepsilon$-Trees with a single insert operation and, hence, provide a significant speed up compared to B-Trees.

2.3.5. Evaluation

Putting all results together, B$^\varepsilon$-Trees have a very good performance and seem superior to B-Trees. Table 2.2 depicts the theoretical runtimes of B-Tree and B$^\varepsilon$-Tree: The B$^\varepsilon$-Tree with $\varepsilon = 0.5$ has the same asymptotic worst case runtime as the B-Tree for all operations, but improves amortized insert, upsert, and delete by a factor of $\sqrt{B}$. This results in an asymmetry between query and insert performance and therefore the B$^\varepsilon$-Tree is also called a *write-optimized data structure*. B$^\varepsilon$-Trees are especially suited for write heavy workloads.

Regarding implementation complexity, a B$^\varepsilon$-Tree is only slightly more complicated than a B-Tree with the added buffers for internal nodes. As the B$^\varepsilon$-Tree is still quite

	B-Tree	B$^\varepsilon$-Tree	B$^\varepsilon$-Tree with $\varepsilon = 0.5$
Tree height	$\Theta(log_B N)$	$\Theta(\frac{1}{\varepsilon} log_B N)$	$\Theta(log_B N)$
Query	$\Theta(log_B N)$	$\Theta(\frac{1}{\varepsilon} log_B N)$	$\Theta(log_B N)$
Insert, Upsert, Delete (amortized)	$\Theta(log_B N)$	$O(\frac{1}{\varepsilon B^{1-\varepsilon}} log_B N)$	$O(\frac{1}{\sqrt{B}} log_B N)$
Insert, Upsert, Delete (worst case)	$\Theta(log_B N)$	$O(\frac{1}{\varepsilon} log_B N)$	$O(log_B N)$

Table 2.2.: Theoretical comparison of B-Tree and B$^\varepsilon$-Tree

young, it is not so well understood as the B-Tree. Many concepts for B-Trees should be applicable to B$^\varepsilon$-Trees due to their similarity though.

In practice, we have three tuning parameters for a B$^\varepsilon$-Tree: node size bounds, node size, and fanout. As for B-Trees, the presented node size bounds should be relaxed to avoid too frequent splits and rebalances. For node size and fanout, we have a different situation: While the node size was a trade-off parameter for range query and (random) query/insert performance in B-Trees, the insert performance significantly benefits from larger nodes in B$^\varepsilon$-Trees. So only (random) queries tend towards smaller nodes. Hence, most B$^\varepsilon$-Tree operations favor a much larger node size as we have seen for B-Trees. If we increase the node size from 4 KiB to 4 MiB, the theoretical insert performance suggests that we have a speed up of 32 for insert. Range queries perform much better with larger node sizes too as we need much less seeks.

There is only one implementation of B$^\varepsilon$-Trees used in production the author knows of which is *PerconaFT* [Per17].[4] This implementation has a node size up to 4 MiB and a fanout from 4 to 16. By the pigeon hole principle, this ensures that on each flush at least 256 KiB of data are moved. Due to this invariant, writes are efficient on typical storage devices and the maximum write amplification is limited to acceptable levels.

All in all, the B$^\varepsilon$-Tree maintains the worst case performance of a B-Tree, but has a much better amortized insert and upsert performance. In combination with larger node sizes, we can expect to improve insert performance by one or two orders of magnitude compared to B-Tree. Likewise range queries are also faster due to the larger nodes. In practice, the query performance is expected to be slightly slower because of the increased tree height.

Before describing how to build a *key-value store* using B$^\varepsilon$-Trees, we will briefly compare the B$^\varepsilon$-Tree to another very common write-optimized data structure: the Log-Structured Merge Tree (LSM).

[4]Before the acquisition of Tokutek by Percona in 2015, PerconaFT was also known as *TokuFT*.

Data Structure	Insert	Upsert	Query	Range Query
B$^\varepsilon$-Tree with $\varepsilon = 0.5$	$\frac{log_B N}{\sqrt{B}}$	$\frac{log_B N}{\sqrt{B}}$	$log_B N$	$log_B N + \frac{k}{B}$
B$^\varepsilon$-Tree	$\frac{log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{log_B N}{\varepsilon}$	$\frac{log_B N}{\varepsilon} + \frac{k}{B}$
B-Tree	$log_B N$	$log_B N$	$log_B N$	$log_B N + \frac{k}{B}$
LSM	$\frac{log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{log_B^2 N}{\varepsilon}$	$\frac{log_B^2 N}{\varepsilon} + \frac{k}{B}$
LSM + BF	$\frac{log_B N}{\varepsilon B^{1-\varepsilon}}$	$log_B N$	$log_B N$	$\frac{log_B^2 N}{\varepsilon} + \frac{k}{B}$

Table 2.3.: Amortized asymptotic runtime of operations for B$^\varepsilon$-Trees, B-Trees, Log-Structured Merge Trees (LSM) [ONe+96], and LSMs with Bloom filter (BF). Based on [Ben+15]

2.3.6. Comparison of B-Tree, B$^\varepsilon$-Tree, and LSM

Log-Structured Merge Tree is due to Patrick O'Neil in 1996 [ONe+96] and is the most commonly used write-optimized index data structure today. There are multiple variants of LSMs, but all of them feature a very good insert performance and mediocre query performance without optimizations. Table 2.3 compares the amortized asymptotic runtime of various operations. Without a Bloom filter, LSMs have inferior query performance compared to B$^\varepsilon$-Trees and B-Trees. But with a Bloom filter, LSMs cannot provide a fast upsert operation. In addition, the range query performance is inferior. Compared to B-Trees, LSMs improve on insert performance, but lose on query performance. In contrast, B$^\varepsilon$-Trees strictly improve compared to B-Trees.

In practice, LSMs need background work to function. It can be quite tricky to ensure that the background work is done fast enough as "[...] existing log structured techniques improve write throughput but sacrifice read performance and exhibit unacceptable latency spikes" [SR12]. B$^\varepsilon$-Trees do not need any background work and, hence, are probably easier to implement and tune for high performance.

2.4. Storage Stack

A *storage stack* is a stack of software layers and hardware components for managing data. At the top layer, a storage stack provides an interface to manage the data. Most of the time, this is a POSIX-compatible filesystem. At the bottom of the stack, there are multiple storage devices such as hard disks and SSDs. There are a number of key features every storage stack should provide:

Scalable performance The throughput of the storage stack should increase nearly linear with the number of underlying storage devices. Depending on the actual configuration, this might affect only read or write throughput.

Data redundancy There are many hardware components in every storage stack that

may fail: storage controllers, cables, and most notably hard disks. Regarding hard disks, recent studies suggest that up to 10% per year experience failures [Bai+08]. Therefore, a storage stack needs to save data redundantly so that it is resilient against disk failures. As the disk capacity keeps growing faster than the disk throughput, the demand for double or triple redundancy rises [Lev10]. Especially during reconstruction data loss is quite likely as "[a] significant number (8% on average) of corruptions are detected during RAID reconstruction" [Bai+08]. Hence, the storage stack should support at least double parity.

Multiple data sets The available space of storage stack should be partitionable into multiple data sets so that the user can group their data into logically separated sets.

Dynamic configuration The storage stack should be configurable after the initial creation. This involves adding more storage devices and managing data sets such as creating and deleting data sets and changing their size.

Snapshots Each data set should be snapshottable on its own so that the state of a data set at a specific point in time can be preserved. This is especially useful for preventing data loss due to deleting or overwriting data by accident. It should be possible to create multiple snapshots and to rollback a data set to an older state specified by a snapshot. Regarding good space efficiency, the snapshots should be copy-on-write.

Easy administration The storage stack administration should be easy and the administrative commands should prevent accidental misuse and data loss. This includes for example comprehensive documentation, command completion on the shell, and an option to "dry run" commands.

There are also features which are less commonly implemented in older storage stacks but are very appreciated today due to the ever increasing data volume:

Data integrity The storage stack should guarantee data integrity by checksumming data. Checksum mismatches occur as often as 0.5% per disk and year [Bai+08]. Without checksumming data, these data losses remain undetected.

On-disk consistency Due to the increasing data volumes, the runtime of filesystem checks increases drastically. If a storage stack (or filesystem) can guarantee on-disk consistency, no such checks are needed so that the startup time after a crash is greatly reduced.

Clones A clone is a writable snapshot. Together with copy-on-write, clones enable efficient data sharing, for example between multiple VMs which are based on the same disk image.

Send/Receive The term *send/receive* stands for transferring data sets or snapshots to another storage pool. Due to copy-on-write snapshots storage stacks can compute the delta between snapshots very efficiently and, hence, significantly reduce the amount of data transferred. This feature is very useful for incremental data backups.

Caching A storage stack could cache "hot" data on separate fast storage devices such as SSDs to increase the performance. Especially, the latency of read requests can be reduced when fetching data from SSDs instead of hard disks.

Sharing free space between data sets For administrators, it is very convenient when all data sets share their space on an underlying storage pool so that the free space of each data set does not need to be managed by hand.

Quota Likewise, it is useful when the storage stack accounts the space used by users, groups, or projects so that the data consumption can be limited by quotas.

Compression The storage stack should support the transparent compression of user data as this is a very simple but effective data reduction technique.

Authenticated encryption As an extension to data checksumming, a storage stack could use authenticated encryption to provide confidentiality and authenticity of the data in addition to the data integrity. Because the storage stack has to save a checksum per data block anyway, adding metadata for encryption does not involve a huge overhead in metadata size or management. This feature is very useful for mobile computer as it protects against loss or theft. In contrast to full disk encryption, this concept guarantees authenticity of the data.

In the next chapter, we will introduce the storage stack of this thesis and then in the following chapter, we will present two very popular storage stacks and compare the design of these two against the design of this storage stack.

3. Design

In this chapter, we will describe and discuss how to build a storage stack with key-value store interface using the B^ϵ-Tree as the basic building block. The presented storage stack provides data integrity and supports advanced techniques such as multiple data sets, snapshots, and multiple underlying storage devices using different strategies such as striping, mirroring, and parity. We will start off with a short notation definition for describing interfaces and a conceptual overview over the layered storage stack. Then, each layer of the storage stack will be described in more detail.

3.1. Notation

Throughout the *Design* chapter, we will use a notation for interfaces and types which is based on the syntax of the Rust programming language. We have three different kinds of passing a object to function: by-value, by-reference, and by-mut-reference. These are represented by `T`, `&T`, and `&mut T`. By-value parameters transfer the ownership of the object `T`. By-reference and by-mut-reference pass a reference to the callee. When using by-reference and by-mut-reference, the owner retains the ownership. We can only construct a by-mut-reference to an object if we have unique, mutable access to the object. When accepting by-reference, the object may not be mutated by the callee. Function signatures are declared as follows:

Interface

```
function_name(name: Type) -> ReturnType
f2(p1: &T, &mut U) -> &mut V
```

The first line declares a function named `function_name` which has one parameter with the name `name` and the type `Type`. The function will take the ownership of value passed as parameter `name` and returns a value of type `ReturnType` whose ownership is transferred to the caller. The second line defines a function `f2` with two parameters which are passed as references. The object of type `T` passed as parameter `p1` may not be modified by `f2`, but the object passed as the second unnamed parameter may be modified by `f2`. The caller retains the ownership of both objects. `f2` returns a mutable reference to an object of type `V` which may be modified by the caller. The caller does not own the object.

A struct definition looks like the following:

Type

```
struct StructName {
  field_name: FieldType,
  AnonymousFieldType,
}
```

Note that the anonymous field types are solely used for convenience and Rust does not actually support unnamed fields. In addition to structs, Rust also supports tagged union types whose instances are one of the defined variants:

Type

```
enum EnumName {
  VariantOne,
  VariantTwo(FieldType),
}
```

3.2. Conceptual Overview

This storage stack uses multiple layers to reduce the complexity of the design and to simplify the reasoning for each component (see Figure 3.1). Many concepts shown in this chapter are heavily based on ZFS which will be presented in detail later on in Section 4.2.

At the bottom of the storage stack, we have the *Vdev Layer* (short for *virtual device*) and the *Storage Pool Layer* (SPL). The vdevs are responsible for reading and writing data to the disks. A vdev can either be a single disk, a mirror of multiple disks, or a group of disks with parity data – similar to RAID5 and *raidz1* in ZFS. The SPL is responsible for striping the data over the vdevs. Like for ZFS, the lower layer already makes use of checksums to provide data integrity and advanced data recovery options. The SPL provides a block device like interface but with checksums.

On top of this interface is the *Data Management Layer* (DML). The DML handles generic objects which are identified by an *object reference*. The DML caches often accessed objects and tracks modifications of objects. Modified objects will be written back when needed. Write back includes compression and allocation of disk space. The DML employs the redirect-on-write technique. For allocation, the DML calls an *allocation handler* which provides access to the allocation bitmaps.

The *Tree Layer* manages multiple B^ε-Trees. Each B^ε-Tree node is tracked as a DML object. Each B^ε-Tree has a message based key-value interface with byte sequences as keys and values. The length of keys and values and the size of messages is limited to reasonable sizes: A key should be at most 1024 bytes long. Values and messages should be at most 128 KiB of data. Each message may contain arbitrary data and execute arbitrary code on application to data. Additional to querying data for a key and inserting

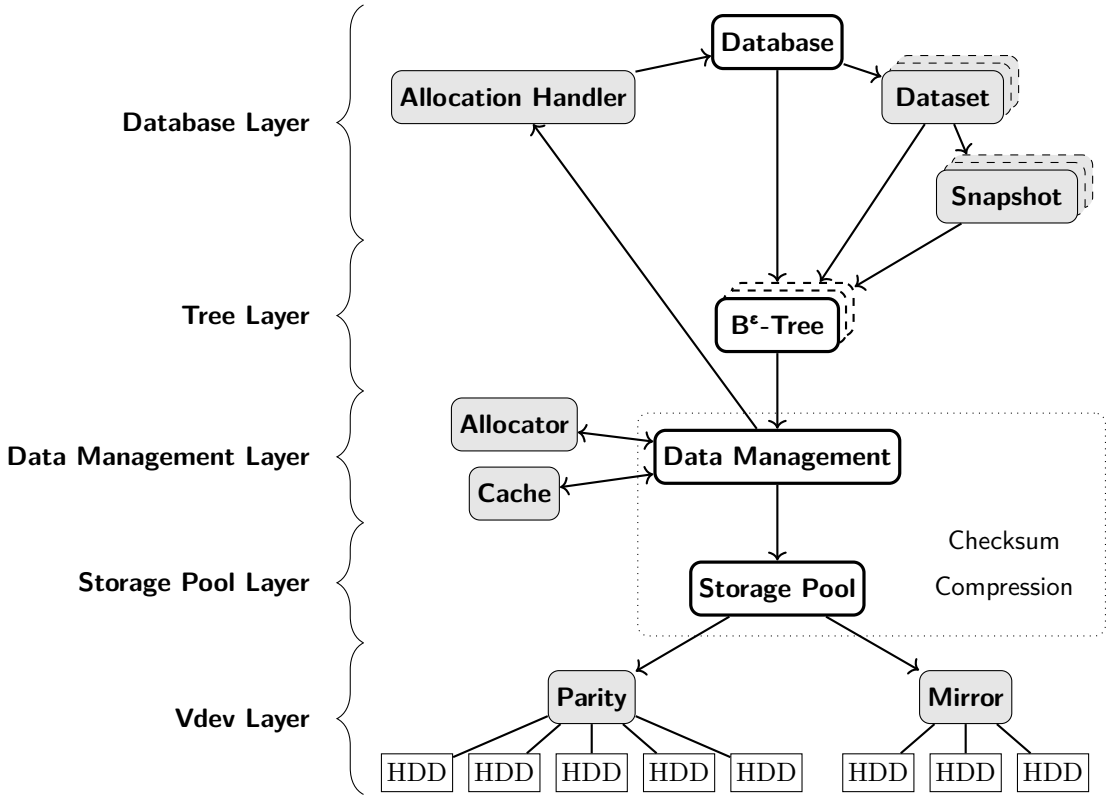

Figure 3.1.: Layer concept of the key-value store

a message for a key, the tree also provides a range query and a range delete operation which are more efficient for large key ranges.

The top-most *Database Layer* is built on top of the B$^\varepsilon$-Trees. This layer manages multiple data sets which can be individually snapshotted. Each data set provides a key-value store interface and is represented by a B$^\varepsilon$-Tree. Likewise, snapshots also provide a (read only) key-value store interface and are represented by a snapshot of the data set's B$^\varepsilon$-Tree. There is also a root B$^\varepsilon$-Tree in which the allocation bitmaps and the information about the data sets and the snapshots are stored.

There are two key concepts of this storage stack: First, we will use the B$^\varepsilon$-Tree as the sole index data structure to again reduce the complexity of this design. Having to manage only one data structure greatly reduces the amount of code in the implementation. Second, by applying the path-copying technique [Dri+86], we have copy-on-write B$^\varepsilon$-Trees which enable efficient snapshots.

In contrast to the two presented storage stacks later on, this storage stack does not provide a POSIX compatible filesystem interface. A POSIX filesystem has many operations which must be supported and strict semantics regarding atomicity of concurrent reads and writes. Additionally, userspace filesystem implementations are quite slow due to the context switching and mode switching overhead as the kernel has to switch to our userspace filesystem process for each syscall. See [VTZ17] for a detailed performance analysis of FUSE, the userspace filesystem framework used on Linux. Due to the com-

plexity and the performance issues, this storage stack will instead export a key-value store like interface which is still very similar to a filesystem but with a much smaller and cleaner interface.

In the rest of this chapter, we will focus on each layer to depict and discuss the design and implementation in greater detail.

3.3. Vdev Layer

As already mentioned in the introduction, the Vdev Layer is responsible for reading and writing data from/to one or more block devices. It operates on 4 KiB block size and provides the following interface to the upper layers, excluding error handling for brevity:

Interface

```
 read(size: Block, offset: Block, Checksum) -> Data
scrub(size: Block, offset: Block, Checksum) -> Data
write(Data, offset: Block)
flush()
actual_request_size(size: Block) -> Block
size() -> Block
```

`Block` is an unsigned integer which denotes either a size or an offset in multiples of the block size. `Data` is a pointer to a byte sequence with a length that is a multiple of the block size. A `read` request will read the data located at the given `offset`, verify the data with the given `Checksum` data and return `size` blocks of data. A `scrub` request is very similar but it guarantees that all data blocks corresponding to this request are actually read and verified. A `write` request will write `actual_request_size(data.size)` blocks of data to the given `offset`. `flush` will flush any caches and `size` returns the size of the vdev.

This interface has two significant differences to the block device interface. First, a read request includes a checksum of the data so that the data integrity can be verified by the vdev. Second, if a vdev needs any additional data such as parity data, it is visible to the upper layer due to the *actual request size*. In combination with redirect-on-write in the upper layers, this enables a RAID5 implementation without the "RAID5 write hole" as we will see later on.

In the following we will describe three actual vdev implementations: single, mirror, and parity1. A *single* vdev only manages a single block device as underlying storage. A *mirror* will mirror all data to the underlying block devices. Last but not least, *parity1* will stripe data to the underlying block devices and use parity data to be resilient against one faulty block device.

3.3.1. Single Vdev

The implementation of the single vdev is straightforward. As this vdev does not need any additional data per request, `actual_request_size` always returns the given size. The rest of the operations will be directly mapped to the block device operations:

read Perform a read with `size` and `offset` on the underlying block device and verify the returned data with `Checksum`.

scrub Same as read.

write Write the `data` to the given `offset` on the block device.

flush Call flush on the block device.

size Query the size in blocks of the block device.

3.3.2. Mirror Vdev

The mirror vdev consists of at least two block devices and mirrors the data to all block devices. It has the same size as the smallest underlying block device. If a mirror vdev consists of n block devices, it is resilient against $n - 1$ faulty block devices.

