Bachelor Thesis

Design and Implementation of an Object Store with Tiered Storage

Till Höppner
betree@tilpner.com

August 12, 2021

First Reviewer:
Jun.-Prof. Dr. Michael Kuhn

Second Reviewer:
Kira Duwe

Supervisor:
Jun.-Prof. Dr. Michael Kuhn

Abstract

The B^ε-tree storage stack is a library built upon the B^ε-tree data structure, which manages block devices directly to offer a key-value interface to embedding Rust and C applications. In order to exceed its size limitations, a higher-level object abstraction is built on top of the key-value store by means of splitting into fixed-size chunks. It offers a convenient and misuse-resistant interface, as well as support for user-provided metadata. In addition to a command-line management and analysis tool, the object store functionality is integrated into the JULEA storage framework as a new object backend.

Given the varying performance properties and costs of modern storage devices, an advanced system has the potential of partially realising the benefits of more expensive homogeneous storage by combining different types and utilising each to their strengths. Two approaches of combining different storage devices into a tiered storage system are explored and implemented. Under the assumption that the user is aware of an object's underlying structure, the selected approach allows for a user-specified prioritisation of object parts. The evaluation results suggest that correct prioritisation can lead to performance gains when the access patterns conform to the object prioritisation, and incurs negligible performance cost when operated with a homogeneous storage pool.

Contents

Chapter 1.

Introduction

Data storage remains an essential part of many computational tasks, including high-performance computing such as weather prediction [Smart et al., 2019], or web services with large amounts of user-provided content like social networks [Beaver et al., 2010].

Object stores have recently risen in popularity as a storage abstraction supporting the storage of large values under arbitrary names. Due to fewer interdependencies and relaxed guarantees relative to filesystems, object storage systems enjoy greater implementation flexibility, especially in the context of efficient distributed storage. The wide availability of cost-efficient public cloud offerings established the niche of object storage for large read-mostly data storage.

A promising approach to the challenges of modern storage technology has been explored with the B^ε-tree storage stack [Wiedemann, 2018], introducing a key-value store implemented in terms of a write-optimised data structure, the eponymous B^ε-tree. Although its key-value interface permits small values, it falls short in the presence of larger inputs. In order to support a wider variety of usecases, an extension of the B^ε-tree storage stack towards an object store is desirable, as user and application data is frequently of an unknown or unpredictable size.

The landscape of modern storage devices exhibits a vast contrast in throughput, access latencies, and cost efficiency: common hard-disk drives (HDDs) are appreciated for their traditionally low price-performance ratio, but are also characterised by slow read/write speeds and long access latencies when compared to modern solid-state disks (SSDs).

Some access patterns change over time, e.g. that of a picture being shared online in a social network, which is more likely to be accessed soon after the initial sharing by notified friends, than many years later. In an effort to minimise the picture retrieval times, the service provider might want to keep recently uploaded submissions on faster SSDs, while slowly moving over ageing entries to cheaper HDDs.

It has been a long-standing goal of data storage to utilise different storage properties of available media, and attain some of the benefits of faster storage devices at a fraction of the cost of a homogeneous deployment.

Objective

The objectives of this thesis are threefold:

The design and implementation of an object store interface for the B^ε-tree storage stack, that extends the scope of the key-value store to objects of arbitrary size, accompanied by a convenient

and misuse-resistant interface that facilitates a multitude of applications. Additionally, a mechanism for the utilisation of the different properties found in modern storage devices is desired, aggregating cost-efficient HDDs and more capable SSDs into a tiered storage system with the potential to outperform an HDD-only system at significantly lower cost than an SSD-only system. Thirdly, the implementation of an object storage backend for an existing storage framework (JULEA), to make the new object functionality immediately available to existing applications without requiring any integration effort.

Outline

Background introduces concepts necessary for the remainder of this thesis, such as different storage abstractions and distinct usages of dissimilar storage devices.

Related work provides a sampling of different storage organisation approaches, to illustrate the breadth of the design space, and assist in comparing this work.

Object store develops the requirements and design of an object store built on top of the B^ε-tree storage stack.

Tiered storage selects an approach to tiered storage, and designs an incremental storage tier selection system.

Database improvements details a variety of miscellaneous improvements to the B^ε-tree storage stack, which are not specific to either object- or tiered storage.

Applications lists and describes two integrations of the B^ε-tree storage stack library.

Evaluation briefly advocates the testing approach of the B^ε-tree storage stack, before measuring its performance in varied situations.

Summary condenses the primary results of this work and suggests an assortment of future improvements.

Chapter 2.

Background

In this chapter, the B^ϵ-tree storage stack and related concepts are introduced, to be built upon in the remainder of this thesis.

2.1. Storage abstractions

When storing data on a computer system, there are multiple storage abstractions available, each with different advantages and disadvantages (summarised in Table 2.1). Among them are:

Block storage Modern storage hardware is designed around the concept of block storage, allowing operating systems and applications to read from and write to numbered ranges of fixed-size blocks. The tasks of space accounting and user data organisation are left to the user of the block storage interface. A block device is generally managed by a single party, because different organisational strategies could lead to data corruption without shared cooperative space accounting.

Even though block storage is typically associated with physical hardware, it can also be used over a network with the network block device (NBD) or internet small computer systems interface (iSCSI) protocols [Becker, 1999], or as virtual block devices with e.g. software redundant array of independent disks (RAID) [Curry et al., 2010].

Even providing a comparatively simple block interface can bear surprising complications, if the underlying storage technology does not match the abstraction closely. For example, users are sometimes expected to cooperate with the storage device itself (physical or virtual) via TRIM (SATA)/UNMAP (SCSI) commands to communicate that a given block range is no longer in use. This is important for lower level abstractions like the flash translation layer of an SSD[1], which can drop the unused blocks from its internal mapping and reduce the amount of unnecessary page copies during subsequent program-erase cycles. Similarly, a virtual machine guest system can notify its thinly provisioned virtual block devices of unused blocks, so that the host system can reuse the freed space for other purposes.

Block devices are usually operated via the SATA and SCSI[2] protocols, although e.g. Linux-based operating systems expose them in terms of more familiar POSIX file operations to user applications.

[1] Though it can be beneficial for HDDs as well, when using shingled magnetic recording (SMR) [McMillen, 2020].
[2] Which includes serial attached SCSI (SAS)

Key-value storage Key-value storage differs from block storage in multiple ways, most importantly the indexing method via arbitrary byte sequences instead of block addresses and the relaxed constraints on value sizes: although the associated values are often still limited in size, and are fetched or set in their entirety, they no longer need to be exact multiples of any block size. The used keys can similarly be of varying sizes, though the key-value store may impose maximum length limits and character set restrictions. The keys themselves are conceptually independent of each other, and do not need to be contiguous, but they are required to be unique across their namespace [Elmasri and Navathe, 2017].