`actual_request_size` returns the given size as this vdev does not need any additional data. The rest of the operations is slightly more tricky to implement compared to the single vdev:

read Select any block device and try to perform a read with `size` and `offset` on the block device and verify the returned data with the `Checksum`. On success, return the data. Otherwise sequentially try to read from the other block devices and verify the data. If any read and verify succeed, try to rewrite all failed block devices and then return the data.

scrub Similar to read, but read from all block devices and verify all data blocks.

write Write the `data` to the given `offset` on all block devices. This operation succeeds if at least one write operation succeeds.

flush Call flush on all block devices.

size Query the size in blocks of the block devices and return the minimum.

3.3.3. Parity1 Vdev

The parity1 vdev consists of at least three block devices and stripes the data with additional parity blocks so that it is resilient against one faulty block device. This vdev is very similar to RAID5, but with two major differences: The offsets are not remapped to hide the parity data but the parity data location is visible to the upper layer. In addition,

the stripe width is not fixed and depends on the data size so that each request is always a full stripe. The upper layer has to call `actual_request_size` prior to writing data to determine the actual size on disk for a specific data size. On each read or write request, the vdev has to calculate the geometry of the stripe with the given offset and size and with the disk count.

Example: Table 3.1 shows a parity1 vdev with 5 disks. Each stripe is depicted by a different color. As we have five disks, we need one parity block per four data blocks. For this implementation, the parity blocks for every stripe are located on the disk at which the stripe begins. If we have a stripe with a number of data blocks which are not a multiple of four, we lose some space efficiency – like for the second stripe. Regarding the mapping between data segments and stripes, the first stripe corresponds to a data segment of 8 blocks at offset 0. The second stripe is a data segment of three blocks at offset 10. The last stripe is a 10 block data segment at offset 21.

In the following we will show how to calculate the stripe geometry for a request, how to build the parity, and how to recover data. Afterwards, we sketch how to implement the vdev interface. In the end, we discuss possible extensions to this concept.

Stripe Geometry

Let B be the sequence of data blocks for the request, s be the data size (in blocks), o the offset, and n the number of disks. With these parameters, we can calculate the stripe geometry using p the number of parity blocks required and q the number of disks which have long data chunks.

$$p := \lceil \frac{s}{n-1} \rceil$$

$$q := \begin{cases} s \bmod n-1 & \text{if } s \bmod n-1 \neq 0 \\ n-1 & \text{otherwise} \end{cases}$$

$$d_i := (o+i) \bmod n$$

$$o_i := \lfloor \frac{o+i}{n} \rfloor$$

$$s_i := \begin{cases} p & \text{if } i \leq q \\ p-1 & \text{otherwise} \end{cases}$$

$$B_i := B[\hat{o}_i..\hat{o}_i + s_i] \qquad \text{where } \hat{o}_i := \sum_{j=1}^{i-1} s_j$$

d_i, o_i, s_i represent the i-th disk index, disk offset, and size where disk d_0 contains the parity data of size s_0 at offset o_0. Likewise, the disks with the index d_1 to d_{n-1} contain the data chunk B_i of size s_i at offset o_i. Hence, the concatenation of the data chunks from the disk with index d_1 to d_{n-1} is the actual data.

Disk Block Offset	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
0	P_0	D_0	D_2	D_4	D_6
1	P_1	D_1	D_3	D_5	D_7
2	P_0	D_0	D_1	D_2	P_0
3	D_0	D_1	D_2	P_0	D_0
4	D_1	P_0	D_0	D_3	D_6
5	D_8	P_1	D_1	D_4	D_7
6	D_9	P_2	D_2	D_5	–

Table 3.1.: Example block layout for a parity1 vdev consisting of five disks. Each stripe is depicted by a different color. (Inspired by a blog post of Matthew Ahrens [Ahr])

Parity Generation and Recovery

The parity data blocks P are generated by xor-ing all disk data chunks: $P := \oplus_{i=1}^{n-1} B_i$.

If we want to recover the data chunk of a failed disk with index $d_j, j > 0$, we can xor the parity blocks with the remaining data chunks:

$$B_j := P \oplus (\oplus_{i=1}^{j-1} B_i) \oplus (\oplus_{i=j+1}^{n-1} B_i)$$

After recomputing B_j, we can verify the data using checksum. If the verification fails, at least one other disk returned faulty data and we have to give up.

During the scrub, the disk with index d_0 might fail which contains the parity data for this stripe. In this case, we have to recompute and rewrite the parity data.

If all read operations succeed, but the data does not match the checksum, we can try a combinatorial reconstruction: For each data disk, we assume that it failed and try to reconstruct the data using the normal recovery process. If the checksum matches for any of these, we know which disk returned faulty data, and we can proceed as before.

Implementation

This is the only vdev for which `actual_request_size` does not return the given size as we have to save additional parity data:

$$\texttt{actual\_request\_size}(s) = s + p \qquad \text{where } p := \lceil \frac{s}{n-1} \rceil$$

read Reconstruct the stripe geometry and read all data chunks (but not the parity blocks). If the checksum verification succeeds, return the data. Otherwise fetch the parity blocks and try to recover the data. If this succeeds, try to rewrite the failed block device and then return the data.

scrub Similar to read, but always fetch and verify the parity blocks.

write Derive the stripe geometry, build the parity blocks, and write the data and the parity to disk. This operation succeeds if at most one write operation fails.

flush Call flush on all block devices.

size Query the size in blocks of the block devices and return the minimum multiplied by the number of disks.

Possible Extensions

The concept of *parity1* can be extended to vdevs with more than one parity disk per stripe so that these vdevs are resilient against multiple faulty block devices. A downside of multiple parities is that generating parties beyond the first parity is not trivial and becomes significantly more computing intensive. For more information, see [Anv07] which gives an introduction to the Linux RAID6 implementation.

Due to the increased complexity in generating parities and recovering from errors, the implementation is left to future work.

3.4. Storage Pool Layer

The Storage Pool Layer is a rather thin layer that manages one or more vdevs. The vdevs are handled like a JBOD, i.e. they are an independent group of vdevs. This layer roughly provides the same interface as the Vdev Layer, but with a different offset type: the *DiskOffset*. This type includes a vdev ID and a block offset so that this layer knows on which vdev the data resides. The upper layer is responsible for sharding the data across all vdevs for good performance.

Contrary to vdevs, this layer also guarantees sequential ordering for reads and writes on the same *DiskOffset* so that no barriers are needed for most operations of the upper layer.

3.5. Data Management Layer

The Data Management Layer (DML) manages objects for the upper layer and is responsible for caching these objects in memory, tracking the modifications to objects, and the write-back of modified objects. This layer uses copy-on-write, i.e. it will never overwrite active data. Additionally, it supports compressing the on-disk objects.

An unmodified, on-disk DML object is represented by an *ObjectPointer*:[1]

Type

```
struct ObjectPointer {
  DiskOffset,
  compressed_size: Block,
  Compression,
  Checksum,
  Generation,
  Info,
}
```

This structure contains all information needed to read the object from disk (`DiskOffset`, `compressed_size`, and `Checksum`) and unpacking the object from the data (`Compression`). The `Generation` field is used by the upper layer for tracking the age of an object. The `Info` can be used to tag the object with additional information.

The upper layer can access DML objects by a handle called *ObjectReference*. This handle is an enumeration type and represents the states of an object in which it was encountered at the last access (i.e. it is lazily updated):

Type

```
enum ObjectReference {
  Unmodified(ObjectPointer),
  Modified(ModifiedNodeId),
  InWriteback(ModifiedNodeId),
}
```

So, a DML object can be either unmodified, modified, or in write-back. In the unmodified case, the object is identified by the *ObjectPointer*. In the other cases, it is identified by a *ModifiedNodeId* which is just a unique number for each modified node. An important invariant of the *ObjectReference* is that there is only one *ObjectReference* pointing to a mutable object.

[1] The ObjectPointer is very similar to the block pointer in ZFS.

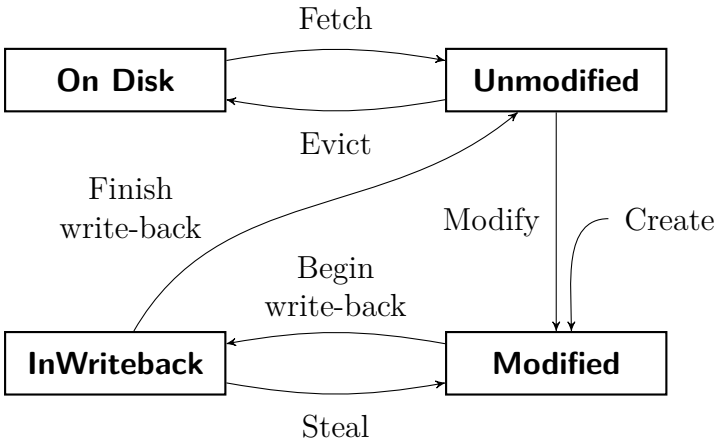

Figure 3.2.: DML object state cycle

The states of an *ObjectReference* correspond to the actual object state in the cache of the DML. The object states can change according to the state cycle (see Figure 3.2). Every DML object starts in the *Modified* state. From this state, it transitions into the state *InWriteback* by starting the write-back to disk. Then, it can be either stolen (if mutable access to the object is requested during the write-back) or the write-back finishes and the object is now unmodified. Unmodified objects can be evicted from the cache and fetched into the cache. Finally, if mutable access is requested, the object transitions back into the modified state.

In the cache, objects are tracked with a unique key called *CacheKey*. This struct can be derived from an *ObjectReference* and is defined as follows:[2]

Type

```
enum CacheKey {
  Unmodified(DiskOffset),
  Modified(ModifiedNodeId),
  InWriteback(ModifiedNodeId),
}
```

As the *ObjectReference* is lazily updated, we encounter situations where the object is not present in the cache under the corresponding *CacheKey*. Due to the object state cycle, we can identify into what states the object must have transitioned: If the object was unmodified and is not present, it must have been evicted from cache and we can fetch it. If the object was modified and is not present, it transitioned to the *InWriteback* state. If the object was in that state and is not present, the write-back must have finished. The object cannot be stolen as this *ObjectReference* is the only one pointing to the object by the *ObjectReference* invariant.

[2]Note that the `DiskOffset` is a unique identifier for unmodified objects as this layer will never overwrite active data due to copy-on-write.

Despite the quite complex state handling, the interface provided by this layer is simple, again excluding error handling for brevity:

Interface

```
insert(Object, Info) -> ObjectReference
get(&mut ObjectReference) -> &Object
get_mut(&mut ObjectReference) -> &mut Object
remove(ObjectReference)
evict()
write_back(&mut ObjectReference) -> ObjectPointer
try_get(&ObjectReference) -> Option<&Object>
try_get_mut(&ObjectReference) -> Option<&mut Object>
```

Note: `Option<T>` is either a `T` or null.

If the upper layer intends to modify an object, it has to call `try_get_mut` or `get_mut` so that the DML can track that this object is modified and needs to be written back to disk later on. A write-back can be forced by calling the `write_back` method which returns an *ObjectPointer*. The upper layer should regularly call `evict` to evict excessive cache entries. `try_get` and `try_get_mut` are similar to `get` and `get_mut`, but only need an immutable reference to the *ObjectReference* and fail if the corresponding object is not in the correct state or not present in cache.

As the DML needs additional information about the objects, the upper layer has to implement the following interface for the objects:

Interface

```
get_size_in_bytes(&Object) -> Integer
pack(&Object) -> Data
unpack(Data) -> Object
get_child_obj_refs(&mut Object) -> Iterator<&mut ObjectReference>
```

A DML object may contain references to other objects which can be accessed by the DML by calling `get_child_obj_refs`. On write-back, the DML has to ensure that the dependent objects are written back beforehand as the on-disk object will contain the *ObjectPointers* of its children.

Before the actual write-back of an object can happen, the DML has to allocate space first as we do not overwrite active data. The whole space of the Storage Pool Layer is partitioned into segments which are identified by an *SegmentId*. Each of these segments has an allocation bitmap which represents the free space in this segment at a block granularity. The upper layer is responsible for persisting these allocation bitmaps. Hence, this layer needs a callback for reading and updating allocation data which is provided by the *Allocation Handler*. This handler provides the following interface:

Interface

```
get_allocation_bitmap(SegmentId) -> Bitmap
allocate_at(DiskOffset, Size)
copy_on_write(&ObjectPointer)
current_generation() -> Generation
```

The callback `get_allocation_bitmap` shall return the segment bitmap where each active block is marked as in use. The DML calls `allocate_at` on allocation so that the upper layer can update the bitmap. When a DML object transitions to *Modified* via *Modify* or *Stolen* (see Figure 3.2), the DML calls `copy_on_write` to signal that an object will be copied for modification. The upper layer has to decide whether the old object is still in use (for example, due to a snapshot). If it is not in use, it must free the allocated space. Note that this space is not immediately available for allocation as the data is still active until the upper layer's state has been synchronized to disk. The last callback `current_generation` shall return the current generation of the upper layer. On write-back, the DML will query the current generation and saves it in the generation field of the *ObjectPointer* of the object.

3.5.1. Implementation

In the following we will outline how to implement this layer's interface and give short notes on correctness. The correctness of the algorithms is closely related to the object state cycle.

`get(&mut ObjectReference) -> &Object`

1. Check whether the object is in the cache. If present, return the object.
2. If the *ObjectReference* is in *Unmodified* state, fetch the object, insert it into the cache, and return the object.
3. Call `drive_state_transition` and go back to step 1.

Note If an object is not in *Unmodified* state, it must be cached. As the *ObjectReference* is lazily updated, it may not reflect the actual state of the object. Hence, if the object is not present and the *ObjectReference* is not in *Unmodified* state, the *ObjectReference* is not up-to-date and we can call `drive_state_transition` to update it.

`drive_state_transition(&mut ObjectReference)`

Precondition: *ObjectReference* is not in *Unmodified* state and the object is not present in the cache.

1. If the *ObjectReference* is in *InWriteback* state, get the *ObjectPointer* of the written back object identified by the *ModifiedNodeId* and transit to the *Unmodified* state.

2. If the *ObjectReference* is in *Modified* state, transit to *InWriteback* state.

Note If the precondition is satisfied, the *ObjectReference* is not reflecting the object state and needs to be updated. Due to the "unique access to *ObjectReference*" invariant, the *Modify* and *Steal* transitions cannot occur concurrently. Therefore, there is only one possible object state transition. After up to two calls, the *ObjectReference* state will match the state of the object. (*Modified* $\rightarrow$ *InWriteback* $\rightarrow$ *Unmodified*)

`get_mut(&mut ObjectReference, Info) -> &mut Object`

1. If the *ObjectReference* is in *Modified* state: If the object is in the cache, return the object. Otherwise call `drive_state_transition`.

2. If the *ObjectReference* is in *InWriteback* state and not present in the cache, call `drive_state_transition`.

3. If the *ObjectReference* is in *Unmodified* state:
   - If the object is not present in the cache, fetch the object.
   - Call the handler callback `copy_on_write`.

4. Allocate a new *ModifiedNodeId*.

5. Change the *CacheKey* of this object and the *ObjectReference* to *Modified* using the new *ModifiedNodeId*.

6. Return the object.

Note After the first step, the object cannot be in *Modified* state. After the second step, the object is either in *Unmodified* state or it is in *InWriteback* state and present in cache. After the third step, the object is present in cache. In step 5, the object state transitions via *Modify* or *Steal*. Therefore, the object is in *Modified* state.

If the object was *Unmodified*, `copy_on_write` will be called immediately. Otherwise, the object was *InWriteback* and the `copy_on_write` call will be called by `evict`.

`evict()`

1. Select a new object in the cache for eviction. Return if all objects have been already selected.

2. If the object is in *Unmodified* state, remove the object from the cache and return.

3. If the object is in *InWriteback* state, go back to step 1.

4. Call `handle_write_back`. Go back to step 1 if the write-back is unsuccessful.
5. Check if the object is still present in the cache.
6. If yes, save the *ObjectPointer* identified by the *ModifiedNodeId*.
7. If not, call `copy_on_write`.

Note The algorithm ensures that objects are only evicted if they are in *Unmodified* state. If an object is in *Modified* state, we will try to write back the object to disk via the `handle_write_back` call. This might fail if the object points to other objects which are not in *Unmodified* state. After the write-back, we check if the object is still present under the same key. If it is not, it must have been *stolen*. In this case, we will call `copy_on_write` as the `get_mut` cannot have done this due to the missing *ObjectPointer*.

`handle_write_back(CacheKey, &mut Object) -> ObjectPointer`

Precondition: *CacheKey* must be in *Modified* state.

1. Call `Object.get_child_obj_refs()`. For each *ObjectReference*:
   a) If in *Unmodified* state, continue.
   b) If not present in the cache, call `drive_state_transition` and go to step a.
   c) Otherwise signal an unsuccessful write-back because of the child identified by the *ObjectReference*.
2. Change the *CacheKey* to *InWriteback*.
3. Pack the object and compress the data.
4. Allocate.
5. Write the data to disk.
6. Return *ObjectPointer*.

Note Before writing back an object, we check whether all objects referenced by this object are in *Unmodified* state. This involves updating outdated *ObjectReferences*.

`write_back(&mut ObjectReference) -> ObjectPointer`

1. If the *ObjectReference* is in the state *Modified* and the object is not present in cache, call `drive_state_transition`.
2. If in the state *InWriteback*: If present in cache, wait for write back. Call `drive_state_transition`.
3. If in the state *Unmodified*, return the *ObjectPointer*.

4. Retrieve a mutable reference to the *Object* from the cache and call `handle_write_back`.
5. If the write-back was successful, transit the *ObjectReference* to the state *Unmodified* and return the *ObjectPointer*.
6. Otherwise, call `write_back` with the child's *ObjectReference* and go back to step 4.

Note After the first two steps, the *ObjectReference* is up-to-date. If the object is not in *Unmodified* state, we try to write back the object. If this fails due to modified child objects, we can call `write_back` on these. The precondition is satisfied as we have unique, mutable access to the *ObjectReference* of the parent object and the *ObjectReference* invariant guarantees that the parent is the sole owner of the child *ObjectReference*.

3.5.2. Concurrency Control

If we want concurrent access to DML objects, we only need to protect the *ObjectReferences* with locks. If we use read-write locks for *ObjectReferences*, we can call `try_get` and `try_get_mut` as shortcuts before calling `get` and `get_mut`. This increases the efficiency and concurrency if the object is present in cache as these calls only need read access to the *ObjectReference*.

3.6. Tree Layer

The Tree Layer implements B$^\varepsilon$-Trees on top of the DML layer. Each B$^\varepsilon$-Tree node is represented as a DML object. Each B$^\varepsilon$-Tree is identified by a *TreeId* which is chosen by the upper layer. The *TreeId* is injected as *ObjectInfo* into each DML object that belongs to the tree. A *Tree* object holds an *ObjectReference* to the root node of the tree and each internal node holds *ObjectReferences* to its child nodes. This layer does not provide an *Allocation Handler* for the DML.

The B$^\varepsilon$-Tree holds key-value pairs in the leaves and key-message pairs in the internal node buffers. Keys, values, and messages are arbitrary byte sequences and transparent to this layer. Keys are ordered lexicographic. The upper layer provides a *message action* which defines the message behavior:

Interface

```
apply(Key, Message, Option<Value>) -> Option<Value>
apply_to_leaf(Key, Message, Option<Value>) -> Option<Value>
merge(Key, upper: Message, lower: Message) -> Message
```

Note: `Option<Value>` is either a `Value` or null.

`apply` will be called during queries whenever a message has to be applied to the optional underlying value. When a message is flushed to a leaf, the messages will be transformed into an optional value by calling `apply_to_leaf`. When a message is

inserted into an internal node which already contains a message for the key, the messages are merged with `merge`. All in all, the upper layer has full control over the message functionality and can even execute code when a message is finally applied to a leaf which might be useful for lazy accounting.

This layer provides a default implementation of a *message action* which supports the following message types:

`insert(Data)` Unconditionally overwrites the value with `Data`.

`delete` Unconditionally deletes the value.

`upsert(offset: Integer, Data)` Either inserts `Data` into the value if present or constructs a new value. `Data` will be located at the given `offset`. The value will be zero padded to `offset`.

This layer provides the following interface, again excluding error handling for brevity:

Interface

```
new(TreeId) -> Tree
open(TreeId, TreePointer) -> Tree
sync(Tree) -> TreePointer
insert(Tree, Key, Message)
get(Tree, Key) -> Option<Value>
range(Tree, KeyRange) -> Iterator
range_delete(Tree, KeyRange)
```

A *Tree* can be created with `new` or an existing tree can be opened with `open`. The upper layer has to pass a `TreePointer` to `open` which contains an *ObjectPointer* to the root node. This pointer can only be acquired by calling `sync` beforehand which writes all nodes of this tree to disk. Hence, the upper layer is responsible for saving the `TreePointer` on sync so that this tree can be reopened later on. Using `insert`, we can insert messages into the tree. For retrieving values, there are two methods: `get` and `range`. `get` retrieves a single value and returns null if it is not present. `range` can be used for iterating over key-value ranges and is more efficient than calling `get` many times. Deletion of entries can be accomplished by inserting deletion messages. This needs support for a special message type in the *message action* of the upper layer which unconditionally deletes values. When deleting a range of keys, calling `range_delete` will be much more efficient than inserting one deletion message per key.

3.6.1. B$^\varepsilon$-Tree Improvements

This implementation of a B$^\varepsilon$-Tree vastly differs from the original description of a B$^\varepsilon$-Tree in Section 2.3 and contains many improvements so that the B$^\varepsilon$-Tree can be used more easily and efficiently in practice.

Variable-Sized Data

The implementation supports variable-sized keys, values, and messages. For the node size, we use the actual on-disk space a node consumes. This is a very natural way of defining a node's size as transferring data from and to disk is the dominating cost. Compared to uniform sizes, this allows for greater flexibility for the upper layer and better space efficiency if the data size significantly differs among keys and/or values.

Message Merging

Due to the *message action* interface, we can merge any two messages into one message. This increases the space efficiency when a value for a specific key is often updated: In the case of inserting many small messages with the same key into the tree, these messages can be merged into one message. If these messages only overwrite data, the final message will be as small as the last message. Additionally, we only have to save at most one message per key in each internal node which simplifies the B^ε-Tree data structures and algorithms.

Retroactive instead of Proactive Policies

Instead of proactively flushing and splitting a node, we fix a too large or too small node after the operation. As we pin a modified node in memory during these operations, the too large node cannot be written back to disk in the meanwhile so that we never write anomalous nodes. Fixing the invariants after an operation occurred simplifies the algorithms and makes them easier to understand.

Tree Shrinking

For B^ε-Trees, the support of delete operations was only outlined in Section 2.3. This implementation actually supports rebalance between nodes and the shrinking of the tree. To simplify the algorithms, only leaf nodes will be rebalanced. Internal nodes with too low fanout will be merged unconditionally.

Additionally, a range of keys can be deleted very efficiently with the `range_delete` operation which is implemented by walking the parts of the tree that might contain keys in the given range. If the algorithm can conclude that a subtree only contains keys which are in the range, the whole subtree will be deleted. On subtree deletion, leaf nodes do not have to be fetched from disk which yields a huge performance boost over a naive implementation.

Relaxed Node Size Bounds

Rebalancing and merging nodes is quite expensive because we have to fetch and modify the sibling node. Therefore, we increase the allowed node size bounds so that the maximum node size is four times the minimal node size. With these larger bounds, rebalances and merges occur less often.

Tree Walk before Insertion

This optimization relies on caching. Previously, we always inserted messages at the root node and then flushed messages down the tree. When inserting many messages into the same key range, this will result in many flushes into the same subtree over and over again. In this implementation, we will walk down the tree as long as the nodes are already modified and do not contain a message with the same key and then insert the message into the final node. This is efficient as we already have to "pay the cost" for modified nodes: fetching the node and writing it back to disk. This optimization is especially useful for large sequential insertions because the whole path down to the leaf will be modified and the messages would have been flushed to the leaves anyway.

Node Split Depends on Minimum Flush Size

The second optimization also improves sequential insert performance by increasing the fanout of internal nodes and, hence, decreasing the tree depth. Previously, we used a proactive split policy by splitting nodes with at least $2t$ entries before inserting or flushing to the node. Keeping the internal nodes' fanout at a reasonable level of $\Theta(B^{\varepsilon})$ allowed us to guarantee a minimum flush size of $\Theta(B^{1-\varepsilon})$. This is the key to the good insertion performance of the B$^{\varepsilon}$-Tree.

But this minimum flush size can be guaranteed without splitting internal nodes at a fixed upper limit before flushing. Instead of that, we handle too large internal nodes as follows: We examine the largest child buffer of the internal node. If its size is at least the configurable minimum flush size, we will flush this buffer. Otherwise, we will split this node. So regarding configuration, we replaced the upper bound for internal node size by the minimum flush size.

When we use this concept, the tree will dynamically shape according to the write pattern and, hence, it will automatically optimize itself for the workload. If we use sequential insertion, the corresponding tree part will be very wide as the internal nodes in this tree part can maintain the required minimum flush size even when their fanout is very high. Simultaneously, the tree will maintain a low fanout on random insertions so that the tree still offers a good random insertion performance. This optimization also benefits random point query performance after a sequential insertion because of the high fanout and low tree depth.

Concurrency Control

The Tree Layer can be used concurrently. For better concurrency, each *ObjectReference* and each tree node is protected by a read-write lock. Most of the time, only read locks are needed for the *ObjectReference* as the *ObjectReference* will be in the correct state and the tree node will be cached. Only `insert` and `range_delete` need to write lock tree nodes as they are the only two operations that modify tree nodes. Therefore, this simple concept enables highly concurrent access to this layer.

When we walk down a tree, we use lock coupling [BS77]. Lock coupling means that we lock the child node before unlocking the parent node. This guarantees that the tree

path is valid and cannot be modified by others in between. Additionally, lock coupling is deadlock free.

Configurable Parameters

The tree parameters are configurable. The following parameters are available:

`MIN_LEAF_NODE_SIZE` The minimum leaf node size in bytes. Below this threshold, a leaf node will be merged or rebalanced. Defaults to 1 MiB.

`MAX_LEAF_NODE_SIZE` The maximum leaf node size in bytes. Above this threshold, a leaf node will be split. This should be at least four times larger than `MIN_LEAF_NODE_SIZE`. Defaults to 4 MiB.

`MAX_INTERNAL_NODE_SIZE` The maximum internal node size in bytes. Above this threshold, we flush or split an internal node. Defaults to 4 MiB.

`MIN_FLUSH_SIZE` The minimum flush size in bytes. Below this threshold, we will not flush the internal node but split it. Defaults to 256 KiB.

`MIN_FANOUT` The minimum fanout. Below this threshold, an internal node will be merged. Defaults to 4 children.

The fanout threshold at which an internal node may be split, can be calculated by dividing the maximum internal node size by the minimum flush size. Additionally, the following invariant must hold:

$$2 \cdot \texttt{MIN\_FANOUT} \leq \left\lceil \frac{\texttt{MAX\_INTERNAL\_NODE\_SIZE}}{\texttt{MIN\_FLUSH\_SIZE}} \right\rceil$$

This equation ensures that if an overflowed internal node cannot be flushed, the resulting nodes of the split have at least the minimum fanout. If this equation is not satisfied, there can be overflowed internal nodes that can be neither flushed nor split.

3.6.2. Implementation

`get(Tree, Key) -> Option<Value>`

1. Get the root node.
2. If the current node is an internal node:
   a) Select the child that corresponds to our search key.
   b) If this buffer contains a message for the key, append it to our message list.
   c) Fetch the child node and go back to step 2.
3. Otherwise, it is a leaf node.

4. Retrieve the value for the key if present.
5. Apply all messages in our messages list in reverse order to the value.
6. Return the final value.

Notes This algorithm walks down the tree along the path that corresponds to the key. Every message for this key must be on this path. Hence, we can collect all relevant messages. The messages have a chronological order: Messages that are located in a subtree of a node must be older than messages in the respective node. Therefore, we can apply the messages that we have seen on this path in reverse order so that we recover the current state of the value for this key.

`insert(Tree, Key, Message)`

1. Get the root node in mutable state.
2. While the current node is an internal node, does not contain a message for the key, and the corresponding child node for the key is in mutable state, set the child node as current node.
3. Insert the message into the current node. Merge or apply the message if needed.
4. If the current node is an internal root node and has only one child, flush all messages to the child node and set the child node as root node and current node.
5. While the current node is too large:
   a) If the node is a leaf, split the node, select the larger one as current node, and continue the loop.
   b) Otherwise, the node must be an internal node.
   c) Select the largest child buffer of this node.
   d) If this buffer is smaller than the minimum flush size, split the current node, select the larger one as current node, and continue the loop.
   e) Get the child node in mutable state.
   f) If the child node is an internal node with too low fanout, merge it with a sibling, and continue the loop.
   g) Flush all messages in the child buffer to the child node. Merge or apply the messages if needed.
   h) If the child node is a leaf and too small, merge or rebalance with sibling.
   i) If the child node is a leaf: While the child node is too large, split the child node and select the larger node as child.
   j) Set the child node as current node.

Notes The first two steps in this algorithm are the "Tree Walk before Insertion" optimization described beforehand. Then the message is inserted into a node on the path corresponding to the key. We retain the invariant that the messages are chronological ordered because we do not skip a node which already contains a message with the same key. The main complexity of this method belongs to the repairing of the tree state. Basically, we walk the tree from the current node downwards by pushing the messages of too large nodes to child nodes. If we cannot maintain the minimum flush size on flushing from an internal node, we split this node. This is the "Node Split Depends on Minimum Flush Size" optimization. Leaf nodes are split, merged and rebalanced as needed.

`range(Tree, KeyRange) -> Iterator`

The iterator returned by this operation is defined as follows:

Type

```
struct RangeInterator {
  Buffer,
  low_key: Key,
  high_key: Key,
  done: bool,
}
```

`Buffer` is a buffered sequence of key-value pairs. Initially, the buffer is empty, `low_key` and `high_key` are set according to the given key range, and `done` is false. The iterator pops key-value pairs from the head of the buffer. If the buffer is empty and `done` is false, we fill the buffer with the following algorithm:

1. Walk the tree along the path corresponding to the `low_key`.
2. Collect all messages along that path.
3. Save the key range of this path if present. The range is defined by the last seen left and right pivot keys for the nodes on the path.
4. Discard all messages outside of this key range.
5. Merge the leaf entries with the collected messages ordered by key into the key-value buffer by applying the messages in reverse order.
6. Discard all buffer entries outside of the range of `low_key` and `high_key`.
7. If we have seen a right pivot key, set `low_key` to the successor of this key.
8. Otherwise, set `done` to true.
9. If `low_key` is higher than `high_key`, set `done` to true.

Notes The iterator maintains an unvisited key range. On buffer fill, we collect the data that belongs to the leftmost root-to-leaf path in the key range and then update the key range accordingly. As this path may contain messages or data which is outside of the range, we have to trim the buffer. If the last right pivot key on this path is higher or equal to high key of the key range, we are finished as the tree does not contain anymore data in the key range.

Note that in this case it is not sufficient to check that the buffer is empty after executing the buffer fill algorithm because the buffer can be empty due to the fact that the selected root-to-leaf path may contain no live data because of deletion messages.

`range_delete(Tree, KeyRange)`

- Call `range_delete_inner` with the root node in mutable state.

`range_delete_inner(MutableNode, KeyRange)`

1. If this node is a leaf, remove all entries in the key range, and return.
2. Otherwise, this node is an internal node.
3. Remove all messages in the given key range from the buffer.
4. Select all child nodes that belong to the key range.
5. Call `range_delete_inner` for the leftmost and rightmost child node.
6. Remove the subtrees that belong to the child nodes between the leftmost and rightmost child node.

Notes This algorithm walks the tree and visits all subtrees that may contain data in the key range. If an internal node has at least three children that are in the key range, the data of the inner children is contained in the key range and, hence, the child nodes can be removed without any further checks.

`sync(Tree) -> TreePointer`

1. Call the DML method `write_back` with root node *ObjectReference*.

Notes The write-back algorithm ensures that all child nodes of the root node will be written back to disk before the write-back of the root node.

3.7. Database Layer

The Database Layer manages multiple data sets and snapshots. Data sets and snapshots provide a key-value store interface, i.e. they are maps from byte sequences to byte sequences. Each data set uses a B^ε-Tree to save its data. A snapshot of a data set is just a snapshot of the corresponding B^ε-Tree. Additionally, there is a root B^ε-Tree that holds the allocation bitmaps and the metadata. This layer also saves a *Superblock* on the underlying storage pool which contains the *TreePointer* of the root B^ε-Tree so that this database is persistent.

This layer has the following interface, again excluding error handling for brevity:

Database Interface

```
open(Configuration) -> Database
create(Configuration) -> Database
open_dataset(&mut Database, Name) -> Dataset
create_dataset(&mut Database, Name) -> Dataset
snapshot_dataset(&mut Database, &mut Dataset, Name)
delete_dataset(&mut Database, Dataset)
sync(&mut Database)
```

Dataset Interface

```
get(&Dataset, Key) -> Option<Data>
insert(&Dataset, Key, Data)
delete(&Dataset, Key)
range(&Dataset, KeyRange) -> Iterator
range_delete(&Dataset, KeyRange)
open_snapshot(&mut Dataset, Name) -> Snapshot
delete_snapshot(&mut Dataset, Snapshot)
```

Snapshot Interface

```
get(&Snapshot, Key) -> Option<Data>
range(&Snapshot, KeyRange) -> Iterator
```

A database can be opened or created. Both calls accept a `Configuration` parameter which specifies the configuration of the *Storage Pool Layer*, i.e. which block devices are grouped into vdevs of what type. Then the user can use the `Database` object to manage data sets which can be opened, created, deleted, and snapshotted. Each data set is identified by a unique name which is a byte sequence. Each snapshot is identified by the corresponding data set and name. With the `Dataset` and `Snapshot` objects, we can actually retrieve data by calling `get` and `range` which are directly mapped to the corresponding Tree Layer operations. Snapshots are read-only, but data sets provide

`insert`, `delete`, and `range_delete`. This layer has the invariant that each existing `Dataset` and `Snapshot` object points to different data sets and snapshots.

Before outlining the implementation of this layer, we will focus on some important details. Internally, a data set is identified by a `DatasetId` which is an integer. A snapshot belongs to a data set and the `Generation` in which it was created. The `Generation` is an integer which will be bumped on each `sync` of the Database. This layer uses the `Generation` field of the DML *ObjectPointer* to track the age of the tree nodes. Hence, we know the "birth" generation of each tree node. As snapshots are identified by their creation generation, we can check quickly whether a tree node cannot be used by a snapshot.

This layer also uses the `TreeId` of the Tree Layer for the tree nodes which is just the `DatasetId` of the data set to which the tree node belongs. Therefore, the *Allocation Handler* of this layer knows to which data set a DML object belongs.

3.7.1. Superblock

This layer reserves two blocks at the beginning of each vdev to have two slots to save the Superblock. The Superblock contains the `TreePointer` of the root B^ε-Tree and is self-checksummed. At the end of `sync`, one of the two slots will be updated with the current Superblock on every vdev depending on whether the current `Generation` is odd or even. On `open`, all slots are read and the intact Superblock with the highest `Generation` will be used.

The outlined `open` implementation is correct because this layer guarantees that if there is a Superblock with a correct checksum on disk, the corresponding trees have been committed to disk successfully. (For details, see the implementation of `sync(&mut Database)`.)

3.7.2. Root B^ε-Tree

As already mentioned, the root B^ε-Tree contains the allocation bitmaps and metadata for data sets and snapshots. In the following we show how the data is laid out as key-value pairs in the B^ε-Tree. As the keys are actually byte sequences, we have to use an encoding for key types which are not already byte sequences. For integers, we simply use big endian encoding as the lexicographic order of the bytes of a big endian encoded integer matches the order of the integer. Therefore, the integer order will be preserved in the tree. We use a single byte (denoted by `XX`) as prefix to avoid key collisions.

The root B^ε-Tree contains the following `key -> value` maps:

Last used DatasetId `00 -> <DatasetId>`
: As the `DatasetId` have to be unique, we have to save the last used `DatasetId`.

Allocation Bitmaps `00 <SegmentId> -> <Bitmap>`
: For each allocation segment, we save the corresponding bitmap identified by the `SegmentId`. The bitmap has a zero byte for each unallocated block and a 1 byte

for each allocated block. We have to use one byte per bit as the upsert message has only byte granularity.

Data set name lookup table `01 <Data set name> -> <DatasetId>`
With this table we can lookup the corresponding `DatasetId` for a given data set name. Having a separate short identifier for data sets results in shorter keys for the following key-value pairs.

Data set metadata `02 <DatasetId> -> <DatasetData>`
We save the following struct which contains all metadata about a data set:

Type
```
struct DatasetData {
  previous_snapshot: Option<SnapshotId>,
  data_set_tree: TreePointer
}
```

Snapshot name lookup table `03 <DatasetId> <Snapshot name> -> <SnapshotId>`
For looking up the `SnapshotId` for a given snapshot name.

Snapshot metadata `04 <DatasetId> <SnapshotId> -> <SnapshotData>`
For each snapshot, we save the following metadata:

Type
```
struct SnapshotData {
  previous_snapshot: Option<SnapshotId>,
  snapshot_tree: TreePointer
}
```

Dead list `05 <DatasetId> <Generation> <DiskOffset> -> <DeadListData>`
The dead list of each data set tracks DML objects which are dead in the data set but still in use by snapshots. Each dead list entry corresponds to the data blocks of a DML object. The key contains the `Generation` in which the DML object died and its `DiskOffset`. Due to the key's layout, the dead list is sorted by the generation. The value contains the following data:

Type
```
struct DeadListData {
  size: Block,
  birth: Generation
}
```

The dead lists will be filled by the *Allocation Handler* whenever a DML object is not in use anymore by a data set, but it is still in use by at least one snapshot. On snapshot deletion, we walk the dead list and actually deallocate the blocks of DML objects which are solely used by the deleted snapshot.

3.7.3. Allocation Handler

We provide an *Allocation Handler* as the allocation bitmaps are persisted in the root B^ε-Tree of this layer. The implementation exploits a side effect of copy-on-write storage stacks: We have an essentially free snapshot of the last synced state. This aspect will be used in `get_allocation_bitmap` as it retrieves the previous allocation bitmap.

`get_allocation_bitmap(SegmentId) -> Bitmap`

1. Retrieve the allocation bitmap for this segment from the current tree and the previously synced tree and bitwise-OR these two bitmaps.

2. In the resulting bitmap, mark the on-disk blocks of the root node of the old root tree as used if appropriate.

Notes By definition, this function shall return a bitmap where active blocks are marked as used. Active blocks are blocks which contain data that we may not override. Hence, blocks are active when they are currently allocated or have been deallocated in this generation, i.e. an active block is either currently allocated or was allocated in the previous generation. Therefore, we can simply bitwise-OR the allocation bitmap from the current root B^ε-Tree and the bitmap from the previously synced root B^ε-Tree to build such a bitmap.