In contrast to block storage, space allocation and management is handled by the key-value store, and the saved key-value associations can (in principle, not necessarily every implementation) be retrieved, changed, and deleted concurrently, without requiring the cooperation of every user that would be necessary with block storage.

Filesystems Commonly available with every popular desktop operating system, filesystems offer a hierarchical organisational structure for arbitrarily-sized files. This hierarchy presupposes that keys can no longer be chosen freely, but come with additional constraints: file keys (commonly referred to as paths) are constructed from multiple segments, joined by path separators, and preceding path segments must already have been created before invoking certain operations [Pate, 2003].

Files themselves are usually addressable with byte granularity, and can themselves have arbitrary sizes in byte increments up to the filesystem's capacity or active quota.

Although individual key-value stores may have a custom permission system, access control is an integral part of popular file systems, integrated with the operating system user registry, and capable of hierarchical per-file restrictions on the actions other users are allowed to perform. In addition to a set of fixed per-file attributes, many common filesystems offer the association of user-supplied key-value pairs with files, to store application-defined metadata.

Object stores Object storage is characterised by variable-sized named data associations called *objects*, which are often accompanied by per-object configurable metadata. These objects are indexed similarly to key-value stores and file systems, but without any hierarchical constraints. Object stores are often used to store large amounts of binary user data, but also find use in high-performance computing (HPC) environments [Smart et al., 2017].

Similarly to filesystems, object stores permit arbitrarily-sized data with arbitrary (unaligned) access offsets and lengths, and manage the available space internally [Factor et al., 2005]. Although the set of operations overlaps between different implementations, the actual interface is specific to each object store, and the feature set and specific consistency guarantees may differ. Even though object stores do not account for local system users, particularly the object stores with network interfaces have fine-grained access policy systems in place, defining the set of users who are permitted to perform an action on a subset of objects [Backes et al., 2018].

In addition, object stores sometimes feature namespaces or *buckets*, which provide a means of grouping objects and providing coarse-grained access control [Backes et al., 2018]. Within such a namespace, each object is independent from other objects, unlike the filesystem abstraction, which encodes file relations hierarchically. This comparative lack of interdependence simplifies

Abstraction	Indexed by	Value size	Access granularity	Metadata
Block	Logical block address	Block size (e.g. 512 B)	Multiples of block size	None
Key-value	Arbitrary byte string	Variable, often limited to 4 MiB or less	Entire value	None
File	Hierarchical path	Variable	Byte	POSIX attributes and xattrs
Object	Arbitrary byte string	Variable	Byte	Custom metadata

Table 2.1.: Comparison of the discussed storage abstractions.

distributed storage because operations on one object can be fulfilled independently of concurrent or past operations on other objects.

2.2. Heterogeneous storage

Many storage systems are capable of utilising the diverse properties of current storage devices, and presenting them to the user with a unified interface. Although the classification is not always explicit, similar storage devices are grouped together into **storage classes**. This stratification into storage classes could originate from significant differences in throughput or latency, like with modern SSDs and HDDs, but a storage system can also optimise for other metrics like durability or cost [Tang et al., 2016].

Due to the favourable capacity-cost ratio of HDDs, they often serve as one of the lower tiers, with an upper tier comprised of faster flash-based storage [Lüttgau et al., 2018].

An important aspect regarding redundancy, capacity, and performance is whether data exists only on a single class, or on multiple, and whether it is moved or copied onto different storage classes. This represents the difference between tiered storage and caching:

- **Distributed (Move)**: Data spans multiple classes, but individual parts are not redundantly stored on multiple classes.
- **Replicated (Copy)**: Data is fully present on a single class, with full or partial copies on another class.

Although these terms are not used consistently across publications and software documentation, **tiered storage** refers to the usage of multiple storage classes without data duplication in this thesis, and storage classes used for tiered storage are **storage tiers**. The distinction is important when considering redundancy: if a device class is only used to redundantly store data to accelerate future access, its failure will not cause the loss of data, whereas a storage class containing the only copy of a datum is not expendable, and its failure will result in lost data. Thus, there is more incentive to use redundancy aggregations with tiered storage.

In addition, the method of class assignment can be automatic or user/client-initiated:

- **Dynamic**: Data is automatically re-assigned to another class at runtime, e.g. due to access patterns or its age.
- **Static**: Data is assigned a class during creation, and does not change class unless instructed to.

When categorised by these two criteria, the following four usage types of heterogeneous storage arise:

Dynamically assigned (re)distribution Perhaps the most complex of all four cases, systems performing dynamic distribution do not only move data between storage tiers, but also decide which data to move on their own, e.g. according to previously defined policies or observed access patterns [Zhang et al., 2010]. This process of automating data migration in a tiered storage system is also called information lifecycle management (ILM) [Alatorre et al., 2014].

Write-caching in write-back mode (but not write-through caching) can also be considered dynamic redistribution, because writes are temporarily only present on the cache class before being written to the underlying primary data class. Contrarily, write-through caching ensures written data is written to both classes before completing a write operation, and would thus not be considered dynamic redistribution.

Statically assigned distribution With static distribution, the data is moved across storage classes only at the user's or client application's instruction, not automatically. The system is not limited to moving entire objects/files to another class, it could also partially move a datum and keep the rest in its original position.

Dynamically assigned replication Dynamic replication of data from a slower tier onto a faster tier represents a typical read-caching usecase, where data is copied onto a faster cache device to speed up future read queries. This is implemented in e.g. the Linux virtual block layer cache *bcache*[3].

Statically assigned replication For example, the live replication of an SSD onto a RAID mirror of two HDDs does not increase read or write performance of the live system, but it does provide some measure of data loss resistance. By a wider definition, many local and remote backup processes fall into this category.

2.3. B$^\varepsilon$-tree storage stack

The B$^\varepsilon$-tree storage stack is a library written in Rust which primarily provides a key-value storage interface by managing on-disk B$^\varepsilon$-trees [Wiedemann, 2018], available under either the the MIT or Apache 2 license. For this purpose, it implements multiple layers often provided by the operating system in an effort to improve performance by tailoring each component specifically to the needs of the whole system. This includes a storage layer to manage raw block devices and construct higher-order devices like mirrors and parity aggregations, and its own cache to complement the operating systems page cache[4].

[3] https://bcache.evilpiepirate.org/
[4] Which is disabled in Section 6.2.

2.3.1. B$^\varepsilon$-tree

The B-tree data structure is a generalisation of balanced binary search trees (BST), which preserves the ordering of its nodes and the logarithmic query complexity, but increases the logarithmic base by adapting the amount of child nodes (fanout) [Comer, 1979]. This makes it more suitable as a data structure for usage on HDDs, where the access time is not only determined by the total amount of transferred data, but also the amount of disk seeks.