As the allocation bitmap of the old root tree misses the allocation of its own root node, we mark its blocks as used. (See the `sync(&mut Database)` implementation for more details on this inconsistency)

`allocate_at(DiskOffset, Size)`

1. Insert an upsert message into the root B^ε-Tree that updates the allocation bitmap of the segment corresponding to the given `DiskOffset`.

`copy_on_write(&ObjectPointer)`

1. Check whether the data set of this object has a latest snapshot and that this snapshot is older than the object, i.e. the snapshot's generation is equal or greater than the object's birth generation.

2. If there exists such a snapshot, we insert the on-disk blocks of this object to the dead list of the data set. The `Generation` in the key for the dead list is the current `Generation`.

3. Otherwise, we insert an upsert message into the root B^ε-Tree that deallocates the blocks by updating the corresponding allocation bitmap.

Notes Every time a tree node is copied for modification, we have to check whether we need its block on disk, i.e. the corresponding data set has at least one snapshot which uses this block. If the block was born before the latest snapshot was created, the block must be in use by this snapshot because otherwise, it would have been deallocated before the snapshot creation.

3.7.4. Implementation

`open(Configuration) -> Database`

1. Find the newest intact Superblock as described in Section 3.7.1.
2. Open the root B^ε-Tree given by this Superblock.

Notes As outlined, it is guaranteed that if we see a Superblock with a correct checksum, the corresponding state has been committed to disk successfully. Hence, the newest intact Superblock corresponds to a consistent on-disk state.

`create(Configuration) -> Database`

1. Overwrite all Superblock slots with zeros.
2. Create a new root tree for this database.
3. Allocate two blocks at the beginning of each vdev.
4. Call `sync(&mut Database)`.

Notes We have to erase old Superblocks that are potentially existing on disk so that they cannot be selected accidentally in an `open` call later on.

`sync(&mut Database)`

1. Write back all open data set trees.
2. For each modified data set, update the `TreePointer` in the metadata.
3. Write back the root tree.
4. If this write-back included any nodes other than the root node, go back to step 3.
5. Ensure that all writes have been committed to disk by flushing the Storage Pool Layer.
6. Update the corresponding Superblock.
7. Flush the Storage Pool Layer again.
8. Bump the generation.

Notes When we write back a tree, we have to allocate for every modified node. Hence, the allocation data in the root tree will be updated. This results in a feedback loop if we write back the root tree. Therefore, after finishing the write-back of the root tree, the on-disk state is missing some allocations that occurred during the write-back. At least the root node allocation is missing as the allocation occurs after the root node has been packed and (optionally) compressed so that we know its exact on-disk size. But we can limit the number of missing allocations: The repeated write-back of the root tree in step 3 and 4 guarantees that the allocation bitmaps are up-to-date except for the allocation of the root node itself. To fix this inconsistency, the *Allocation Handler* is responsible for marking the on-disk part of the root node of the previously synced root tree as allocated.

`snapshot_dataset(&mut Database, &mut Dataset, Name)`

1. Check that there is no snapshot with the same name for this data set.
2. Write back the data set tree.
3. Use the current generation as the `SnapshotId`.
4. Save the snapshot metadata.
5. Insert the name into the snapshot lookup table.
6. Update the data set metadata to reflect the new snapshot.
7. Call `sync(&mut Database)`.

Notes Snapshot creation is very lightweight and only needs a `sync` operation.

`delete_snapshot(&mut Dataset, Snapshot)`

1. If this is the newest snapshot, update the data set metadata and set the data set's previous snapshot to the previous snapshot of this snapshot.
2. Remove the snapshot name entry in the lookup table.
3. Remove the snapshot metadata entry.
4. Walk the dead list of this data set from the birth generation of this snapshot plus one to the birth generation of the next snapshot.
5. For each dead list entry, check whether the block was born after the previous snapshot has been created.
6. If yes, delete this list entry and deallocate this block.

Notes On snapshot deletion, we have to deallocate all dead blocks that are in use by this snapshot but not by any other snapshots. Clearly, these blocks were born before this snapshot and died after this snapshot. The following two additional conditions guarantee that a block is not in use by any other snapshot:

Born after the previous snapshot Otherwise, this block must be still in use by the previous snapshot.

Died before the next snapshot Otherwise, this block must be still in use by the next snapshot.

Hence, we filter the dead list for blocks that match these conditions and deallocate these blocks.

`delete_dataset(&mut Database, Dataset)`

1. Walk the data set tree and deallocate every node.
2. Deallocate every block in the dead list of this data set.
3. Remove the dead list, the snapshot metadata, and the snapshot name lookup table.
4. Remove the data set metadata and the name entry in the data set lookup table.

Notes Every data block referenced by a data set belongs to a tree node either in the current data set tree or the tree node is referenced by a dead list entry. Hence, this algorithm deallocates exactly every block referenced by this data set in step 1 and 2.

4. Comparison Against Existing Storage Stacks

In this chapter, we introduce and analyze the Linux storage stack as a mature solution and ZFS as a modern approach. Both storage stacks will be compared against the storage stack design presented in this thesis. In this comparison, we especially focus on data integrity, usability, the implementation of snapshots, and overall efficiency of the storage stacks.

4.1. Linux Storage Stack

We will start off with the typical Linux storage stack which mainly consists of the logical volume manager *LVM* on top of a RAID. At first, we will give an overview of this stack. Before we compare the design of this stack against our design, we describe two components of the Linux storage stack in detail as their implementation vastly differs compared to our storage stack: the RAID functionality and the implementation of snapshots.

4.1.1. Conceptional Overview

Traditionally, the storage stack on Linux consists of POSIX filesystems at the top, a logical volume manager in between, and either hardware RAID controllers or a software RAID at the bottom of the stack (see Figure 4.1).

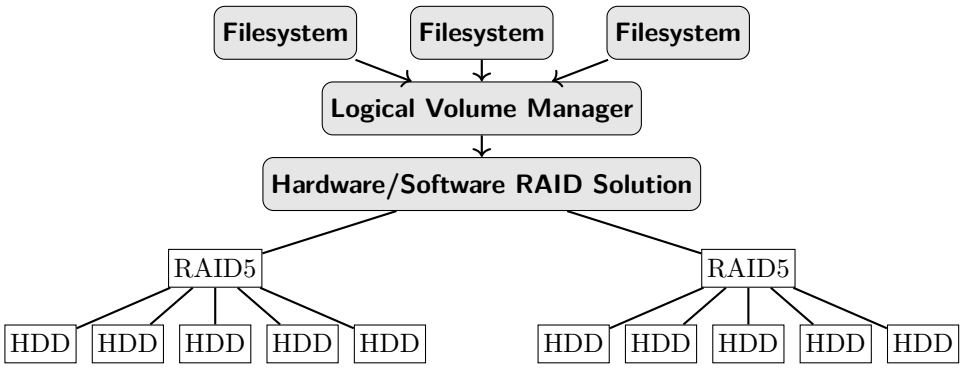

Figure 4.1.: Linux storage stack

The bottom layer manages many hard disks which are combined into logical units called RAIDs. Depending on its *RAID level*, a RAID provides either better performance, reliability, or both compared to a single disk. This layer exports a block device interface to access the data on the underlying RAIDs – i.e. a linear space of data which can be read and written in blocks (mostly 512 bytes) with a sync operation.

On top of the RAID sits a *logical volume manager* such as LVM. This layer has one or more underlying block devices called *physical volumes* which are combined into a *volume group*. The *volume group* provides *logical volumes* which again have a block device interface. Hence, this layer subdivides the lower block device(s) into multiple smaller block devices. In contrast to static partitioning, the LVM allows for dynamic creation, growing, and shrinking of the *logical volumes* and, hence, is much more flexible. Additionally, LVM can snapshot *logical volumes* and supports clones, too.

Finally, on top of each LVM *logical volume* we have POSIX filesystems in the storage stack such as *ext4* or *XFS*.

4.1.2. RAID Levels

There are four commonly used RAID levels which define the behavior of a RAID group:

RAID0 The data is striped in fixed chunk sizes across all disks in this unit. If one disk fails, the data is lost.

RAID1 The data is mirrored in fixed chunk sizes across all disks in this unit. If all disks fail, the data is lost.

RAID5 The data is striped in fixed chunk sizes with an additional parity data chunk per stripe across all disks in this unit. The location of the parity data depends on the data offset so that the parity data is evenly distributed among the disks. If a disk fails, we can read the parity data chunk to recompute the missing data in the stripe. If two disks fail, the data is lost.

RAID6 Similar to RAID5, but with two parity data chunks per stripe. Hence, if three disks fail, the data is lost.

RAIDx0 RAIDx0 such as RAID10, RAID50, and RAID60 is not a RAID level per se, but stands for striping data among multiple RAIDs of the same kind – i.e. it is a RAID0 on top of other RAIDs. If an underlying RAID fails, the data is lost.

Example: The RAID5 groups shown in Figure 4.1 use the space of one disk in each stripe for parity blocks and, thus, are resilient against one disk outage per group. The data of the LVM will be striped across both groups which is also known as RAID50.

4.1.3. LVM Snapshots

With LVM, we can create copy-on-write snapshots of individual *logical volumes*. The snapshot data will live in a separate *logical volume* whose space has to be preallocated

but it can be dynamically resized. For each snapshot, LVM keeps track of modifications of the original logical volume since the snapshot creation in an *exception table*. Before a block will be overwritten by a write in the logical volume, LVM copies the previous data block to the snapshot's logical volume and inserts an entry into the exception table. If the snapshot has not enough space to save the data block, the snapshot will be disabled and becomes unusable.

Note that since Linux 3.2 there is also support for thin provisioning pools in LVM. These pools handle snapshots of thinly provisioned logical volumes quite differently than the described approach [TOO16; Dev17b].

4.1.4. Design Comparison

The concepts of both the Linux storage stack and our storage stack have a layered approach where multiple components are stacked on top of each other. In the Linux storage stack, every layer besides the top most layer provides a block device interface. In contrast, each layer of our storage stack exposes a different interface. First, a block device interface with additional checksums at the Vdev and Storage Pool Layer, then an object-based interface at the Data Management Layer, and finally, a key-value store interface at the Tree and Database Layer. Hence, these layers are more tightly integrated and more dependent on each other while the layers of the Linux storage stack have a simpler API surface. These design decisions result in many important differences between the two storage stacks which we will now discuss in greater detail.

Customizability

The Linux storage stack is customizable for different needs because each lower layer can be easily removed or replaced. This is possible because of the identical layer interfaces. Regarding our storage stack, each layer is dependent on its bottom layer by design. This flexibility of the Linux storage stack might come in handy if we only need a portion of the functionality, for example snapshots. In that case, we could only use some layers and, hence, reduce the complexity of the whole setup.

For example in the evaluation (see Chapter 7), we do not use snapshots or multiple data sets and therefore, we do not need the LVM layer of the Linux storage stack. By removing this layer, the benchmark setup becomes easier.

Convenience and Robustness

As already mentioned, the Linux storage stack consists of independent layers. Therefore, each layer is managed by a different set of tools. The hardware RAID-controller can be managed by a vendor supplied command line tool, the Linux software RAID is administrated via `mdadm`, LVM can be configured by using the `lvm` program, and each filesystem has specific mount options and a filesystem check program. This is inconvenient for the administrator who has to remember the different programs and command line options. Additionally, the administrator has to use multiple programs for a single task.

For example, creating a new filesystem on top of LVM consists of multiple steps: Creating a new logical volume, creating a new filesystem on top of this logical volume, adding an entry to the `fstab` file, and finally mount the filesystem. Each step involves a separate program. Managing snapshots has a similar complexity. In contrast, our storage stack can create a new data set or a new snapshot using a single API call. Thus, it is much easier to use.

Another aspect is the robustness of the snapshot implementation. As outlined in Section 4.1.3, the LVM snapshot implementation is susceptible to outages. If the administrator did not dedicate enough space for a snapshot, the snapshot may become unusable if the original logical volume receives too many writes. This is highly inconvenient and error-prone. Our design has no such issues with respect to snapshots.

Also the robustness against failures is a major difference between these two storage stacks. Our storage stack guarantees that the state on disk is always consistent due to its copy-on-write principle. Hence, it does not need any check after a crash or power outage. Contrary to this, every layer in the Linux storage stack might have an inconsistent state on disk and for each layer different actions need to be taken. Each filesystem may be checked with its specific `fsck` program, LVM provides `vgcheck` and `pvcheck`, and the RAID needs a *resync* to synchronize the data on the disks – this process is also called *scrubbing*.

Metadata Overhead

Regarding the Linux storage stack, the simplicity of the interfaces between each layer implies some downsides. For example, the LVM and each filesystem have to manage allocations. The LVM allocates multiple blocks of the underlying block device to upper block devices and each filesystem allocates block of the underlying block device to the file data of an inode. This results in doubled costs and complexity for allocations and is due to the block device interfaces.

In our design, the upper layers are agnostic of the actual location of data contrary to the block device interface. For example, it is transparent for the Tree Layer where the tree nodes are located on disk. This allows the Data Management Layer to decide where to write the data as it can be written anywhere. Hence, only one layer is actually in charge of allocating disk space to data blocks. This results in less overhead for allocations in terms of computation, disk space, and management.

Snapshot Efficiency

The LVM snapshot implementation is rather simple and limited by the layer interfaces. As already brought up, each LVM snapshot needs a separate preallocated logical volume to which the data is copied on every write to the original logical volume. In turn, this means that the write throughput on the logical volume depends on whether the logical volume has snapshots as each write to the logical volume is preceded by a copy for each

existing snapshot.[1] Even if we only have one snapshot, the write throughput is effectively limited to at most *one third* in theory. Practically, this limitation is much more severe with a *10-fold* to a *100-fold* slowdown [Dok14].

Besides that, the LVM may even copy data which is not in use by a snapshot because this layer may not know whether a data block was actually in use by the filesystem at that point in time.

All in all, it is obvious that LVM snapshots are designed for short term snapshots, for example for taking a consistent backup of a logical volume.

As our storage stack is designed around copy-on-write, it has native support for snapshots. Because each overwrite will be redirected to unused space, having a snapshot for data set simply means that we do not deallocate (logical) overwritten data. Hence, our design does not have any of the issues listed above. Instead, the major issue is freeing unused data blocks on snapshot deletion. This is solved rather efficiently with the dead list as explained in the Section 3.7.4. Still, deleting snapshots is more complex and expensive compared to LVM, but this trade-off is definitely worth the effort.

Data Integrity

A severe disadvantage of the Linux storage stack is the missing data integrity in the whole concept. The RAID layer can only protect against failing disks that report an error for the operation. If a disk signals a successful read but returned wrong data, the upper layer may detect this with checksums but there is no way to repair the data despite the RAID may having a mirror or parity chunks because the RAID cannot verify the integrity of the data read.

Additionally, system failures are critical for RAIDs, too. The RAID-controller and each hard disk have a write cache for better performance which may not be flushed to disk on power failure. On the next read of these blocks, we have a mismatch between mirrors or the data chunks and the parity chunks like in the previous described case. Likewise, as we cannot verify the integrity of the blocks, we cannot repair this case. Even without write caches, this may occur as we cannot atomically write to multiple disks. For RAID5, this problem is also known as the "RAID5 write hole" which is the time frame in which the parity chunks do not match the data chunks. As mentioned, a RAID can be *resynced* after a failure, but this operation may corrupt intact data as the layer cannot know which chunk is corrupt. This is especially a problem when it comes to *silent data corruption*. Recent studies suggest that "RAID reconstruction discovers about 8% of the checksum mismatches in nearline disks" [Bai+08].

On contrary, our storage stack includes data integrity by design. Every data block is protected by a checksum. Except for the Superblock, this checksum is stored in the upper block that references the block. Thus, the state of our storage stack builds a *Merkle tree* [Mer87] so that we can verify the correctness of every referenced data block by solely having an intact Superblock and walking down the tree – this operation is

[1] An approach to avoid this linear slowdown of the write throughput is to chain the exception tables of snapshots so that each block will be copied only once. However, this involves a linear slowdown of the read throughput of snapshots and the snapshots will depend on their ancestors.

called *scrubbing*. Compared to saving the checksum inside the data, this approach also detects *identity discrepancies* which can be caused by lost or misdirected writes (see [Bai+08, Section 2.3] for more details).

To sum up, our storage stack fully protects the data's integrity, but the Linux storage stack cannot provide data integrity by design.

4.2. ZFS

After analyzing the Linux storage stack, we will now look at ZFS which is a much more modern approach to designing storage stacks. ZFS provides POSIX compatible filesystems and block devices on top of a storage pool and solves most of the problems mentioned in the previous section. Again, we give an design overview first, and then focus on details regarding snapshots and allocation. Afterwards, we compare ZFS to our design.

4.2.1. Conceptional Overview

Once more, ZFS [Bon+03] uses a layered approach (see Figure 4.2). The innovation of ZFS are the reinvented layer interfaces. Compared to the Linux storage stack, the layer interfaces are aligned to the actual needs of each layer. Together with copy-on-write, this is the key to a powerful, feature rich, and efficient storage stack.

The bottom layer, the *Storage Pool Allocator* (SPA), replaces the RAID layer but the SPA has a much richer interface than the usual block device interface of a RAID layer. The SPA combines block devices into logical units called *virtual devices* – short *vdevs*. The data of the storage pool will be spread over the virtual devices. Each virtual device can either be a single block device, a *mirror* of two or more devices, or a *raidz1*, *raidz2*, or *raidz3* which build a group with single, double, or triple parity respectively. The functionality and the data layout of *raidz* is very similar to the *parity1* vdev of our design as every data block is a whole stripe that will be distributed across the disks of a raidz group. There are also other special purpose vdevs – for more information, see `man 8 zpool`.

As the name of the SPA suggests, this layer handles allocations, but also variable block sizes, compression, and checksumming. Each data block read and written by the SPA is identified by a *block pointer* which contains the offset on disk, the compressed size, the checksum, and some other metadata. With the checksum in the block pointer, the SPA can verify the integrity of the data blocks on disk and, hence, does not suffer from the described downsides of RAIDs. In a disk group with parity, it even has the possibility to recover from a read which returned wrong data by applying a combinatorial reconstruction of the data and parity blocks (for details of this functionality, see Section 3.3.3).

Using the rich interface of the SPA, the *Data Management Unit* (DMU) implements an even richer interface. The DMU is a transactional object store with support for atomic operations, multiple object types, and multiple data sets which support snapshots and clones. Finally, on top of the DMU, the *ZFS POSIX Layer* implements a POSIX-

compatible filesystem. Due to the powerful transaction mechanisms of the DMU, this layer is rather thin.

All in all, ZFS is as flexible as the Linux storage stack, but has less overhead and additionally provides data integrity, efficient snapshots, and much more features which are beyond the scope of this short introduction to ZFS. We will now focus on how ZFS handles snapshots and allocations.

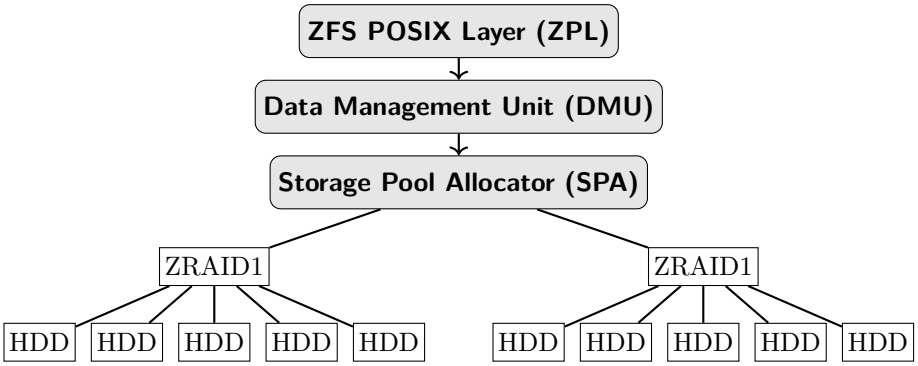

Figure 4.2.: Simplified ZFS storage stack

4.2.2. Snapshots

Similar to our design, the internal state of ZFS consists of many tree structures which are referencing each other. The tree nodes are identified by block pointers. With the checksum in the block pointer, the trees again form a Merkle tree [Mer87] with a single self-checksummed block at the top: the *Uberblock*. Also, ZFS has a notion of generations. Every time the on-disk state is updated, ZFS commits a *transaction group* which is identified by a logical timestamp.

Using these building blocks, the implementation of snapshots is straightforward. Again, ZFS applies the path-copying technique [Dri+86] to all trees. Hence, for supporting snapshots, ZFS just has to not deallocate data blocks if a data set has a snapshot. Instead, these blocks will be saved in a data structure called *dead list* so that ZFS can easily enumerate the data blocks that have to be deallocated if a snapshot is deleted.

4.2.3. Allocation

Because ZFS is a redirect-on-write filesystem, it does not overwrite active data and needs to allocate on every write. Likewise, on logical overwrite, ZFS has to potentially free the old data blocks. Therefore, the allocation subsystem is much more stressed compared to other filesystems that overwrite data and, thus, allocating needs to be very efficient for ZFS. To achieve this goal, ZFS includes some tricks and techniques which we will present in the following.

First of all, each vdev of a ZFS storage pool is split into about 200 independent segments called *metaslabs*. Each metaslab has an associated *space map* which tracks

the free space in this metaslab on sector granularity.[2] On every allocation request, the SPA allocates space in a metaslab and the space map will be updated on the next commit. Likewise on every deallocation, the SPA needs to update the corresponding space map. As the SPA can decide where to allocate blocks, it tries to allocate from the same metaslab so that less space maps have to be updated. Additionally, the space maps have an optimized representation on disk. Instead of a simple bitmap, a space map is a logical log of operations and ZFS appends a log entry for each allocation and deallocation. This turns allocating into an efficient sequential workload. If a space map becomes too large with many overlapping allocations and deallocations in the log, it will be *condensed* and the whole log will be rewritten which is also very efficient.

However, deallocations still emit a random workload. On an aged storage pool, it is highly likely that every space map needs to be updated on a sync due to many deallocations scattered across the pool. Therefore, ZFS needs to issue about 200 small random I/Os per vdev and sync for updating the space maps.

4.2.4. Design Comparison

As indicated in the introductions, the overall design of ZFS and our storage stack is very similar:

1. Both storage stacks use copy-on-write which enables consistent on-disk state and efficient snapshots.

2. Both have a special pointer type for referencing disk blocks – the ObjectPointer and the block pointer. These pointers includes a checksum of the referenced data so that the data integrity can be verified.

3. Both use trees as primary data structure so that the whole storage pool forms a Merkle tree. Hence, the whole storage pool's consistency and integrity can be easily verified.

4. The lower layer of both storage stacks avoids read-modify-write cycles for RAID vdevs by having a dynamic stripe width so that every write is a full stripe write.

The main difference between both implementations is that ZFS is much more feature rich. However, most of ZFS's features could be easily supported by our storage stack because of the similarities between both designs. Nonetheless, there are some notable differences between both designs as we will see in the following.

Allocation

Allocation metadata is handled very differently in both designs. On the one hand, ZFS uses specialized data structures for saving the free space information on disk: the log-structured space maps. On the other hand, our storage stack simply saves bitmaps in a B^{ε}-Tree and inserts upsert messages to update the bitmaps on allocation or deallocation.

[2] The granularity can be controlled by the `ashift` parameter on pool creation.

Despite the sophisticated concept of ZFS, synchronizing the space maps can still be a burden. On the OpenZFS Developer Summit 2017, Dimitropoulos presented a tiny benchmark that simulates a write intensive workload. The analysis of the I/O distribution during this benchmark shows that a non-negligible portion (25%) of all I/Os are due to space map updates [Dim17]. The proposed solution is to flush only one instead of all space maps per synchronization and add another log in front of the space maps (which are logs itself). Then, both allocations and deallocations emit a sequential workload. Altogether, the ZFS concept to handling allocations efficiently becomes quite complex.

By using a B^ε-Tree, our design is much simpler and still efficient in practice. As analyzed previously, B^ε-Trees ingest random workloads very fast. Furthermore, the entries of a B^ε-Tree stay mostly logically ordered. The fragmentation is limited by the leaf node size and the tree height, as for each key there is exactly one root-to-leaf path in a B^ε-Tree on which messages can reside. Hence, the B^ε-Tree is very suited for allocation workloads as the bitmaps are read in sequential order and written in sequential and random order.

Fragmentation

In ZFS, all user data blocks are written as separate units to disk and are identified by a block pointer. As ZFS does not support rewriting a block pointer, ZFS cannot move data on disk. Therefore, the position of the data is fixed upon the first write. This leads to fragmentation of user data. On a large write to a file, the data blocks will be allocated and laid out on disk in the order the write were issued. Hence, the file data block will be in sequential order when we have a sequential workload. But if we have a random workload, the file data will become heavily fragmented. Thus, a following sequential read of the file will emit a random I/O workload. Likewise, file data becomes fragmented when portions of the file are overwritten during a longer period of time as ZFS is a copy-on-write filesystem.

Our design handles the storage of user data differently: the data is inlined into a B^ε-Tree. Therefore, the fragmentation is again limited – as is the case for the allocation bitmaps. Hence, sequential reads have only a fairly limited slowdown due to fragmentation regardless of the write access pattern. However, random writes are probably slower compared to ZFS as the user data has to be moved downwards the tree and is copied multiple times during this process. All in all, our approach is a trade-off of write speed for read speed. As noted in the comparison of B-Trees and B^ε-Trees, this trade-off should be beneficial for most workloads as the read speedup is larger than the write slowdown.

Snapshot Deletion

Both designs keep track of deleted blocks in dead lists. Our design has one dead list per data set which is ordered by death time. ZFS, however, keeps multiple dead lists per snapshot. On snapshot deletion, we have to walk a large part of the dead list and delete some of the entries. So, we sequentially read a part of the dead list and randomly

insert deletion messages into the dead list which is reasonably efficient with a B^{ε}-Tree. In contrast, ZFS just deletes all entries on one dead list of the snapshot and appends all other dead lists of this snapshot to dead lists of the next snapshot. The approach of ZFS should be much more efficient and significantly speeds up the deletion of large snapshots.

Reusability

ZFS is a large, full-blown filesystem in the kernel space. Therefore, the reusability of ZFS and its code is quite limited. But our implementation is a userspace library which can be easily included into other projects. Also, each layer can be used as a base for external code. Regarding ZFS, the DMU can be accessed separately, but it is still in the kernel space and the DMU has been tightly designed around the needs of the upper layers of ZFS.

5. Related Work

There is much related work regarding the improvement of storage stacks and especially the top-most layer: the filesystem. In a practical sense, ZFS [Bon+03] improved and reinvented the whole storage stack at the beginning of the century as we discussed in Section 4.2. By using copy-on-write and adapting the interfaces of the layers to their needs, ZFS becomes a feature rich, pooled storage stack with efficient snapshots and strong data integrity.

Also, the Stratis project [Gro17] aims to provide a better user experience for the Linux storage stack. The goal of Stratis is a single simple-to-use interface that hides the complexity of using various independent layers in a storage stack.

Aside from ZFS and Stratis as practical work, there is also ongoing scientific work on improving the top-most layer of storage stacks. In the following we will focus on two aspects: improving the filesystem efficiency by using novel data layouts on disk and replacing the filesystem by a simpler low level user interface such as a key-value store interface.

5.1. Filesystem Efficiency

Today, filesystems provide good efficiency when reading or writing large files. But often, filesystem operations on small data can be very slow – most notably metadata operations on files or operations that traverse the directory hierarchy. Due to the small data, each operation involves a high overhead. Also, traversing the directory hierarchy is a sequential operation as we have to look up the parent before we can descend to the child.

Many workloads have multiple metadata operations on logical associated filesystem objects, but most filesystems cannot speed up these workloads as the metadata is not laid out on disk to exploit the accesses. Mature filesystems, such as ext4, do group metadata on disk using heuristics, however this is not sufficient.

Hence, we have to avoid fragmentation of the metadata and need a sequential on disk layout of filesystem objects w.r.t. the logical order of the data. Surely, most filesystems already have such layouts for the data of large files. But we also need optimized layouts for the directory hierarchy such as a sequential layout of directories and inodes corresponding to the trace of a *depth-first search*.

A first step regarding this goal is the *TokuFS* presented in [Esm+12]. This userspace filesystem uses a B^{ε}-Tree in which the data of filesystem objects is indexed by the filesystem path to the object. Like our design, the file data is inlined into the tree. These simple adoptions show impressive improvements. TokuFS has a much higher throughput in file creation and scanning compared to ext4 with a 9-fold speed up on creation and a

25-fold speed up on scanning. However, TokuFS has a major downside: renaming a file involves moving the file's data in the B^ε-Tree as it is indexed by the file path.

TableFS [RG13] is a similar approach to increasing the efficiency as it uses a log-structured merge tree and indexes each directory entry by the parent's inode number and the directory entry name. It achieves similar but less pronounced speed ups compared to the results of TokuFS.

At last, we will introduce *BetrFS* which is regularly improved and is the successor to TokuFS.

5.1.1. BetrFS

The first version of BetrFS is presented in [Jan+15a]. Contrary to TokuFS, BetrFS is a kernel filesystem for Linux. Thus, BetrFS can avoid the huge overhead of userspace filesystems. BetrFS uses the same data layout as TokuFS except for the internal block size of file data which has been increased from 512 bytes to 4 KiB. The conducted micro write benchmark shows the potential of using a B^ε-Tree. Executing 1000 4 bytes micro writes within a 1 GiB file is around 100 times faster with BetrFS compared to ext4, XFS, Btrfs, and ZFS. Still, the first BetrFS has the same huge overhead on file renaming as TokuFS and a write amplification of two for every operations due to write-ahead logging. Additionally, deleting files involves reading the file data.

To address the issues, BetrFS 0.2 [Yua+16] includes three enhancements. First, the write-ahead log overhead is avoided by introducing late-binding to data which is directly located in tree nodes. Second, they implemented a new range-delete operation for the B^ε-Tree that does not have to read stale leaf nodes.

The third enhancement is the most interesting one. They introduced the *zone-tree schema* that avoids file renaming overhead without sacrificing the performance of the previous BetrFS version. This new schema is a hybrid between the full file path as index and inode numbers. Instead of using the full file path, BetrFS 0.2 groups the directory tree dynamically into zones which are identified by a number. Then, the filesystem objects are indexed by their zone ID and their path in that zone. As zones are limited in their data size, large files will have their own zone. With BetrFS 0.2, renaming a directory entry is an operation local to the specific zone and, hence, the amount of work is limited. Benchmarks show that BetrFS 0.2 retains the performance best cases of its predecessor while file renaming is at most 4 times slower than ext4. Likewise, the file deletion and write throughput is very competitive.

In the latest paper involving BetrFS, Conway et al. analyze the impact of *aging* for filesystems [Con+17]. Aging denotes the performance degradation of filesystem over their lifetime. As already mentioned, filesystems may place data suboptimally. This occurs especially when many filesystem operations are executed over longer periods of time. The findings in this paper are noteworthy. When using application workloads, all production filesystems analyzed exhibit massive slow downs of up to 50 times. However, "BetrFS exhibits essentially no aging" [Con+17]. Therefore, optimizing the data layout really pays off.

Despite the missing support for directory hierarchies, our storage stack has a very

similar approach compared to BetrFS (and TokuFS). Both concepts use the B$^\varepsilon$-Tree and save user data inlined into the tree to reduce indirections. BetrFS has a block size of 4 KiB whereas we use a block size of 128 KiB. The underlying B$^\varepsilon$-Tree implementation of BetrFS is however much more mature as it uses the database engine TokuDB [Per18; Per17] which is ACID compliant, supports transactions, and features a write-ahead log. Also, BetrFS provides a filesystem interface and not a simpler key-value store interface as we do.

5.2. Low Level User Interfaces

Another approach for improving the performance and usability of storage stacks is to replace the top-most layer: the POSIX-compatible filesystem. Especially for HPC applications, a full-blown POSIX filesystem is often not needed and even becomes a burden. For distributed filesystems, the strong POSIX consistency semantics require a distributed locking mechanism which often limits the performance due to increased communication needs. Some HPC filesystems solve this issue by relaxing the POSIX semantics such as OrangeFS [Koz+14].

A more extensive approach is to replace the filesystem interface by simpler interfaces with a smaller API surface and less consistency semantics. Similar to the NoSQL databases that provide interfaces which are more suited for specific workloads than SQL databases, a storage stack can export a key-value store interface instead of a filesystem when a key-value store is suitable for certain workloads. *Tucana* [Pap+16] is a key-value store whose data structure is based on B$^\varepsilon$-Trees. Compared to HBase and Cassandra from the Hadoop ecosystem which are based on log-structured merge trees, Tucana has lower CPU overhead and a 2- to 10-fold higher throughput in the Yahoo Cloud Serving Benchmark. Their findings suggest that B$^\varepsilon$-Trees are competitive to LSMs which are still much more popular today.

5.2.1. DAOS

DAOS (Distributed Asynchronous Object Storage) is a distributed storage stack with a key-array store interface and support for multi-tier storage systems and byte-granular persistent storage [Dev17a; Lof+16; Lom+16; Dil17; LZ13].

Its key feature is extreme scalability. In contrast to the previously presented storage stacks, DAOS is tailored for HPC applications but like most storage stacks, DAOS has multiple layers. The top-most and optional IO Dispatcher (IOD) layer is responsible for data rearrangement and data processing between the compute nodes and actual storage nodes. It provides burst buffers to absorb the I/O peaks of HPC workloads and a transaction mechanism so that a checkpoint of the HPC application is not visible to others until the checkpoint has been written successfully.

The next layer is called DAOS and provides a parallel file system interface. Through this interface, a user can access the actual data in a consistent, global view. The DAOS layer manages multiple *containers* which are a hierarchical namespace for *objects* which

in turn are either blobs or multi-dimensional arrays. By using arrays instead of blobs, the DAOS layer can automatically stripe the underlying data among multiple storage nodes for better performance. The transactions of the IOD layer are implemented with *epochs* and copy-on-write at the DAOS layer so that there can be multiple versions of the same object.

The bottom layer is the Versioning Object Storage Device (VOSD) layer. This layer provides access to the storage devices and is actually quite similar to the other presented storage stacks. A first prototype even used ZFS at the VOSD layer [Lof+16].

All in all, the design of DAOS describes a concept regarding how to scale out our storage stack for distributed, parallel usage such as for HPC. Like our storage stack, DAOS is implemented in the userspace as this simplifies the overall implementation and it should also provide a performance benefit compared to a full implementation in the kernel.

6. Implementation

In this chapter, we introduce the systems programming language Rust which is used for the whole storage stack implementation. In particular, we will analyze the benefits of using Rust and also compare Rust against other popular systems programming languages. Afterwards, we will have a look at interesting details of the implementation – such as how the I/O is actually performed – and performance optimizations.

6.1. The Rust Programming Language

The implementation of the storage stack is completely written in Rust. Rust is a systems programming language whose first stable version was released in 2015. Rust's distinctive feature compared to other systems programming languages is that it guarantees memory safety and thread safety without the need for an intrusive runtime such as a garbage collection. Hence, it is especially suited as a replacement for C or C++. Having these safeties, we can program more fearlessly without worrying about dereferencing dangling pointers, having invalidated iterators, accessing uninitialized memory, and so forth.

In addition to its safety guarantees, Rust also has other noteworthy features such as its powerful type system which can prevent logic bugs at compile time as we will see later on. Likewise, Rust encourages the *resource acquisition is initialization* (RAII) principle and has integrated concepts for error handling. Furthermore, Rust provides *object-oriented programming* (OOP) based on traits, zero cost abstractions, pattern matching, and type inference. The ecosystem of Rust includes an easy-to-use build tool and package manager, unit tests, automated documentation generation, and much more.

For more information, see the Rust website: `https://www.rust-lang.org`

6.1.1. Rust Compared to Other Popular Languages

In this section, we want to compare Rust to other programming languages which are often used for systems programming: C, C++, and Go. Neither of these languages guarantee memory safety or thread safety.[1] So, developing multi-threaded software can be very challenging in these languages as data races might occur which result in hard-to-debug Heisenbugs, i.e. bugs that seem to disappear when debugged because the bug's behavior depends on the program's environment. This also makes refactoring of code difficult

[1] Despite being a very young systems programming language, Go is neither memory nor thread safe. See [Sta] for more details.

because each refactoring might introduce concurrency bugs. In the following we will take a closer look at other useful features besides safety that the other languages are missing.

C

The widely-used C systems programming language is a very old language that barely provides any abstractions at all. It only has a very basic type system which allows the user to declare structures. Due to the missing generics, many C libraries have the notorious `void` pointer in their interfaces and rely on the casting to the correct type. Likewise, the error handling story of C is very basic. Most functions return an integer which indicates the success or error of the function call. Often, the error handling is forgotten and the error will be ignored by accident. For example, the error propagation of file systems and storage drivers in Linux 2.6 has been analyzed and the findings are astonishing: "1153 calls (13%) drop an error code without handling it" [Gun+08]. Additionally, C makes it hard to ensure that certain actions are performed on exiting a code block such as unlocking a mutex. As error handling often introduces multiple exit paths, this becomes very tricky.

On the other hand, Rust solves all of these problems. It provides high-level abstractions, including generics, due to its type system and the handling of errors is enforced by Rust so that the mistakes cannot happen. Also, Rust provides RAII guard types that ensure that, for example, mutexes are unlocked.

C++

C++ has many improvements compared to C: It supports OOP based on classes and inheritance for better abstractions, RAII for avoiding resource leaks, and templates for generic programming. Still, C++ does not need a runtime and, hence, C++ is often used in embedded programming projects.

Nonetheless, Rust has some advantages over C++. First, the C++ templates are mainly more powerful C macros and do not have static type checking so that the code has to be type checked on each instantiation. Additionally, Rust includes a package manager that makes it easier to reuse code of other people. Also, the type system of Rust is much more powerful so that some invariants of specific types – such as "*can be only be used exactly once*" – can be enforced by the type system which prevents bugs. Regarding OOP, Rust encourages *composition over inheritance* – as recommended by [Gam+95].

Go

Go is very similar to C but with interfaces, better error handling concepts, native coroutines, a package manager, and automated memory management with a garbage collection. Like C, its type system does not support generics, except for the basic `array`, `map`, and `slice` types. Also, Go encourages to avoid concurrency issues by the usage of messages passing between coroutines. This concurrency concept might not be convenient for some problems and, still, Go is not thread safe as already mentioned. In addition,

the runtime needed for the garbage collection makes Go unsuitable for embedded or low level programming as opposed to Rust.

6.1.2. Preventing Bugs with Rust

Considering the following excerpt of the *Storage Pool* interface definition, we will see how Rust's type system prevents bugs:

```
1  /// A `StoragePoolLayer` which manages vdevs
2  /// and features a write-back queue.
3  pub trait StoragePoolLayer: Send + Sync {
4      type Checksum: Checksum;
5
6      /// Reads `size` blocks from the given `offset`.
7      /// Verifies the data with `checksum`.
8      fn read(
9          &self,
10         size: Block<u32>,
11         offset: DiskOffset,
12         checksum: Self::Checksum,
13     ) -> Result<Box<[u8]>, VdevError>;
14
15     // other associated types and functions...
16 }
```

In line 3, we declare the new trait and already specify that for a type that implements this trait, every instance is `Send` and `Sync`, i.e. it can be safely sent to other threads and shared with other threads. Line 4 declares an *associated type* that has to satisfy the `Checksum` trait. This type is chosen by the implementation of this trait. Afterwards, we declare the function `read` that every Storage Pool has to implement. This function either returns an owned sequence of bytes on success or an error on failure.