A B$^\varepsilon$-tree is a tree data structure related to B-trees, which has been further generalised to amortise the cost of write operations, by buffering changes in internal tree nodes and only applying them after sufficient changes have accumulated, thus batching the costly read-modify-write cycles commonly associated with on-disk B-tree insertion [Bender et al., 2015]. As a result of this design goal, it is also classified as a write-optimised data structure (WOD).

This is accomplished by dedicating a portion of the space available in internal nodes to holding messages for its child nodes. These messages encode an action to be taken upon flushing to the target node, and can sometimes replace a costly read-modify-write cycle with a single message insertion. The size distribution of an internal node is parameterised by the B and $\varepsilon \in (0, 1)$ parameters. Although other descriptions use *items*/*key-value pairs* as the unit of B [Bender et al., 2015], the B$^\varepsilon$-tree storage stack defined B to be the used block size, and it is thus measured in bytes [Wiedemann, 2018]. With B being the size of the entire node, the B$^\varepsilon$-tree storage stack designates a space usage of B^ε for the pivots of internal nodes, and the remaining $B - B^\varepsilon$ for the purpose of buffering messages. Due to variable message sizes, these byte limits do not translate directly into item counts. For more details on this distinction, refer to [Bender et al., 2015] and the original thesis on the B$^\varepsilon$-tree storage stack [Wiedemann, 2018].

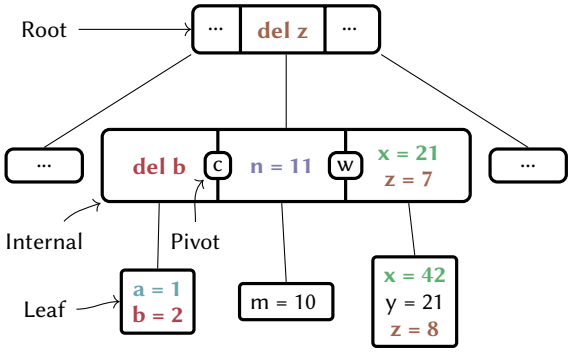
As shown in Figure 2.1 (left side), a B$^\varepsilon$-tree node can be either an internal node, or a leaf node. An internal node with n children contains n child buffers, and $n - 1$ pivot elements which separate the key spaces of each child. Messages intended for the key range of a child node are buffered in the respective child buffer, which also holds the reference to the child node itself. Leaf nodes store a sorted collection of key-value pairs, implemented internally with an in-memory B-tree in the B$^\varepsilon$-tree storage stack, but this is an implementation detail and the operations of a B$^\varepsilon$-tree do not rely on a particular leaf node representation.

The message type used with a B$^\varepsilon$-tree is configurable, and the set of operations which can be described by a single message determines the tree operations which can be buffered. As an example, the following three operations could be buffered given an appropriate message type:

- **Insertion of a new entry**, by encoding a message that will insert the new entry.
- **Modification of an existing entry**, by encoding a message that will apply some change to an entry. This could be a complete or partial replacement, or a type-aware modification like the incrementation of a stored integer.
- **Deletion of an existing entry**, by encoding a message that will delete that entry.

In the case of the B$^\varepsilon$-tree storage stack, the implementation diverges from the standard description of a B$^\varepsilon$-tree: internal nodes only need to hold up to a single message for each key, by adding the requirement that messages must support merges with another message and yield a new message describing the combined effect of both original messages in the correct order.

This restriction simplifies the management of messages, and permits advanced message types to reduce their size during merges if one submessage cancels the effects of another.



Key	a	b	n	x	z
Root					del z
Internal		del b	n = 11	x = 21	z = 7
Leaves	a = 1	b = 2		x = 42	z = 8
Result	1	del	11	21	del

Figure 2.1.: Exemplified point queries (right) into a B^ε-tree with two internal levels (left). The messages buffered in each segment of an internal node are intended for the key range of its corresponding child node.

When querying for the current value of a key, the tree is traversed by starting at the root and then repeatedly selecting the child node with a key range matching the queried key, until a leaf node is encountered. During this traversal, all encountered messages intended for the queried key are accumulated, and replayed in correct order on top of any potential value found in the leaf node, to present the user a view where all messages are immediately applied.

Figure 2.1 demonstrates the buffering of modifying actions with a simplified message type, which only permits deletions in the form of **del x**, and replacements like **x = 9**. It displays a subtree of an example B^ε-tree, and lists the relevant messages and results of a selection of point queries.

- **a**: This represents the simplest case, point queries occur similarly to non-buffering structures, as there are no buffered messages for **a**, and the final value of 1 is found in the leaf node.
- **n**: The insertion message for **n** (**n = 11**) has not been flushed all the way to a leaf node yet, but the message restores the value during playback, so the final value is the same as if it had been fully flushed: 11.
- **x**: Although a value of 42 is retrieved from the leaf node, the internal node contains a replacement message **x = 21**. When playing back that message, the intermediate value of 42 is replaced with 21, which is returned as the final value.
- **b**: This entry has been deleted, but the deletion message has not been fully flushed yet, so the leaf still contains the old value. The application of **del b** will delete the intermediate 2 found in the leaf node, and report to the caller that no entry for **b** exists.
- **z**: Neither of the previous values (**z = 7** and **z = 8**) affect the final result, as the deletion message **del z** in the root node deletes any intermediate value, in order to correctly indicate the non-existence of **z** to the caller. The deletion message **del z** is the newest of the displayed messages, and its insertion is unlikely to require any disk activity as the root node is usually already cached.

With these simplified examples, the point query algorithm could be aborted upon finding the first message, because it always replaces the previous value upon application in this example. However, other message types can be used with the B$^\varepsilon$-tree storage stack, which might not replace the value but instead make smaller changes to it (e.g. the integer addition mentioned earlier).