This interface prevents the following kind of bugs: Using the newtype pattern [Con16b], we ensure that every caller knows that the `size` parameter specifies the size in blocks and not, for example, in bytes. Likewise, the `offset` is of type `DiskOffset` which is just an unsigned integer, but the type guarantees certain invariants on this integer. Last but not least, the third parameter `checksum` must be of the `Checksum` type that this implementation supports so that no accidental type confusion is possible. The return type `Result` is a sum type which enforces that the function either succeeds or fails. The caller has to check for a potential error. If the functions fails, there is no possibility to accidentally access the data block pointer and, hence, uninitialized memory. Also, the type for the read byte sequence is `Box<[u8]>` which specifies that the caller is responsible for deallocating it afterwards.

The implementation of the whole storage stack uses these safety measures all over the place and these measures have prevented countless of bugs during multiple refactorings.

6.2. Implementation Details

Now, we will have a look at the actual implementation of the storage stack. As its design has already been described in Chapter 3 and its structure is very similar to the design, we will focus on interesting implementation details instead of walking through the code.

6.2.1. Performing I/O

A very important aspect of the implementation is how it actually performs the disk I/O as the I/O interface can be performance limiting. Hence, choosing the "right" I/O interface is the key to good performance. In Linux userspace, we have multiple I/O interfaces to choose from:

Sync I/O Simply call the synchronous I/O syscalls `pread` and `pwrite` from the client thread. The thread will be suspended until the I/O has been executed.

Linux Async I/O The Linux kernel provides its own asynchronous I/O interface. With the `io_submit` syscall, we can queue multiple I/Os which will be executed asynchronously and maybe simultaneously. With the `io_getevents` syscall, we can query or wait for completed I/Os.

POSIX Async I/O The POSIX standard also defines an asynchronous I/O interface. *Glibc* which is the default C library on Linux, implements this interface by executing synchronous I/O syscalls on a thread pool.

Sync I/O with Thread Pool Similar to the glibc, we can also use our own thread pool to offload synchronous I/O syscalls.

The implementation needs an I/O interface which allows us to execute multiple I/Os in parallel so that we can exploit the performance of many HDDs. Especially parity1 vdevs needs this as each request to this vdev will be split into multiple smaller I/Os – one for each disk in the group. Hence, the *Sync I/O* interface is unsuitable as it serializes the I/O requests.

A small benchmark which measured the latency for six read I/Os issued to six disks indicated that the *Linux Async I/O* interface actually has a 10% higher latency compared to the "poor man's" asynchronous I/O interfaces. Also, the interface is much more complex to use and lacks support for `fsync` [Cor16]. Hence, this interface was not examined any further.

So we remain with two I/O interfaces whose implementations are nearly identical. The implementation uses the *Sync I/O with Thread Pool* interface as its implementation is very easy in Rust because Rust already supports synchronous I/O in its standard library and there are Rust packages that provide thread pools which can be managed very easily.

6.2.2. Guaranteeing Invariants

As mentioned in Chapter 3, the ObjectReference type has an invariant:

There is only one ObjectReference pointing to a mutable object.

Because the algorithms of the Data Management Layer rely on this fact, it is important to guarantee this invariant to prevent bugs. Likewise, the Data Management Layer interface has many functions which need a unique, mutable access to objects. With Rust, we can easily implement the invariants for these types and interfaces as we will see in the following.

Rust has a concept of ownership and borrowing and uses these concepts to enforce the following rule: For each object, there is either shared immutable access or unique, mutable access.[2] So, if our interface requires mutable access to the objects, Rust already guarantees that this access is unique. Now, to guarantee the ObjectReference invariant, we only need to preventing cloning of ObjectReference objects. As the objects of types are not cloneable by default in Rust, this is essentially a freebie in Rust.

6.2.3. Cache

The design does not further specify what kind of cache shall be used and leaves this as an implementation decision. The implementation uses the simple CLOCK cache [Cor68]. CLOCK is similar to Second Chance and is a 1-bit approximation to the popular LRU cache. The CLOCK cache consists of a circular list which contains the cache entries and each cache entry has an associated *reference bit*. Its advantages is the low overhead compared to LRU: Whereas LRU needs to update a linked list structure on each cache access, CLOCK simply sets the associated reference bit. On eviction, we walk the circular list and inspect each cache entry. First, we inspect the cache entry at the head of the list. If its reference bit is not set, this entry will be evicted. Otherwise, the reference bit will be reset and the circular list is advanced by one so that the head of the list is now the tail of the list. On insertion, the cache entry is appended to the tail of the list.

This cache implementation should be sufficient with respect to performance. Many modern caches focus on low overhead and *scan resistance* such as the ARC [MM03] used in ZFS. Scan resistance means that first, a cache can detect when an operation involves inserting many cache entries which are used only once, for example when a file is sequentially read. Second, the cache protects more frequently accessed cache entries from eviction due to the insertion of the cache entries from the scan operation. This should result in an overall higher hit ratio. But scan resistance is probably not important for us because of the following.

For tree structures, it is important that the upper levels of the tree are cached. Regarding our implementation, a typical scan operation is the B^ε-Tree range query. But the range query implementation regularly accesses the upper nodes of the tree as it repeatedly walks root-to-leaf paths. Hence, the reference bits of the upper tree nodes are often "refreshed". Therefore, scan resistance is not an important feature as we solely use B^ε-Trees as data structures.

[2] Otherwise, there would be shared, mutable state which is the root of all evil with respect to thread safety.

6.2.4. Deadlock Avoidance

The design describes a (possibly) circular usage of the Database Layer, the Data Management Layer, and the Allocation Handler. As the implementation uses locks for tree nodes, this can lead to deadlocks. The following scenario shows that this is actually possible:

> An insertion into the root tree locks the root node and inserts the message. As the (internal) root node overflows, some messages shall be flushed to a child node. As the child was unmodified beforehand, the Allocation Handler receives the callback `copy_on_write` so that the blocks of the old child node can be deallocated. The Allocation Handler wants to update the allocation bitmaps and this leads to an insertion into the root tree which needs to lock the root node. At this point, the thread deadlocks.

As the implementation must be deadlock free, this deadlock has to be avoided. This is achieved by breaking up the cycle: Instead of directly inserting the deallocation messages that occur in `copy_on_write` calls, the Allocation Handler will delay these messages. This has no effect on allocations later on if the node was written in a previous generation as its blocks may not be overwritten in this generation. The implementation ensures that the delayed messages are inserted before the next sync.

6.3. Optimizations

The implementation includes some basic optimizations to improve the performance which will be presented in the following.

6.3.1. Zero-Copy Leaf Nodes

The implementation uses the bincode [Con17a] format to serialize data for on-disk storage. Despite the fact that bincode is a space efficient binary format, profiling showed that the data serialization comes at a significant overhead: During large range queries, about 30% of the runtime of the library is consumed by the library function `memcpy` which moves data around in memory. `memcpy` is used when deserializing keys, messages, and values in the tree nodes.

To avoid the deserialization overhead, the implementation has a special on-disk data structure for leaf nodes so that leaf nodes do not have to be deserialized if the node is not modified. Reading a leaf node in this data structure does not involve any data copies at all – hence, it is called *zero-copy*. The data structure is a sorted, packed key-value list with all fields in little endian so that a typical computer architecture can efficiently execute all leaf nodes operations on this data structure.

With this optimization, the CPU usage in the micro benchmark dropped significantly and `memcpy` only accounts for 2% of the total runtime.

6.3.2. Prefetch

For range queries, the implementation has a simple prefetching mechanism: Every time, the algorithm does the root-to-leaf walk, the immediate right leaf sibling will be prefetched if it exists. Despite the simplicity, this optimization shows decent improvements for range queries in a micro benchmark: On a single disk, the throughput improves by a factor of 1.5. On striping with two disks, we gain a speed up of 4.

Still, the prefetching could be more aggressive and fetch multiple nodes in parallel, but due to the increased complexity this was not investigated any further.

6.3.3. Vdev Selection

Last but not least, there are multiple spots where the implementation can choose between multiple vdevs to perform an I/O. The mirror vdev, for example, can read data from any of the mirrors. Likewise at the storage pool level, we can allocate and write to any of the top-level vdevs. "Intelligent" decisions of vdevs at these spots can result in significant speed ups.

Ideally, the algorithms should make informed decisions based on the current utilization of the underlying devices like ZFS does for example. The implementation uses simpler algorithmic decision making as it is nontrivial to compute the utilization of devices. The mirror vdev will read from a device depending on the offset requested. Every 32 MiB the preferred device changes so that a mirror HDD has a high utilization on sequential reads. At storage pool level, the top-level vdevs are selected in a round-robin fashion for allocations.

7. Evaluation

In this chapter, we evaluate the performance of the three present storage stacks: our storage stack, ZFS, and ext4 on top of an MD device (Linux software RAID). We will analyze the scaling of sequential throughput with increasing disk counts, the write throughput of our storage stack with different cache sizes, and finally, we will run a realistic database workload with random writes and sequential reads.

7.1. Methodology

All benchmarks are conducted on a server with a quad-core E3-1225 v3 @ 3.2 GHz CPU and 12 GiB RAM. Six Seagate ST3500418AS 500 GB HDDs are connected to the computer via an LSI SAS3081E-R SAS HBA controller. The server runs Debian 9.3 with Linux 4.12.13 and ZFS on Linux 0.7.3. The used Rust compiler was rustc 1.24.0-nightly (f9b0897c5 2017-12-02). Fio 2.16 was used as the benchmark program.

The following configurations were tested:

db The storage stack of this thesis compiled with the mentioned Rust compiler in release mode and with the default cache size of 256 MiB except where otherwise specified.

ZFS A ZFS storage pool with default settings except where otherwise specified.

ext4 An ext4 filesystem on top of a MD device. The ext4 filesystem has been created with the appropriate `stride` and `stripe_width` options so that ext4 can exploit the underlying disk configuration. Also, `lazy_itable_init` and `lazy_journal_init` were deactivated so that the filesystem is fully initialized before the first mount.

For testing the proposed storage stack, a fio engine has been written which directly calls the functions of the storage stack library. For ext4 and ZFS, the default `psync` engine has been used which opens a file on the filesystem and uses `pread` and `pwrite` to perform I/O. On ext4, the file will be preallocated with `posix_fallocate`. On ZFS, `ftruncate` is used instead as ZFS on Linux does not support preallocation.

7.2. Storage Stack Scaling

The first benchmark analyzes the scaling of sequential I/O throughput with increasing disk counts. In theory, striped configurations scale linearly with the number of disks and

RAID5 scales linear with the number of data disks, i.e. number of disks minus one. For mirrors, the read throughput scales linearly, but the write throughput slightly decreases.

As workload, this benchmark reads and writes data sequentially with a 128 KiB block size. The following configurations are used: Striping with 1 to 6 disks, RAID5 with 3 to 6 disks, and mirror with 1 to 3 disks. The results are depicted in Figure 7.1, Figure 7.2, and Figure 7.3. Now, we will analyze the results.

7.2.1. Striping

Starting off with the ext4 read performance on striped disks, we can already see a near perfect scaling. From one to two and two to three disks, the performance is within 5% of the perfect scaling. Beyond three disks, ext4 even exhibits a perfect scaling. Write performance looks very similar at first. For two and three disks, the performance is within 10%, and for four and five disks, it scales perfectly. But for six disks, the write performance falls off dramatically as we already have the same throughput with four disks. There is no obvious reason why the ext4 write performance does not scale beyond five disks. Admittedly, this configuration is not common in practice.

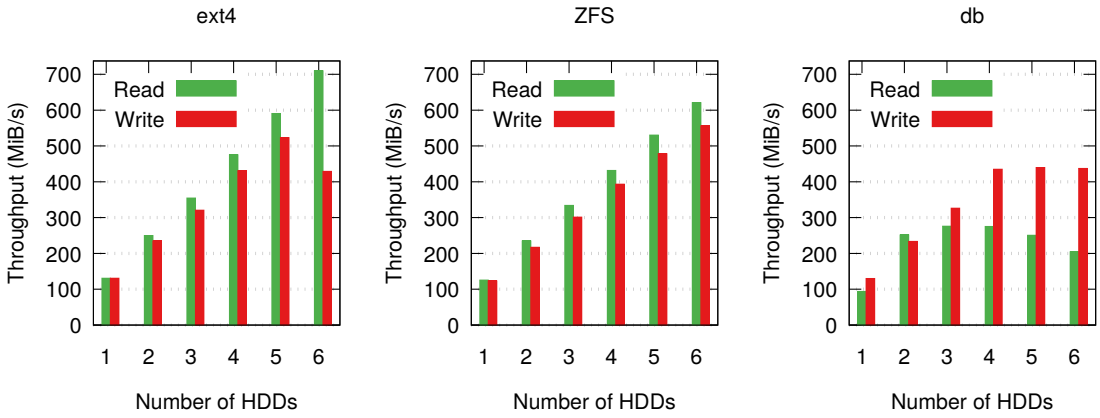

Figure 7.1.: Sequential read and write scaling at 128 KiB block size with striping across 1 to 6 disks.

Next, we look at the performance of ZFS. Similar to ext4, the read and write performance scales a little less for two and three disks than the scaling beyond three disks. Nonetheless, ZFS shows a very steady scaling up to six disks for read and write throughput.

As last candidate, we have our storage stack whose performance scaling is not steady at all and the overall performance is mediocre compared to ext4 and ZFS. Beginning with the read performance, we can see that our storage stack has a much lower throughput with about 95 MiB/s for one disk compared to the competitors whose throughput is about 125 MiB/s. Interestingly, the tide turns with two striped disks. Now, our storage stack has the best read performance even though ext4 is just 1 MiB/s slower. With three disks, we gain only about 10% and beyond that the read performance falls off. Regarding

the write performance, the throughput is about the same as ext4's and scales very well up to four disks. With five or six disks, it stagnates.

All in all, ext4 and ZFS show a good sequential read and write performance with striped disks. The write performance of our storage stack is also very good. The read performance however requires further analysis. The extraordinary read throughput with two disks is probably due to the optimizations presented in Section 6.3. As the immediate sibling of a leaf node is prefetched and the node are striped in a round-robin fashion on write-back, our storage stack fetches exactly one node from each of the two disks in parallel. This results in a perfect utilization of both disks. For striping across more disks, we probably need to prefetch multiple nodes in parallel. It is not clear why the read performance falls off with five or six disks instead of stagnating. A short follow up benchmark indicates that increasing the number of I/O threads in the thread pool does not influence the read performance fall-off.

7.2.2. RAID 5

Regarding RAID5, ext4 again shows good read throughput and a near perfect scaling. The write throughput is considerably slower and also scales less well. However, this is expected due to the parity overhead on write. The read throughput of ZFS is overall very steady scaling and a little less compared to ext4. Regarding the write throughput, ZFS starts with a higher throughput at three disks compared to ext4, but falls off drastically and stagnates at six disks.

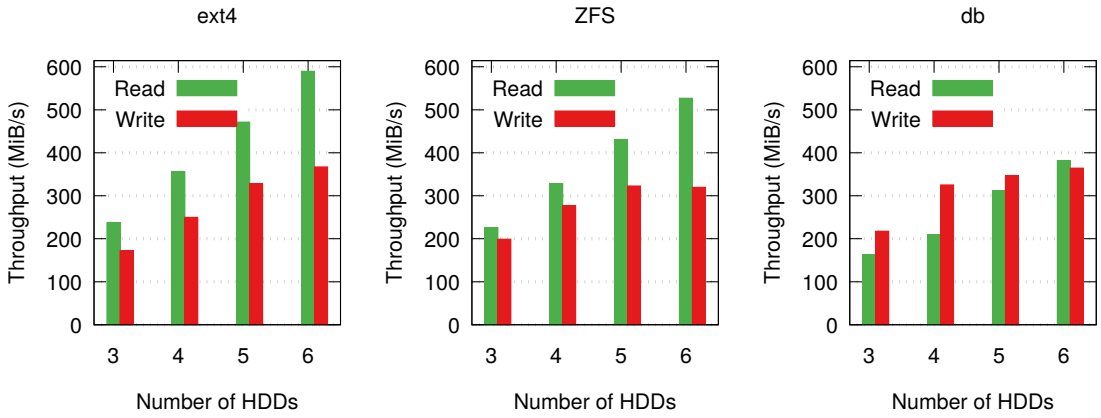

Figure 7.2.: Sequential read and write scaling at 128 KiB block size with RAID5 across 3 to 6 disks.

Surprisingly, our storage stack exhibits the highest write performance. Especially at lower disk counts, our storage stack is remarkably faster. At higher disk counts, the margin decreases as the write performance does not scale well beyond four disks. The read throughput is again quite low as ext4 is about 50% faster. In contrast to striping, the read throughput scales up to six disks.

All things considered, the throughput and scaling with RAID5 is very similar to striping. Again, ext4 comes out on top and ZFS is a very competitive runner-up. Likewise, our storage stack has superior write performance but mediocre read performance.

7.2.3. Mirror

As the last scaling benchmark, we will look at the performance of mirrors. The results are depicted in Figure 7.3.

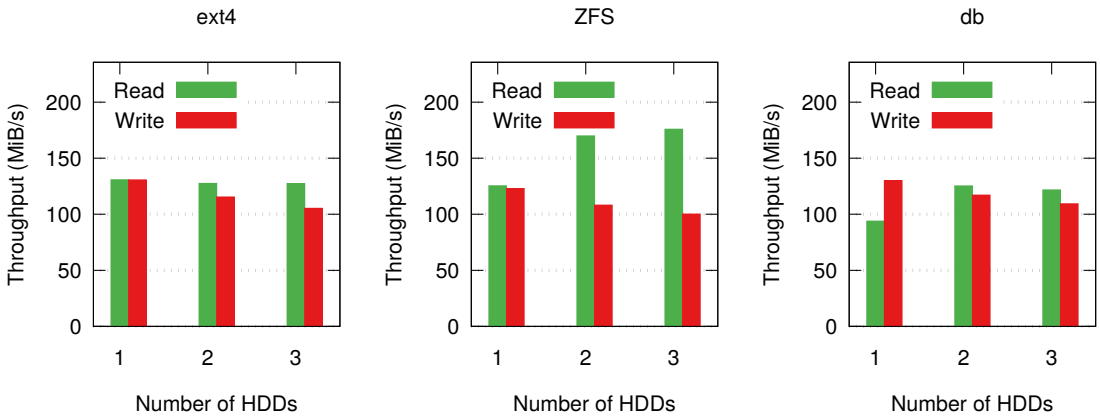

Figure 7.3.: Sequential read and write scaling at 128 KiB block size with a mirror across 1 to 3 disks.

Regarding the write performance, all storage stacks have a similar throughput. Likewise, each storage stack exhibits a slight slow down of about 10% per added mirror disk. Regarding read performance, ext4 does not scale at all with multiple mirror disks while ZFS and our storage stack have a speed up of about 35% compared to the single disk case.

Looking at the typical case, a mirror of two disks, ZFS excels at the read throughput with about 170 MiB/s compared to about 125 MiB/s for ext4 and our storage stack.

7.3. Write Throughput with Different Cache Sizes

With the second benchmark, we evaluate the influence of the cache size on write throughput in our storage stack. As we cache dirty nodes and insert new data at the root node, a larger cache allows us to better layout incoming data in the tree before accessing the disk. Hence, random write workloads should gain a higher throughput with a larger cache. For the sequential workload, we will use 128 KiB as block size. For the random workload, we use the following block sizes: 4 KiB, 128 KiB, 1 MiB, and 4 MiB. We will test the write throughput for each of these access patterns with the following cache sizes: 256 MiB, 512 MiB, 1 GiB, 2 GiB, and 4 GiB. This benchmark uses only a

single disk so that scaling issues do not influence the result. In the following, we will analyze the results which are depicted in Figure 7.4.

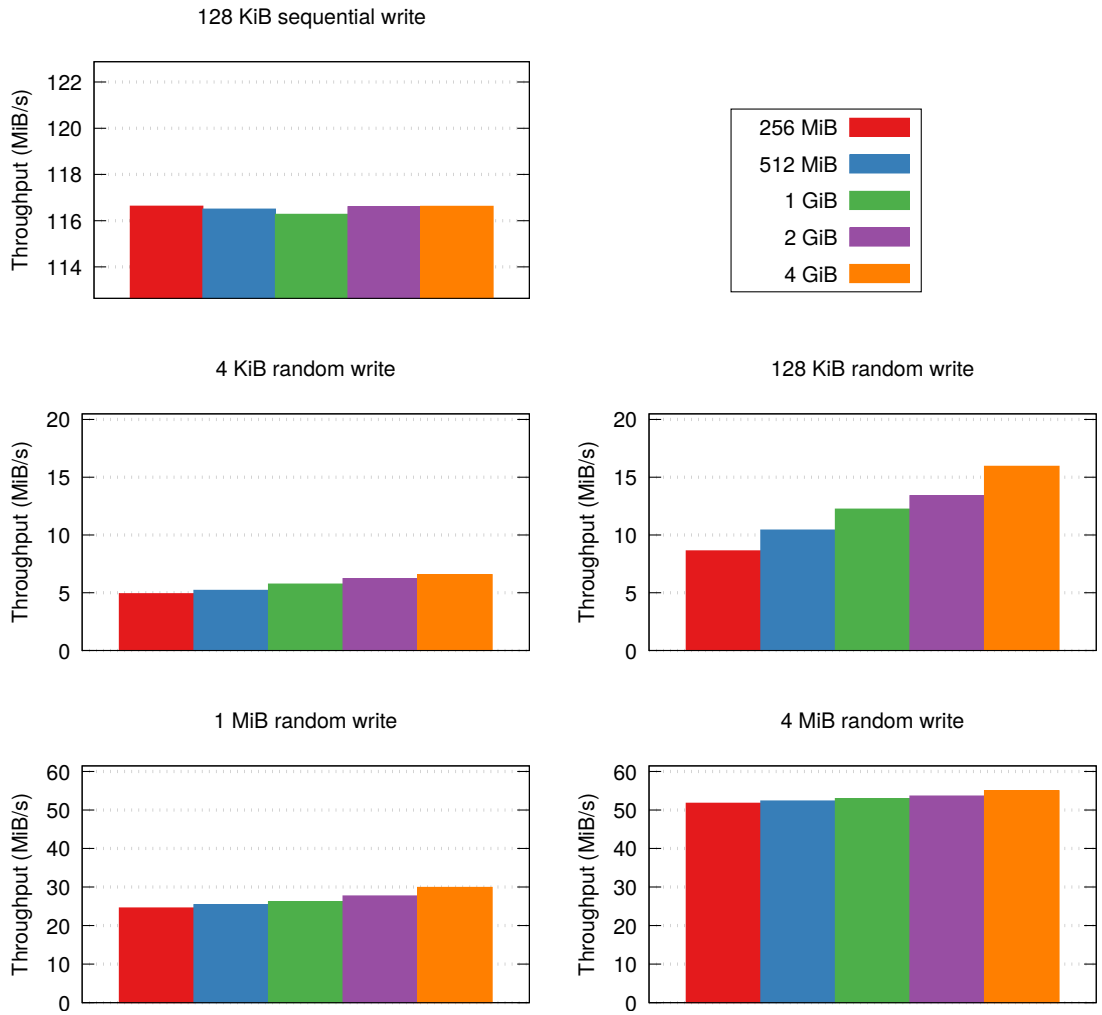

Figure 7.4.: Write throughput with different access patterns and cache sizes from 256 MiB to 4 GiB.

The cache size does not benefit the sequential write throughput at all. Regarding the random write throughput, we see notable improvements with larger cache sizes. For 4 KiB block size, the average throughput gain is about 7.5% per cache size doubling and respectively 34% in total. At 128 KiB block size, we see the highest gain with 16.5% per step and 85% in total. With larger block sizes, the gain falls off and is only about 5% per doubling for 1 MiB and 1.5% for 4 MiB.

Coming to a conclusion, there is a notable throughput gain for smaller block sizes and random access patterns. For larger block size, there is a gain but not as pronounced. As expected, sequential workloads are not affected at all. Hence, for random workloads with small block sizes, it is wise to increase the cache size as it is defaults to 256 MiB. Note

that the cache size negatively affects the sync performance. After an intensive write phase, the cache is full of dirty nodes which have to be written back to disk prior to the sync which takes a long time if the cache is large.

7.4. Database Workload

The last benchmark simulates a database workload. On insertions or updates of tuples, databases tend to issue a large number of small random writes because the tuples are typically not in the same order as the database's index. (Otherwise, we do not need an index in the first place.)

With a high volume of small random writes, a database workload shows the influence of these writes on data layout and fragmentation. This benchmark involves three phases. First, a 20 GiB file will be written sequentially so that the data blocks have a perfect layout. Then, we will issue random writes of 8 KiB block size so that in total 5 GiB of the file will be overwritten. Finally, we sequentially read the file with 128 KiB block size. Caches are emptied after each phase. As in the case of the cache size benchmark, we only use a single disk.

For copy-on-write storage stacks such as ZFS and our storage stack, this workload can lead to excessive fragmentation due to the small partial overwrites which will in turn lead to mediocre sequential read performance. For ZFS, we additionally benchmark with a *record size* set to 8 KiB so that ZFS can avoid read-modify-write cycles. This is also recommended in practice [Con16a]. The results are depicted in Figure 7.5.

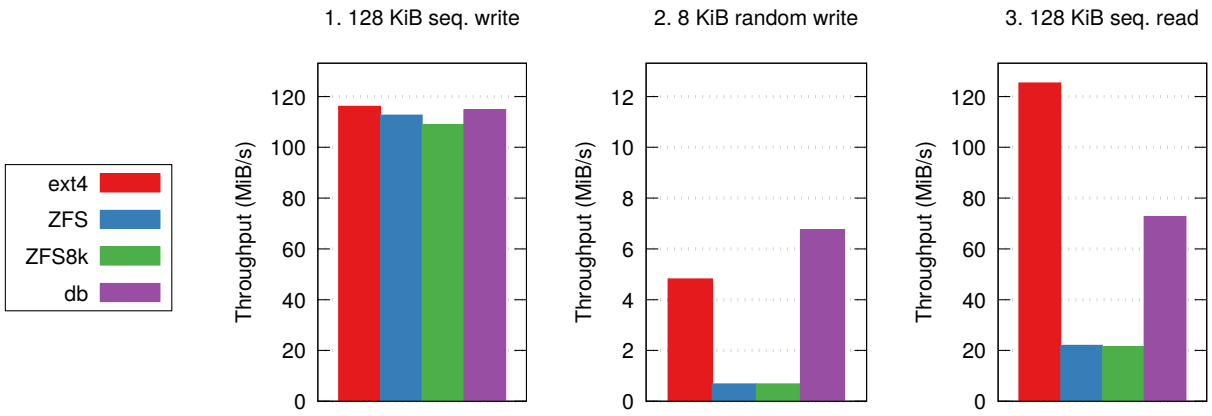

Figure 7.5.: Database workload. First, a 20 GiB file is sequentially written with 128 KiB block size. Then, the file is partially overwritten (5 GiB in total) with a random access pattern and 8 KiB block size. Finally, the file is read sequentially with 128 KiB block size. "ZFS8k" stands for ZFS with record size set to 8 KiB.

Ext4 again shows a decent performance in this benchmark. In the random write phase, ext4 issues a large number of random I/Os but due to the default settings of Linux,

2.4 GiB of dirty data will be cached in the page cache. Hence, a huge number of I/Os can be reordered to increase the performance so that resulting write throughput is about 4.7 MiB/s. As ext4 overwrites the data in place and, thus, no fragmentation occurs, the final read is also very fast with 122 MiB/s.

ZFS has a slight sequential write throughput decrease with a record size of 8 KiB instead of the default 128 KiB. In the random write phase, the write throughput is only about 0.66 MiB/s. Hence, ZFS must issue lots of small random I/Os. The smaller record size does not influence the performance. The final read is very slow with about 21.5 MiB/s considering that ZFS has shown a throughput of 125 MiB/s for sequential data in the first benchmark. With 8 KiB record size, ZFS even is about 500 KiB/s slower compared to the default settings. The low read throughput of ZFS is due to the fact that the file data becomes heavily fragmented on disk because of the small random writes.

Finally, our storage stack has a high write throughput of 6.6 MiB/s in the second phase and a good read throughput of 71 MiB/s in the final phase. As the B^ε-Tree is a write-optimized data structure, it can ingest random writes very well. Hence, our storage stack achieves a high write throughput even with small random writes. At the same time, the sequential read throughput does not suffer that drastically like for ZFS because we limit the maximum possible fragmentation of the file data. With the default settings, the nodes of the B^ε-Tree have a fanout between 4 and 16 and a size between 1 MiB and 4 MiB. Hence, the tree's height is about 5 for this workload. Assuming we can cache a full root-to-leaf path, a sequential read of this tree's data needs between $1.\overline{3}$ and $0.2\overline{6}$ seeks per MiB and less than one seek per 2 MiB on average. Due to the low number of seeks needed, the throughput is correspondingly high.

This benchmark shows the benefits of having a write-optimized data structure. Compared to ZFS, our storage stack is not only 10 times faster in the write phase, but also about 3 times faster in the read phase. Ext4 is also quite fast in the write phase due to the excessive write caching. The low performance of ZFS in the write phase is probably not due to the writes of the actual data but due to metadata and/or allocation overhead which might involve a large number of small random reads and writes.

7.4.1. Conclusion

As outcome of these benchmarks, we have the following conclusions:

Ext4 is the winner for sequential workloads

This result is not surprising at all considering that ext4 is the only storage stack tested that is not using copy-on-write and uses extents. Hence, ext4 has the lowest overhead which results in good sequential performance.

The sequential performance of ZFS is very competitive

With the design of ZFS in mind, the overall performance of ZFS is impressive. Especially, the quite consistent scaling indicates a well-thought-out code base for sequential

workloads.

Our storage stack exhibits an exceptionally high sequential write throughput

Surprisingly, our storage stack has the highest sequential write throughput. This is probably due to a larger block size which increases the overall efficiency. The Linux software RAID uses a chunk size of 512 KiB by default and ZFS uses block sizes up to 128 KiB. In contrast to this, our storage stack uses node sizes ranging from 1 MiB to 4 MiB. Especially for parity generation, the larger block size should be beneficial.

Our storage stack does not scale beyond 4 disks for sequential writes

This may be due to the I/O interface used and requires further investigation.

Our storage stack has mediocre sequential read performance

Probably, the prefetching is not aggressive enough. We have to prefetch multiple nodes at the same time. Also, offloading this work to a background thread might help.

Read throughput of mirrors does not scale well

The proclaimed linear scaling of read throughput did not occur. Astonishingly, ext4 on top of the Linux software mirror does not scale at all which is a missed optimization. Regarding ZFS and our storage stack, a two disk mirror does improve read throughput but only by 35%. Adding a third disk does not improve the read throughput any further.

Our storage stack's random write throughput slightly benefits from large caches

For small block sizes and especially for 128 KiB, we can have a decent performance increase of up to 40% by doubling or quadrupling the default cache size of 256 MiB.

Our storage stack shines at random (over-)write workloads

The last benchmark shows that limiting fragmentation by design can matter. While the read throughput of ZFS greatly decreased from 122 MiB/s to 21.5 MiB/s, our storage stack only went from 91.5 MiB/s to 71 MiB/s as it retains the sequential data layout much better. Hence, it is possible to build copy-on-write storage stacks that prevent excessive fragmentation by design.

8. Conclusion

In this thesis, a storage stack has been designed, implemented, and evaluated. At first, we looked at the theoretical background of building index data structures for persisting data on disk. Besides the common B-Tree, we presented the B^ε-Tree which is a generalization of the former. The B^ε-Tree is a write-optimized data structure which, unlike the B-Tree, caches data in its internal nodes before flushing parts of it to a subtree. Due to this technique, the B^ε-Tree has a very good insertion performance and can use larger nodes than the B-Tree which results in better utilization of the disk's bandwidth. Also, we defined a list of key features that a storage stack should support.

Using the B^ε-Tree as basic building block, we presented a storage stack design in a layered approach with a key-value store interface. The bottom layers, the Vdev Layer and the Storage Pool Layer, are responsible for striping data among multiple disks and providing redundancy to protect against disk failures. Due to the use of checksums, the Vdev layer can detect and repair corrupt data even if the underlying disks return wrong data. On top of the Storage Pool Layer is the Data Management Layer that handles generic objects which may contain references to other objects. The Tree Layer implements a B^ε-Tree by using the DML objects as tree nodes. The top-most Database Layer provides data sets which save their data inlined into B^ε-Trees. By using copy-on-write, the storage stack can easily provide snapshots of the data sets and, in addition, the on-disk state is always consistent.

Then, this storage stack was theoretically compared to the Linux storage stack and ZFS where it had an outstanding snapshot efficiency compared to LVM and an integrated concept of limiting the data fragmentation in contrast to ZFS.

The latter was also practically evaluated in benchmarks including ext4, ZFS, and the storage stack of this thesis. Despite not being on a par with the competition, our storage stack provided decent sequential performance. In a database workload which can result in heavy data fragmentation for copy-on-write storage stacks, our storage stack excelled by being several times faster than its direct competitor ZFS.

All in all, we have seen how to build a modern copy-on-write storage stack with support for many convenient features. Using copy-on-write as an enabler for important features such as snapshots and on-disk consistency comes normally at the price of excessive fragmentation for certain workloads. We have successfully avoided this issue by using the B^ε-Tree which inherently limits fragmentation.

9. Future Work

In this chapter, we will look at possible extensions to and optimizations of the implementation of the storage stack. Also, other directions for future work with this storage stack are stated. At the end, we will take a look at proposed future work regarding the B^ϵ-Tree.

9.1. Extensions to the Implementation

The storage stack of this thesis is still missing many key features mentioned in Section 2.4. Most of them can be implemented quite easily because of certain design choices. Also, there are some obvious extensions and optimizations. In the following we will discuss the features and sketch the implementation for some of these.

Double-Parity and Beyond Our Vdev Layer only provides a single parity vdev but this can be easily extended to provide vdevs with multiple parity blocks per stripe. The complicated part is the parity generation and recovery as the single parity with XOR is just a special case and the mathematics behind parities is quite complex. See [Pla97] for an excellent introduction into Reed-Solomon coding with an arbitrary number of parities.

Accounting Currently, the implementation does not account any space information. It would be very useful to add space accounting for data sets and snapshots so that the user can see the space consumption of each data set and snapshot. The implementation is straightforward and solely requires to count block (de)allocations and block deaths for every data set.

Better Prefetching As outlined in Chapter 7, the sequential read throughput could be increased by a better prefetching algorithm which prefetches multiple nodes in parallel.

Clones Similar to ZFS and LVM, the implementation should support clones, i.e. writable snapshots. This feature can be implemented by a list of clones for each snapshot and a new property of every data set called *origin* which is the snapshot of origin if appropriate. If a snapshot has a clone, it may not be deleted. If a data set is a clone, it does not deallocate blocks but always appends these to its dead list. This concept results in fully functional clones that even can be snapshotted or cloned.

Rollback We should support the rollback of a data set to a specified snapshot. This involves deleting newer snapshots and deallocating all blocks that were allocated after the snapshot was created.

Authenticated Encryption We can provide transparent authenticated encryption per data set quite easily by encrypting each DML object individually and adding corresponding fields to the ObjectPointer. However, the implementation also needs to provide flexible encryption key management.

Send/Receive Like ZFS, the implementation should support the sending/receiving of data sets and snapshots to/from other storage pools. This feature needs to enumerate changes to a data set and a stream representation for these changes and other nontrivial extensions to the storage stack. The generation field of the ObjectPointer will be very useful here as it shows how old a DML object is so that the change set can be very efficiently determined.

Scrub The implementation contains an unfinished implementation for scrubbing data as now, we can only scrub data of an DML object. The Tree Layer needs a method for walking a B^ε-Tree with the possibility of pruning subtrees. On top of this interface, the Database Layer could walk trees that correspond to data sets and snapshots and, hence, scrub their data.

Pool Expansion For now, the storage pool of a Database object is fixed after creation and cannot be expanded. A trivial expansion which should be supported is to add another top-level vdev to the storage pool. No further actions are needed except adding it to the corresponding data structures in the Storage Pool Layer.

Partial Leaf Nodes The implementation has a high read amplification for small random reads as a whole leaf node has to be fetched on cache miss. By splitting leaf nodes into smaller parts and checksumming each of these, we could fetch only the relevant part of the leaf.

9.2. Integration into the HPC Environment

Many HPC applications dump checkpoints of the program state in regular intervals so that in case of an interruption the computation can be resumed without losing too much time. Today, this checkpoint dumping is typically implemented by writing to a filesystem so that each checkpoint is a separate file. With an integration of our storage stack, the application could instead write its data dump to a data set and then snapshot it. This has the advantage that the data is logically grouped together and checkpoints can be easily enumerated.

9.3. Support for Open-Channel SSDs

Normal SSDs include a *Flash Translation Layer* (FTL) which is an abstraction layer that provides a block device interface. The FTL abstracts from the complex internal geometry of the SSD and certain restrictions to write operations. It provides a logical block mapping and is responsible for wear leveling and garbage collection. With the FTL, an SSD can be used as a direct replacement for an HDD. However, the FTL comes at a cost of unpredictable overhead.

There is ongoing work to provide a Linux subsystem called *LightNVM* which makes use of *Open-Channel SSDs* which do not have an included FTL [BGB17]. Compared to the Linux storage stack, our storage stack already avoids the overhead of multiple mapping tables in different layers. With LightNVM, we could avoid another mapping table – namely the FTL. It would be interesting to see how this will affect the overall performance and especially latencies. First attempts with application-driven FTLs seem promising [Gon+16].

9.4. Extensions to the B$^\epsilon$-Tree

We also propose two possible research topics for the B$^\epsilon$-Tree itself due to the high potential in the underlying basic idea of B$^\epsilon$-Trees which is incorporating the possibility of delaying work until necessary into the design of a data structure.

9.4.1. Snapshotting Tree Parts

One aspect of the B$^\epsilon$-Tree that increases efficiency is that we use less indirections. For example, the value of a key-value pair is inlined into the tree instead of saving a pointer to the actual value in tree. This allows us to avoid another seek on disk. However, we are still using multiple B$^\epsilon$-Trees for multiple data sets as the snapshot granularity is a whole B$^\epsilon$-Tree for now due to the path-copying technique.

Hence, an interesting research subject is how we can snapshot a part of a B$^\epsilon$-Tree – for example given by a key range. Snapshotting the whole tree like before would be quite inefficient due to the fact that we now also keep a snapshot of every other part of the tree. One approach would be to snapshot the whole tree and then on every node death we efficiently decide whether the tree node actually contains data needed for this snapshot.

9.4.2. Combining Copy-On-Write with Extents

On the one hand, modern filesystems often use extents to reduce overhead. For example, ext4 allocates up to 128 MiB extents to large files instead of blocks with the default size of 4 KiB. Hence, we have a metadata efficiency increase by up to five orders of magnitude. Also, extents decrease the allocation overhead and result in less fragmentation with the delayed allocation technique. On the other hand, modern filesystems like ZFS use

copy-on-write to implement important features such as snapshots. The downsides of copy-on-write are increased fragmentation and allocation overhead as it needs to allocate on every write.