2.3.2. Architecture

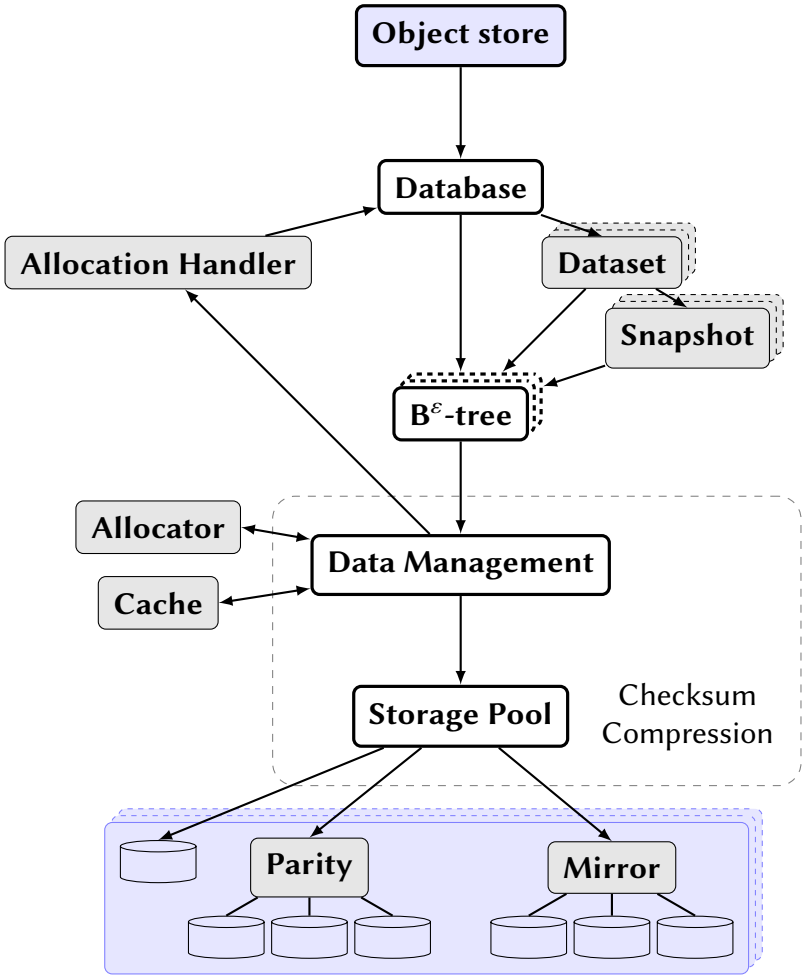


Figure 2.2.: Conceptual overview of the B$^\varepsilon$-tree storage stack architecture, reconstructed from a figure in [Wiedemann, 2018]. The object storage and tiered storage extensions of this thesis are highlighted in blue.

The B$^\varepsilon$-tree storage stack is constructed from several components, as depicted in Figure 2.2, with most of them implementing an interface to allow interoperation of different implementations of each role [Wiedemann, 2018]. The database layer manages datasets and snapshots, wraps them with a user-friendly API, and ties together the other components into a functioning system, including e.g. the database configuration and initial reading of the database superblocks. Operations on a B$^\varepsilon$-tree interact with the data management layer to handle tree manipulation and reshaping as necessary, during which it can simply request database objects from the data management layer. Indeed, the data management layer transparently serves mostly opaque

objects to the upper layer[5], without any knowledge of the tree nodes contained within. It ties into the cache system to minimise disk accesses, but will query the storage layer on cache misses. In the other direction, it is capable of cycling allocation segments, requesting block ranges from those segments, and then instructing the storage layer to perform the actual write after informing the allocation handler to persist the allocation in the allocation bitmap.

Storage system The storage system is built around modular virtual devices also called **vdevs**, which can be organised in a tree-like fashion. A leaf vdev can be an operating system file, which includes raw block devices like disk partitions. These leaf vdevs can then be aggregated into one of two different new vdev types:

Mirror A mirror vdev writes the received data to all its child vdevs, and can sustain the failure of all but the last vdev. Although it is limited in capacity and write speed to the smallest and slowest device respectively, the mirror can distribute reads across its child vdevs to surpass the read transfer speeds of a single device.

Parity The parity vdev distributes its writes to its children, and generates parity data in order to sustain the failure of any single child vdev. Unlike the mirror vdev, which could choose to read only from a single disk, the parity vdev has to read from multiple of its child vdevs to reconstruct the requested data, similarly allowing for higher read throughput than a single disk.

The storage system additionally maintains a job queue and thread pool, onto which it distributes incoming write and read requests. In cooperation with the data management layer, which generates the initial checksums when writing data, each vdev is capable of verifying its own data against a checksum while reading.

Key-value interface The database layer uses the dataset abstraction to support the use of multiple B^{ε}-trees. Each dataset can be named with an arbitrary byte string, and later re-opened with that same name.

Datasets are parameterised by the used message type, and the available operations differ if the user instantiates a dataset without the default message type. Once a dataset has been created and opened, the following key-value operations are available for all datasets:

insert_msg Insert an already constructed message into the tree for a given key.

get Retrieve the value for a given key, while applying all encountered messages in the correct order. This operation is also called a point query.

range Retrieve a range of key-value pairs for a specified key range.

Additionally, datasets using the default message type support the following convenience functionality, which are all wrapping the previous three functions and generating the message automatically:

insert Inserts an insertion message, which will overwrite the value for a given key entirely with a specified byte sequence, or create a new entry if none existed.

[5] Although it is aware of copy-on-write semantics

upsert An upsert message replaces the value of an entry only partially, substituting a specified range of an entry with a new byte sequence.

delete Inserts a deletion message for a given key.

range_delete Efficiently deletes entire tree nodes instead of inserting individual deletion messages. Unfortunately, the corresponding tree operation is not fully implemented, and does not perform the necessary rebalancing/merging after removing entire subtrees within a key range.

2.3.3. Implementation language

The B^{ε}-tree storage stack is written in Rust, which is a comparatively new programming language released in 2015[6], and intermittently developed at Mozilla Research for the purpose of furnishing a safe and performant language suited for the complexities of web browser development, particularly if concurrency is involved [Rust team, 2016]. It features strong static typing with local type inference, a statically-verified deterministic resource management concept designed around value ownership and borrowing, as well as compiler-assisted reasoning about thread safety [Klabnik and Nichols, 2019, ch. 16.04/p. 368-369], and is commonly compiled via the LLVM[7] compilation infrastructure.

Due to the deterministic control over resource usage while retaining memory safety by default[8], Rust is well suited to performance-sensitive low-level abstractions such as operating system kernels, or storage systems. Indeed, Rust is not an uncommon choice for the implementation of a storage system, as there are multiple local storage systems with a key-value interface [Neely, 2017, Meunier, 2016, Persy developers, 2017], a distributed key-value store, which serves as the backend for multiple databases built on top (TiDB[9], Zetta[10], and Tidis[11]), and it is even used in Amazon's S3 cloud object storage system [Asay, 2021], albeit in an unspecified capacity.

2.4. JULEA

JULEA is a set of libraries created initially by Michael Kuhn, offering abstraction layers for databases, key-value storage, and object stores. Applications can use it to translate higher-level storage operations into calls to a variety of backends, without being aware of the configured backend, and whether the backend is local or remote [Kuhn, 2017]. It is operable in a client-server configuration, where the application may run on a different host than the storage provider, or can alternatively be used entirely within the application process.

Because the backends conform to a common interface, they can be treated as interchangeable from the application's perspective, decoupling its development process from any specific storage implementation. Although JULEA also provides abstractions for key-value and database

[6] Version 1.0, previous versions have been available since 2010
[7] https://llvm.org/
[8] With the option of bypassing compiler checks if necessary
[9] https://github.com/pingcap/tidb
[10] https://github.com/zhihu/zetta
[11] https://github.com/yongman/tidis

operations, only the object storage abstraction will be considered here. At the time of writing, JULEA offers three functional object storage backends, implemented in terms of POSIX-standard functions, the Gnome Input/Output library *GIO*, and with *librados* from the Ceph project.

An object backend is expected to implement the following operations:

create Creates a new object

open Opens an existing object

delete Deletes an existing object

close Closes a previously opened object

status Queries the size and modification time of an object

sync Ensures any outstanding operations for an object have been completed

read Reads a specified object range

write Writes data into a specified object range

Chapter 3.

Related work

This chapter supplements the previously introduced theoretical concepts of object stores and tiered storage by describing concrete implementations. A few of their design decisions will later be compared to decisions made during extensions of the B^ε-tree storage stack.

3.1. Object storage

Many related storage solutions are of a distributed nature, taking advantage of the reduced interdependencies of key-value and object storage when compared to filesystems, to horizontally scale across multiple servers. Although the B^ε-tree storage stack could be used as a component in a similar distributed system, this section is mostly concerned with the data organisation of each related work, and omits their distributed aspects. Nevertheless, it should be noted that the additional constraints of a distributed system result in different design restrictions than those of a local storage system, and a direct comparison is not always useful.

MongoDB/GridFS The document-oriented NoSQL database MongoDB enforces a size limit of 16 MB per stored document, and specifies an abstraction on top of the document interface to circumvent this size restriction: GridFS. Larger files are split into multiple chunks of a limited size by the database driver, and additional metadata such as the length, creation time, and a checksum are stored in a separate document [Chodorow, 2013]. After an object has been stored, the driver transparently maps read requests onto the previously generated chunks. Updates to individual chunks after the initial creation of an object are not supported [Banker et al., 2016].

Ambry The decentralised object store Ambry, developed by the social networking company LinkedIn to store immutable objects, chooses a different approach: objects are stored in fixed-size files (100 GB) called partitions, and changes like additions or deletions are appended to the respective end of the partition. Space of deleted objects is only freed during a periodic compaction process. Objects are split into chunks with a size of 4-8MB to resolve performance issues, despite not being built on top of a key-value store. Instead of keeping a separate cache, Ambry relies on the operating system's page cache to serve reads of recent data [Noghabi et al., 2016].

Atlas The search engine company Baidu introduced the scalable key-value store Atlas in 2015, which manages to retain all metadata in main memory despite being deployed on low-powered ARM servers with only 4 GB of memory per 16 TB of persistent storage. Atlas imposes a maximum size of 256 kB on values, and identifies them by a truncated SHA1 hash[1], thus providing entry deduplication at no extra cost, and preventing future modifications. Internally, the keys and metadata are stored in a log-structured merge tree (LSM tree), while the values are allocated in patches. After reaching their capacity, each 64 MB patch is split into 8 MB blocks, erasure-coded for redundancy, and distributed to available block servers [Lai et al., 2015].

SeaweedFS SeaweedFS[2] is a distributed object storage system comparable to Ceph in function, with support for tiered storage. Objects are stored in volumes, which are large[3] files stored on ordinary host filesystems. Although SeaweedFS only manages two storage classes by default, it supports arbitrary tagging of volumes, so that the use of more than two storage classes is possible [Lu, 2021].

The documentation suggests that allocation preferences are not specified individually during an object's creation, but that groups of volumes called collections and key prefixes can be designated for allocation within volumes of a particular class. Class reassignment is supported only at a by-volume granularity, by moving the large volume files onto another filesystem.

Ceph Ceph represents one of the most well-known distributed storage solutions available, offering not only an object abstraction, but also scalable block- and file storage. It is built around an array of object storage daemons (OSDs), which serve as the backing storage for higher-level distributed features. In 2017, the project switched the default storage backend from FileStore, which stored data on existing host filesystems such as XFS, to BlueStore, which directly manages block devices with direct IO.

BlueStore allocates block extents for object data and maintains internal and per-object metadata in the embedded key-value store RocksDB, which stores its own data via a minimal filesystem translation layer purpose-built for RocksDB. By utilising the merge-operator functionality of RocksDB, BlueStore is able to record space allocations without incurring the costs of a read-modify-write cycle. Additionally, BlueStore is designed for cooperation with newer zoned storage interfaces, which more closely match the internal implementation of modern SSDs and SMR HDDs. It employs copy-on-write for large write operations, and batches small operations within RocksDB, with a threshold that takes into account the underlying storage hardware: 64 KiB on HDDs, and 16 KiB with SSDs [Aghayev et al., 2019]. There is limited support for tiered storage, in that the write-ahead log and internal metadata can be stored on separate storage devices [Ceph authors, 2017].

[1] From 20 to 16 bytes
[2] https://github.com/chrislusf/seaweedfs, Apache License 2.0
[3] 30 GB by default

3.2. Tiered storage

As presented in Section 2.2, tiered storage can be further classified by the initiator of migration, i.e. whether inter-tier migration is induced by a dynamic policy, or statically assigned by the user/application.

Dynamically assigned tiered storage Dynamic tiered storage systems automatically migrate data between storage classes according to preconfigured policies. Even though they may not be entirely separate concerns, or documented as such, a dynamic system could be divided into a policy engine and a storage system. With this separation, the storage system implements static tiered storage by definition, as the dynamic decision process belongs to the policy engine, at least conceptually.

Such policies could range from simple age-based methods, over more elaborate classification via path prefixes, to complicated policies involving access pattern predictions. The potential cost savings of automatic tiered storage already motivated research into migration policies in 1990, when the lower storage tiers consisted of optical media and tape libraries instead of HDDs [Klastorin et al., 1993]. Multiple research papers have focused primarily on the design and tuning of such policies: [Moinzadeh and Berk, 2000] proposes and analyses a dynamic archival policy on a purely theoretical level, optimising for average read times under the assumption that access frequencies decrease as data ages. Similarly, a variety of upgrade and downgrade policies are explored in [Herodotou and Kakoulli, 2019], within the context of tiered distributed file storage for cluster computing.

Composition of single-tiered systems Tiered storage does not necessarily require a storage provider aware of multiple storage classes, or that each storage class is part of the same system. For example, a periodically executed script which moves old files from a fast filesystem to a slower filesystem and creates symbolic links in their place would constitute a simple tiered storage system[4], despite neither filesystem necessarily spanning multiple storage devices.

In GRANDET, this approach is applied to a selection of Amazon Web Services' (AWS) storage options, each unaware of its participation in a tiered storage system, yet presented in a unified manner to applications via an object and filesystem proxy component [Tang et al., 2016]. Depending on the desired durability, availability, the predicted access pattern, and other requirements, each object is placed on the most economical option among the suitable storage types, which could be an ephemeral local disk, shared block storage, or AWS' own object storage solution. The decision is not only influenced by a set of optional requirements passed during object creation, but also by access patterns of similar objects, where similarity is determined via an optional object tagging system, Additionally, GRANDET can automatically migrate objects to another storage type when access patterns or pricing models change.

This approach of integrating existing storage systems into a storage hierarchy obviates the need of building a custom system aware of tiered storage, but migrations between tiers might be more efficient if performed within a single system.

[4] To provide a consistent unified interface, a few scripts for listing and deletion of files/objects would be necessary, which would e.g. also delete the referenced file when deleting a symbolic link.

Chapter 4.

Object store

Throughout this chapter, the B^{ε}-tree storage stack is extended to support the storage of arbitrarily-sized objects, by building on top of the existing key-value interface. The tradeoffs of necessary design considerations involved in key-value organisation and user-interface design are contrasted, and the final design illustrated.

4.1. Requirements

Initially, the functionality of an object store may not appear to differ significantly from that provided by the already existing key-value interface described in Section 2.3.2. Both modes address a non-hierarchical collection of data by a user-provided arbitrary byte key.

Although the keys and values of both modes are variable in length, the key-value store is only equipped to handle comparatively small entries, and has no support for partial rewrites of a value. In contrast, an object store is expected to store potentially much larger values, up to and exceeding multiple GiB. These sizes require operations to selectively only operate on a subsection of the values, as loading the entire value into memory would be prohibitive.

These requirements resemble traditional file systems in that both provide keyed dynamically-sized data storage with large upper limits on file/object size, but whereas file systems often organise data in a hierarchical fashion with nested directories, an object store is characterised by flat namespaces, without deriving any inter-object relations from the object keys.

4.1.1. Desired operations

A few essential operations are required for basic manipulation of objects:

Create
: A new object is created. Attempting to re-create an already existing object should result in an error.

Read
: A range of an existing object's data is read and returned to the caller.

Write
: An open object is partially or entirely (re-)written.

Delete
: An existing object is deleted. Attempting to delete a non-existing object should result in an error.

If the interface is built around stateful operations, e.g. with object handles or due to an externally needed API, two additional operations become necessary to manage the lifecycle of the per-object-handle state:

Open
: An existing object is re-opened and a handle to the opened object is returned. Attempting to open a non-existing object must result in an error.

Close
: The handle to an existing object is closed. The system may defer certain outstanding operations until handle closure, depending on the targeted consistency and recovery guarantees.

Depending on the usecase, the following additional operations may provide convenience to the user:

Flush/Sync
: Wait for outstanding operations to complete, either for the whole object store, or for a single object.

Listing
: Query the object store for a list of contained objects. The listing may be exhaustive, but could also support filtering the key by prefix or range, as well as more advanced filtering over other metadata.

Rename
: Replace the key of an object. This can be done without special support by using the essential operations to manually copy the source object to a new object, then deleting the old one, but a less expensive implementation would be preferable.

Copy
: Create an identical second object, which is independent of the source object after completion of the copy operation. This may be implemented by referencing unchanged chunks of the old object, to reduce space usage if the copy diverges only partially from the source.

Link/Unlink
: Create/Remove an alias for an existing object. The exact semantics are left unspecified, most notably whether the target is referenced by name or object ID. This determines whether the data readable from a link is preserved after a subsequent rename/deletion of the target object. This would likely require either a way to differentiate between the link and a full object, or object reference counting.

Status
: Object metadata is retrieved, including e.g. size and modification time.

Considering the intended integration into JULEA, all operations required for the implementation of a JULEA object backend need be implemented. In addition to the essential operations, such an object backend requires **Flush**, and **Status**. After pointing out that cheap object listing can be implemented in the B^{ε}-tree storage stack, an operation for the listing of all objects,

optionally filtering by prefix, was added to the set of required operations for JULEA object backends.

4.1.2. Metadata

Metadata is information pertaining to other information, and different kinds of metadata can be associated with individual objects. Certain metadata attributes are implementation details, not meant to be exposed to the user. The following exposed kinds of metadata are considered:

Object size
: Logical size of the object in bytes, updated on every write operation. The size of an object is defined as the largest offset affected by any modification during the object's lifetime.

Modification time
: Timestamp of most recent write operation to an object, updated even when the object data is not altered by the operation, e.g. with a write size of 0, or when the data being written was already present at the target destination.

Arbitrary user-provided key-value pairs
: The user may register additional key-value pairs of per-object metadata, e.g. HTTP headers to serve an object with, an identifier of the original author, or a short summary to be displayed while generating a listing of many large objects.

Object size and modification time differ from custom metadata, in that they are automatically managed and assigned meaning by the object store. In contrast, user-provided custom metadata is only altered at the user's request, and their values are never interpreted. Examples of custom user metadata usage could be the expiry time of an object, the logical owner, or other forms of access control metadata.

Because these two managed attributes are always present for every object, they can be queried with the **Status** operation, whereas a few more operations are required to enable editing and reading of custom attributes:

Metadata List
: Gather all custom attributes of a specific object. Due to the lower expected size of metadata, filtering mechanisms are of a lower priority here.

Metadata Get/Set
: Get or set a specific attribute of an object by name. If attempting to set an existing attribute, the previous value is replaced.

Metadata Delete
: Remove a specific attribute of an object by name. Attempts to remove a non-existent attribute are ignored.

4.2. Data organisation

This section defines the key-value layout of object data, and specifies the behaviour of discontinuous objects.

4.2.1. Object chunking

The key-value interface of the B^ε-tree storage stack, as described in Section 2.3.2, already supports the creation and deletion of key-value associations, where each key and value can be a user-defined byte sequence. Additionally, the values of stored associations can be queried and rewritten. While this is remarkably close to the essential operations defined above, there are a few important distinctions:

Size limit While the key-value interface originally did not check the size of keys and values, the tree layer is unable to handle arbitrarily large byte sequences. As determined by own experiments, attempts to insert values in excess of the maximum node size can result in a failed split of the node in question (as a split requires multiple elements, and a very large insertion could trigger a split with just one element), from which the database is unable to recover[1].

Access granularity Key-value pairs can only be queried entirely. This is sensible for small values, where the entire containing node is already fetched and decoded, so that the cost of returning the entire value is the same as returning just a portion would be. In contrast, the user might want to, or even be required to, process a large object in smaller steps, requiring support for partial queries.

Both of these issues prevent the unmodified use of the key-value interface for objects, but they can be solved by the distribution of larger objects across multiple key-value pairs, henceforth called **chunks**. Each individual chunk's size can be kept below a chosen limit, scaling the amount of chunks with the object's size instead of scaling the size of a singular chunk. Partial queries can be implemented by querying only a subset of the underlying chunks, no longer requiring that the entire value be present before returning a portion of it to the user.

The size of each individual chunk determines not only the access granularity, but also influences how often a given operation will cross chunk boundaries and need to be split into sub-operations. A fixed-width splitting strategy was chosen due to the simplicity of chunk index calculations, but more advanced chunking strategies, e.g. with dynamic widths, might result in reduced I/O amplification when fixed-width chunks are too wide, or reduced CPU usage when fixed-width chunks would be too narrow. The concept of splitting objects for storage in a key-value store also finds a precedent in GridFS, which splits files over multiple entries in MongoDB, as previously described in Section 3.1.

A chunk size of 128 KiB is chosen as a compromise to support both random small operations and large sequential operations with acceptable I/O amplification and CPU overhead, and because it is the default record size of ZFS [OpenZFS project, 2020, Section "Dataset recordsize"]. Dynamic changes to the record size are not currently supported, but the chunk size is only determined by a single compile-time constant, so future changes to the chunk size remain feasible within key-value size limitations.

Internal chunk iteration An operation needs to be split into sub-operations whenever it operates on a user-specified portion of an object, potentially crossing chunk boundaries. To avoid reimplementation of the chunk-splitting logic for each operation, the `ChunkRange`

[1] A maximum value size of 512 KiB has since been added, and attempts to insert larger values will result in an error.

structure is introduced as an internal abstraction, consisting of the two extremes of the spanned range, to be used like shown in Listing 4.1. It can also be constructed from byte offsets, and allows for partial chunks at the beginning and end if the byte offsets are not aligned to the chunk width. This multi-chunk range can then be split into a variable amount of smaller `ChunkRange`s, each only spanning at most one full chunk, via an iterator returned by the `split_at_chunk_bounds` function. The first and last chunks may span less than one full chunk width, and the iterator may also not return any, or just one, chunks in case the range is empty or contained entirely inside one aligned chunk width. The `ChunkRange` API is intended to be used as shown in Listing 4.1, and finds use in the implementations of the **Read** and **Write** operations.

```
let chunk_range = ChunkRange::from_byte_bounds(offset, buf.len() as u64);
for chunk in chunk_range.split_at_chunk_bounds() {
  let len = chunk.single_chunk_len();
  let key = object_chunk_key(self.object_id, chunk.start.chunk_id);

  // now processed up until chunk.end.as_bytes()
}
```

Listing 4.1: Chunking API

4.2.2. Chunk indirection

```
- Foo|0 -> [... first 128KiB of object data ...]
- Foo|1 -> [...]
  ... additional chunks ...
- Foo|... -> [... last (up to) 128KiB of object data ...]
```

Listing 4.2: Direct data layout of key-value pairs to store the object 'Foo'.

A very simple data layout (example in Listing 4.2) could construct keys by concatenation of the object name and the chunk identifier[2]. This would redundantly store the object name once per chunk, require dynamic allocation during key construction on every chunk access, and directly tie the object data to its name. Renaming an object stored in such a fashion (if the name is embedded in the key) would necessitate either renaming keys without changing the values, or a *move* operation of each chunk to a new key, essentially copying the entire object's data. Additionally, obtaining a listing of all objects would have to be implemented by iterating all keys of the tree, which would result in the iteration of every chunk of every object, even though only the first chunk of each would be necessary for the listing. Reading the entire object store to list the contained objects is prohibitively expensive, and would prevent exposing this operation to the user.

A level of indirection, as shown in Listing 4.3, is used to decouple the object name from the constructed keys. By using a fixed-size object identifier in the place of a variable-length object name, and maintaining a mapping of names to object identifiers elsewhere, multiple problems of direct indexing are resolved:

[2]With a null byte as separator, because null bytes are not permitted inside object names

```
- Meta|Foo -> [ 42 ] <- object identifier of object 'Foo'
- Data|42|0 -> [... first 128KiB of object data ...]
- Data|42|1 -> [...]
  ... additional chunks ...
- Data|... -> [... last (up to) 128KiB of object data ...]
```

Listing 4.3: Data layout with indirection

1. Renaming becomes possible without copying the entire object, with a much cheaper change in the indirection mapping.
2. Object listing can be implemented cheaply by iterating the indirection mapping, iteration of data chunks is no longer necessary.
3. Fixed-size object identifiers are still redundant across all chunks of the object, but there no longer is any significant additional cost to using longer object names.
4. Keys constructed from a fixed-size object identifier and another fixed-size chunk identifier are themselves fixed in size, allowing construction without dynamic memory management.

4.2.3. Sparse object semantics

If the write operation allows arbitrary offsets, the user could pass an offset past the current object size, resulting in a sparse object if the operation does not return an error. In this context, any objects with non-contiguous chunk identifiers, or internal chunks smaller than the chunk size, are called **sparse objects** (as opposed to dense objects).

Rejecting writes past the current object size would require reading the size on every read, negating some of the advantage of using the write-optimised B^ε-tree data structure. The object size can not be cached in the object handle, as another client could concurrently open a handle to the same object, and alter the object size. To preserve a write-only **Write** operation, and to facilitate certain usecases where sparse objects could be useful (such as out-of-order creation of an object, which would create a temporarily sparse object and fill in the gaps e.g. in reverse), the write operation can not reject these offsets.

The possible existence of sparse objects prompts another decision: if a client writes to a new object with a large offset once, what data should be returned when reading that object? A few options are:

- Existing chunks could be concatenated during **Read**. Unfortunately, the variable amount and locations of gaps could not be communicated to the caller without significant divergence from the usual interface of **Read** functions. This behaviour would be unnecessarily unintuitive, and most certainly does not match the user's expectation.
- The gaps could be filled with zeroes, up until the object's size. This proposal matches the behaviour of POSIX-compliant filesystems, which similarly fill any gaps with zeroes while reading [POSIX, 2018]. Additionally, the default message type of the B^ε-tree storage stack already zero-fills values during application if an upsert would otherwise result in a sparse key-value pair.

- Expose sparse data as-is, with enough information for the client to implement any of the other options. This should not be the only option, as users would have to translate this information into the desired view even when working with dense objects.

To match the expectations formed by POSIX-compliant filesystems, **Read** presents a zero-filled view of an object, but an additional method exposes the object structure, to permit more efficient handling of sparse objects.

Each call to **Read** incurs the cost of one initial metadata query, as zero-filling should only occur up until the object's size. Concurrent changes to the object will only be observed within the object's original boundaries, i.e. size changes of an object will not take effect for ongoing read operations.

The current design does not expose an operation for the conversion of dense objects to sparse objects without recreation of that object, but such an operation could be easily added in the future.

4.3. Metadata organisation

To efficiently fulfil the previously established operations, object data and metadata needs to be inserted into the key-value database in an organised fashion. The choice of key distribution determines multiple important properties:

Data locality in nodes
: Due to the large node sizes of the B^ε-tree storage stack, if a key lookup to the key-value layer results in a new node being fetched, that node is likely to contain additional keys. It would be desirable for those additional keys to also be relevant to future key-value requests of the object store, reducing the number of node fetches necessary to fulfil the object store's operation. To improve the ratio of relevant to irrelevant information in a given fetched node, measured with respect to the current operation, data and metadata should be organised for locality, so that information is kept closely together in the tree ordering if it's likely to be needed during the same operation.

Data locality on disk
: As an extension of the previous locality property, proximity of related nodes on the block layer may result in reduced access times. These proximate nodes might have already been fetched by the read-ahead mechanism, and on a spinning storage medium it would likely be cheaper to fetch physically close nodes than it would be if they were scattered without such considerations of locality.

Space usage from keys
: Redundant information encoded into the constructed keys can lead to space efficiency overhead. For example, if the full object name is used in every chunk key, and that object name is particularly long, a 1 GiB object with a 256 B name would result in the redundant storage of 2 MiB of identical object names.

Ease of key construction
: The construction of dynamically-sized keys, e.g. because object names are incorporated into the final key in the form of user-provided and arbitrarily-sized byte strings, may require copying into a resizable buffer and cause dynamic allocations. While this effect

is perhaps the least important, the added memory and allocator pressure may negatively affect other parts of the system, and should be avoided where feasible.

This section gradually evolves the final key-value layout, and emphasises the reasons and tradeoffs of each step. The example listings contain abstract keys, their specific encoding into bytes is explained later in this section.

Although the translation mapping from object names to object identifiers already constitutes metadata, the previously shown layouts are missing a method of storing the metadata described in Section 4.1.2.

4.3.1. Fixed metadata

```
- Meta|Foo|object-id -> 42
- Meta|Foo|mtime -> 1619899200
- Meta|Foo|size -> 44040192
```

Listing 4.4: Separate fixed metadata

```
- Meta|Foo -> 42 | 1619899200 |
    ↪ 44040192
```

Listing 4.5: Merged fixed metadata

Like in Listing 4.4, each attribute could be stored separately, assigning one key-value pair per logical metadatum. This simple approach incurs the space overhead of assigning each of these three properties a unique key, and having to insert three messages into the B^{ε}-tree per object.

As the object identifier, size and modification time attributes are present for each object, they could alternatively be merged into a single key-value pair, as shown in Listing 4.5. This is only feasible because they are automatically managed by the object store, and a fixed-size and stable encoding format can be selected for each metadatum.

The space overhead of assigning each of these three properties a unique key and inserting three times as many key-value pairs into the B^{ε}-tree, as opposed to only one packed version, outweighs the simplicity of having three separate values, as each of the properties is only represented with eight bytes each. The modification granularity of the separate variant could e.g. be preserved with the upsert message feature, modifying only a fixed subrange of the metadata value. Without a mechanism for eager merging of messages, there is a risk of unbounded[3] accumulation of overlapping upsert messages, causing a reduction in space efficiency by storing outdated upserts, and diminishing read performance by applying upserts with no effect on the final return value[4]. The more compact merged metadata representation is chosen to minimise storage cost and reduce fixed overhead caused by the key-value operations, such as repeatedly locating the appropriate tree node.

[3]Restricted only by the maximum node size

[4]Both issues are addressed with a custom message type in Section 4.3.3.

4.3.2. Interleaved/Separate custom attributes

```
- Meta|Foo|object-id -> 42
- Meta|Foo|custom1 -> baz
- Meta|Foo|custom2 -> quux
- Meta|Bar|object-id -> 43
```

Listing 4.6: Interleaved custom metadata

```
- Meta|Foo|object-id -> 42
- Meta|Bar|object-id -> 43
- MetaCustom|42|custom1 -> baz
- MetaCustom|42|custom2 -> quux
```

Listing 4.7: Separate custom metadata

Packing the custom attributes into a single key, like proposed for the fixed attributes in Listing 4.5, is hindered by the dynamic size of each attribute. Such a packed format would require a way to quickly find a key-value pair by name, insert new associations, and perhaps even maintain an index of the starting positions and lengths of each packed attribute.

A higher density of object names reduces the number of nodes necessary for a full object iteration. When large or too many key-value metadata pairs are inserted, that density decreases and the iteration process is required to read and discard this custom metadata. In this respect, a separate layout like shown in Listing 4.7 would be favourable for the object iteration speed if custom metadata is not accessed, while interleaving the fixed and custom metadata like in Listing 4.6 would allow the inspection of an object's entire metadata during a single range query.

The interleaved layout is chosen to specifically support object scanning usecases which involve the inspection of custom metadata, as the relevant metadata will be already cached instead of needing to be fetched from a separate subtree. The reduced iteration speed due to lower density is not expected to be significant, if value sizes are kept reasonably low (e.g. storing cryptographic signatures instead of image thumbnails), but no benchmarking was performed.

4.3.3. Message type

As previously explained in Section 2.3.1, the internal nodes of a B^{ε}-tree contain buffered messages. In the B^{ε}-tree storage stack, a single B^{ε}-tree can only have one message type, i.e. messages of different message types can not be mixed inside a single tree. A message type is defined solely by a merge operation, which is used to merge messages intended for the same key, and an application operation, which enacts the encoded change on local data.

Because some read-modify-write operations can be translated into a single message insertion with a custom message type, the chosen message type and its operations are an essential part of an efficient data organisation scheme.

Following an object data write operation, the object metadata should be updated to reflect the prior write. Concretely, this means an update of the modification time and potentially its size as well, if the write increased the total object size.

The modification time is updated to the current system time after a write operation has completed, or when it aborts with an error. If this metadata update is delayed (e.g. if the current thread is interrupted), a concurrent write to the same object could set the modification to a later time, only to be reset back to an earlier time when the delayed thread resumes.

This inconsistency can be resolved by specifying a custom merge behaviour for messages that set the modification time: if both messages specify a modification time, choose the maximum of both, i.e. the later time.

The object size is intended to store the largest byte offset previously written to, but conditionally updating the object size after a write operation would require knowledge of the current object size, to determine if the current write would alter it. This inclusion of a mandatory read would impose a lower limit on the completion time of a write operation, and counter the advantage of using a write-optimised data structure.

This represents a read-modify-write operation[5] which can be optimised to a single message insertion if the merge operation is