Combining these two concepts seems like an obvious choice for modern storage stacks but this is highly complicated. Btrfs, a modern filesystem with copy-on-write extents, has problems with excessive fragmentation on database workloads [Con17b]. The suggested solution is to turn off copy-on-write which is clearly not what we would want to do.

So, designing a storage stack with copy-on-write extents is a non-trivial task. The main design question for copy-on-write extents is what should happen when an extent is partially overwritten. There are two possible approaches:

Copy the extent On partial overwrite, we duplicate the affected extent and overwrite the corresponding part.

Split the extent into parts On partial overwrite, we split the extent into two or three separate extents where one extent exactly covers the overwritten range. Now, we essentially have a full overwrite.

The potential downsides of both approaches are either a huge space and write amplification or excessive fragmentation and management overhead. The key is to balance between these two approaches. If there is only a small amount of changes to an extent, the second approach is better. If there is enough data to justify the copy of the extent, the first approach is better.

Btrfs uses the second approach exclusively, but the user can activate a background garbage collection called *autodefrag* which detects when there are a large number of small writes to a file and will copy heavily fragmented file parts to new extents.

A possibly better solution is to delay small writes before touching the extents at all and only update an extent if there is a large amount of changes. This could be achieved with a special data structure derived from the B^{ε}-Tree. Large writes are written as extent and small writes are inlined into the tree. If an internal node contains too much data, it will flush its data to the children. At the leaf level, all extents are non-overlapping. If a leaf is too large, we can either write out leaf data as new extents or split the leaf.

The critical aspect is the flush condition as we need to balance between fragmentation and space amplification. If an internal node is not large by itself but it contains large extents, a flush is probably beneficial nonetheless.

This concept essentially uses the first approach but enough writes are delayed to justify the copy of the extent.

Bibliography

[Ahr] Matthew Ahrens. *ZFS RAIDZ stripe width, or: How I Learned to Stop Worrying and Love RAIDZ*. URL: https://www.delphix.com/blog/delphix-engineering/zfs-raidz-stripe-width-or-how-i-learned-stop-worrying-and-love-raidz (Retrieved Jan. 3, 2018).

[Anv07] H Peter Anvin. *The mathematics of RAID-6*. 2007. URL: https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf.

[Arg95] Lars Arge. "The Buffer Tree: A New Technique for Optimal I/O-Algorithms". In: *Algorithms and Data Structures, 4th International Workshop, WADS '95, Kingston, Ontario, Canada, August 16-18, 1995, Proceedings*. Ed. by Selim G. Akl et al. Vol. 955. Lecture Notes in Computer Science. Springer, 1995, pp. 334–345. URL: https://doi.org/10.1007/3-540-60220-8_74.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pp. 1116–1127. URL: http://doi.acm.org/10.1145/48529.48535.

[Bai+08] Lakshmi N. Bairavasundaram et al. "An Analysis of Data Corruption in the Storage Stack". In: *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*. Ed. by Mary Baker and Erik Riedel. USENIX, 2008, pp. 223–238. URL: http://www.usenix.org/events/fast08/tech/bairavasundaram.html.

[Ben+15] Michael A Bender et al. "An introduction to B^{ε}-Trees and write-optimization". In: (2015), p. 22. URL: https://www.usenix.org/system/files/login/articles/login_oct15_05_bender.pdf.

[BF03] Gerth Stølting Brodal and Rolf Fagerberg. "Lower bounds for external memory dictionaries". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 2003, pp. 546–554. URL: http://dl.acm.org/citation.cfm?id=644108.644201.

[BGB17] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. "LightNVM: The Linux Open-Channel SSD Subsystem". In: *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*. Ed. by Geoff Kuenning and Carl A. Waldspurger. USENIX Association, 2017, pp. 359–374. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.

[BM70] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indexes”. In: *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*. ACM, 1970, pp. 107–141.

[Bon+03] Jeff Bonwick et al. “The zettabyte file system”. In: *Proc. of the 2nd Usenix Conference on File and Storage Technologies*. Vol. 215. 2003.

[BS77] Rudolf Bayer and Mario Schkolnick. “Concurrency of Operations on B-Trees”. In: *Acta Inf.* 9 (1977), pp. 1–21. URL: https://doi.org/10.1007/BF00263762.

[Con+17] Alexander Conway et al. “File Systems Fated for Senescence? Nonsense, Says Science!” In: *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*. Ed. by Geoff Kuenning and Carl A. Waldspurger. USENIX Association, 2017, pp. 45–58. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway.

[Con16a] OpenZFS Wiki Contributors. *Performance tuning - OpenZFS*. 2016. URL: http://open-zfs.org/wiki/Performance_tuning#Database_workloads (Retrieved Jan. 3, 2018).

[Con16b] Rust Design Patterns Contributors. *Newtype Pattern*. 2016. URL: https://github.com/rust-unofficial/patterns/blob/master/patterns/newtype.md (Retrieved Jan. 3, 2018).

[Con17a] Bincode Contributors. *bincode: A binary encoder / decoder implementation in Rust*. 2017. URL: https://github.com/TyOverby/bincode (Retrieved Jan. 3, 2018).

[Con17b] Btrfs Wiki Contributors. *Gotchas - btrfs Wiki*. 2017. URL: https://btrfs.wiki.kernel.org/index.php/Gotchas#Fragmentation (Retrieved Jan. 3, 2018).

[Con17c] Wikipedia Contributors. *Festplattenlaufwerk*. 2017. URL: https://de.wikipedia.org/wiki/Festplattenlaufwerk (Retrieved Jan. 3, 2018).

[Con17d] Wikipedia Contributors. *Hard disk drive*. 2017. URL: https://en.wikipedia.org/wiki/Hard_disk_drive (Retrieved Jan. 3, 2018).

[Con17e] Wikipedia Contributors. *Hard disk drive performance characteristics*. 2017. URL: https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics (Retrieved Jan. 3, 2018).

[Con17f] Wikipedia Contributors. *LINPACK*. 2017. URL: https://de.wikipedia.org/wiki/LINPACK (Retrieved Jan. 3, 2018).

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[Cor16] Jonathan Corbet. *Fixing asynchronous I/O, again*. 2016. URL: https://lwn.net/Articles/671649/ (Retrieved Jan. 3, 2018).

[Cor68] Fernando J Corbato. *A paging experiment with the multics system*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[Dev17a] DAOS Developers. *DAOS: a Scale-out Key-Array Object Store*. 2017. URL: https://github.com/daos-stack/daos (Retrieved Jan. 3, 2018).

[Dev17b] Linux Kernel Developers. *Linux Device Mapper Thin Provisioning*. 2017. URL: https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt (Retrieved Jan. 3, 2018).

[Dil17] Andreas Dilger. "DAOS: Scale-out Object Storage for NVRAM". 2017. URL: http://materials.dagstuhl.de/files/17/17202/17202.AndreasDilger.Slides.pdf (Retrieved Jan. 3, 2018).

[Dim17] Serapheim Dimitropoulos. "Faster Allocation with the Log Spacemap". OpenZFS Developer Summit. 2017. URL: https://drive.google.com/file/d/0B5fzqkw_-diCeDVZSzl0VnR6Tzg/view (Retrieved Jan. 3, 2018).

[Dok14] Dennis van Dok. *LVM2 snapshot performance problems*. 2014. URL: https://www.nikhef.nl/~dennisvd/lvmcrap.html (Retrieved Jan. 3, 2018).

[Dri+86] James R. Driscoll et al. "Making data structures persistent". In: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*. ACM. 1986, pp. 109–121. URL: http://doi.acm.org/10.1145/12130.12142.

[Esm+12] John Esmet et al. "The TokuFS Streaming File System". In: *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012*. Ed. by Raju Rangaswami. USENIX Association, 2012. URL: https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/esmet.

[Fri+99] Matteo Frigo et al. "Cache-Oblivious Algorithms". In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 1999, pp. 285–298. URL: https://doi.org/10.1109/SFFCS.1999.814600.

[Gam+95] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[Gon+16] Javier González et al. "Application-driven flash translation layers on open-channel ssds". In: *Proceedings of the 7th Non Volatile Memory Workshop (NVMW)*. 2016, pp. 1–2.

[Gra11] Goetz Graefe. "Modern B-Tree Techniques". In: *Foundations and Trends in Databases* 3.4 (2011), pp. 203–402. URL: https://doi.org/10.1561/1900000028.

[Gro17] Andy Grover. *Stratis Software Design: Version 0.8.4**. 2017. URL: https://stratis-storage.github.io/StratisSoftwareDesign.pdf (Retrieved Jan. 3, 2018).

[Gun+08] Haryadi S. Gunawi et al. "EIO: Error Handling is Occasionally Correct". In: *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*. Ed. by Mary Baker and Erik Riedel. USENIX, 2008, pp. 207–222. URL: http://www.usenix.org/events/fast08/tech/gunawi.html.

[Ins13] InstLatX64. *Memory Latency for Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz*. 2013. URL: http://users.atw.hu/instlatx64/GenuineIntel00306C3_HaswellXeon_NewMemLat.txt (Retrieved Jan. 3, 2018).

[Jan+15a] William Jannen et al. "BetrFS: A Right-Optimized Write-Optimized File System". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. Ed. by Jiri Schindler and Erez Zadok. USENIX Association, 2015, pp. 301–315. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen.

[Jan+15b] William Jannen et al. "BetrFS: Write-Optimization in a Kernel File System". In: *TOS* 11.4 (2015), 18:1–18:29. URL: http://doi.acm.org/10.1145/2798729.

[Koz+14] Quincey Koziol et al. *High performance parallel I/O*. CRC Press, 2014.

[Lev10] Adam Leventhal. "Triple-parity RAID and beyond". In: *Commun. ACM* 53.1 (2010), pp. 58–63. URL: http://doi.acm.org/10.1145/1629175.1629194.

[Lof+16] Jay F. Lofstead et al. "DAOS and friends: a proposal for an exascale storage system". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. Ed. by John West and Cherri M. Pancake. IEEE Computer Society, 2016, pp. 585–596. URL: https://doi.org/10.1109/SC.2016.49.

[Lom+16] Johann Lombardi et al. "MS 54 Distributed Persistent Memory Class Storage Model: DAOS-M – High Level Design". Revision 2.3. 2016. URL: https://wiki.hpdd.intel.com/download/attachments/36966823/MS54_CORAL_NRE_DAOSM_HLD_v2.3_final.pdf (Retrieved Jan. 3, 2018).

[LZ13] Johann Lombardi and Liang Zhen. "DAOS Changes to Lustre". 2013. URL: http://opensfs.org/wp-content/uploads/2013/04/Lombardi_Liang_DAOS_LUG13.pdf (Retrieved Jan. 3, 2018).

[Mer87] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology - CRYPTO ’87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378. URL: https://doi.org/10.1007/3-540-48184-2_32.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache”. In: *Proceedings of the FAST ’03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. Ed. by Jeff Chase. USENIX, 2003. URL: http://www.usenix.org/events/fast03/tech/megiddo.html.

[ONe+96] Patrick E. O’Neil et al. “The Log-Structured Merge-Tree (LSM-Tree)”. In: *Acta Inf.* 33.4 (1996), pp. 351–385. URL: https://doi.org/10.1007/s002360050048.

[Pap+16] Anastasios Papagiannis et al. “Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store”. In: *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. Ed. by Ajay Gulati and Hakim Weatherspoon. USENIX Association, 2016, pp. 537–550. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis.

[Per17] Percona. *PerconaFT*. 2017. URL: https://github.com/percona/PerconaFT (Retrieved Jan. 3, 2018).

[Per18] Percona. *Percona TokuDB*. 2018. URL: https://www.percona.com/software/mysql-database/percona-tokudb (Retrieved Jan. 3, 2018).

[Pla97] James S. Plank. “A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems”. In: *Softw., Pract. Exper.* 27.9 (1997), pp. 995–1012. URL: https://doi.org/10.1002/(SICI)1097-024X(199709)27:9%3C995::AID-SPE111%3E3.0.CO;2-6.

[RG13] Kai Ren and Garth A. Gibson. “TABLEFS: Enhancing Metadata Efficiency in the Local File System”. In: *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. Ed. by Andrew Birrell and Emin Gün Sirer. USENIX Association, 2013, pp. 145–156. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren.

[Rod07] Ohad Rodeh. “B-trees, Shadowing, and Clones”. In: *2007 Linux Storage & Filesystem Workshop, LSF 2007, San Jose, CA, USA, February 12-13, 2007*. Ed. by Ric Wheeler. USENIX Association, 2007. URL: https://www.usenix.org/conference/2007-linux-storage-filesystem-workshop/b-trees-shadowing-and-clones.

[Sch06] Patrick Schmid. *15 Years Of Hard Drive History: Capacities Outran Performance*. 2006. URL: https://www.tomshardware.com/reviews/hdd-history,1368.html (Retrieved Jan. 3, 2018).

[Spe13] Intel® ARK (Product Specs). *Intel® Xeon® Processor E3-1225 v3 (8M Cache, 3.20 GHz) Product Specifications*. 2013. URL: https://ark.intel.com/en/products/75461/Intel-Xeon-Processor-E3-1225-v3-8M-Cache-3_20-GHz (Retrieved Jan. 3, 2018).

[SR12] Russell Sears and Raghu Ramakrishnan. "bLSM: a general purpose log structured merge tree". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan et al. ACM, 2012, pp. 217–228. URL: http://doi.acm.org/10.1145/2213836.2213862.

[Sta] Stalkr. *Golang data races to break memory safety*. URL: https://blog.stalkr.net/2015/04/golang-data-races-to-break-memory-safety.html (Retrieved Jan. 3, 2018).

[TOO16] LVM TOOLS. *man lvmthin(7) - LVM thin provisioning*. 2.02.168(2). Red Hat, Inc. Nov. 2016.

[VTZ17] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems". In: *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*. Ed. by Geoff Kuenning and Carl A. Waldspurger. USENIX Association, 2017, pp. 59–72. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

[Was13] Scott Wasson. *Intel's Core i7-4770K and 4950HQ 'Haswell' processors reviewed: Memory subsystem performance*. 2013. URL: https://techreport.com/review/24879/intel-core-i7-4770k-and-4950hq-haswell-processors-reviewed/8 (Retrieved Jan. 3, 2018).

[Yua+16] Jun Yuan et al. "Optimizing Every Operation in a Write-optimized File System". In: *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. Ed. by Angela Demke Brown and Florentina I. Popovici. USENIX Association, 2016, pp. 1–14. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan.

Appendices

A. Usage

In this chapter, we will provide a quick start for using Rust and look at two small examples regarding how to use the storage stack implementation with Rust and C.

A.1. Installing and Using Rust

Rust's official installer is *rustup*. Rust can be installed with the following shell command on Unix-like platforms:[1]

```
curl https://sh.rustup.rs -sSf | sh
```

Then, we want to switch the default Rust compiler to the nightly rust compiler as the implementation uses many unstable features which are only available in nightly compilers. To do so, execute the following command: `rustup default nightly`. After changing the working directory to the source directory of the implementation, we can now execute the following commands:

`cargo build` This command builds the project in debug mode. Cargo will automatically fetch and build all dependencies listed in the `Cargo.toml` file.

`cargo build --release` Similar to the preceding command but builds in release mode.

`cargo test` Builds and runs all tests of the project.

`cargo doc --open` Builds the web-based documentation and opens it in a web browser window afterwards.

A.2. Quick Start

Now, we will build two small example applications that use the storage stack – one using Rust, one using C.

A.2.1. Example with Rust

We will start with Rust. First, we use cargo to create a new application project: `cargo init --bin small_example`

[1] For other installation options, see `https://rustup.rs`.

```
[package]
name = "small_example"
version = "0.1.0"
authors = ["Felix Wiedemann <1wiedema@informatik.uni-hamburg.de>"]

[dependencies]
betree_storage_stack = { path = "../betree_storage_stack" }
error-chain = "0.11"
```

Figure A.1.: `Cargo.toml` of the quick start example

Then, we declare the dependencies in the `Cargo.toml` file of project as seen in Figure A.1. The path to the storage stack implementation probably has to be adjusted. The second dependency is a package for error handling and will be automatically fetched by cargo from the Rust package registry `https://crates.io`.

Now, we still need the actual code. For the `main.rs` file see Figure A.2. Finally, we can run the program by executing `cargo run`:

```
$ cargo run
[ ... cargo builds the program ... ]
Value of foo is [72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

A.2.2. Example with C

For the C version, we need to build the library first by executing `cargo build` in the source directory of the storage stack. This will build a static library and a dynamic library. Our C code is presented in Figure A.3. We can build it with `gcc` as following:

```
$ gcc -L betree_storage_stack/target/debug small_c_example.c \
      -l:libbetree_storage_stack.a -lpthread -ldl -o small_c_example
```

And now, we can execute the program:

```
$ ./small_c_example mirror ./small_example/target/debug/file{0,1}
The value of the key 'foo' is: Hello, World!
```

```rust
// Link to extern dependencies
extern crate betree_storage_stack;
#[macro_use]
extern crate error_chain;

// Import types
use betree_storage_stack::{Configuration, Database, Error};

// The storage pool configuration notation is similar to ZFS.
const CFG_STR: &str = "mirror ./file1 ./file2";

fn run() -> Result<(), Error> {
    let cfg = Configuration::parse_zfs_like(CFG_STR.split_whitespace())?;

    // Create a new database with a storage pool given by our configuration.
    let mut db = Database::create(&cfg)?;

    // Create a new data set with the given name.
    // Would fail if already existing.
    db.create_dataset("test-ds".as_bytes())?;

    // Open a data set given by the name.
    let ds = db.open_dataset("test-ds".as_bytes())?;

    // Insert/overwrite a new key-value pair.
    ds.insert(
        "foo".as_bytes(),
        &[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33],
    )?;

    // Retrieve a key-value pair.
    match ds.get("foo".as_bytes())? {
        Some(v) => println!("Value of foo is {:?}", &v[..]),
        None => println!("The key foo does not exist"),
    }

    db.close_dataset(ds)?;
    db.sync()?;
    Ok(())
}

// This macro from error-chain defines a main function
// which executes the run function and handles potential errors.
quick_main!(run);
```

Figure A.2.: `main.rs` of the quick start example

```c
#include<stdio.h>
#include "betree_storage_stack/bindings.h"

int handle_err(err_t *e) {
    puts("Failed due to:");
    betree_print_error(e);
    betree_free_err(e);
    return 1;
}

int main(int argc, const char *const *argv) {
    if(argc < 2) {
        puts("Usage: <PROG> <CFG STRING>");
        return 1;
    }
    argv += 1;
    argc -= 1;
    err_t *e = NULL;
    cfg_t *cfg;
    if((cfg = betree_parse_configuration(argv, argc, &e)) == NULL)
        return handle_err(e);
    db_t *db;
    if((db = betree_open_db(cfg, &e)) == NULL)
        return handle_err(e);
    betree_free_cfg(cfg);
    ds_t *ds;
    if((ds = betree_open_ds(db, "test-ds", 7, &e)) == NULL)
        return handle_err(e);
    byte_slice_t x;
    if(betree_dataset_get(ds, "foo", 3, &x, &e) == 0) {
        printf("The value of the key 'foo' is: %.*s\n", x.len, x.ptr);
        betree_free_byte_slice(&x);
    } else if (e == NULL)
        printf("There is no value for the key 'foo'\n");
    else
        return handle_err(e);
    if(betree_close_ds(db, ds, &e))
        return handle_err(e);
    betree_close_db(db);
    return 0;
}
```

Figure A.3.: C quick start example

List of Figures

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang M. Sc. Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift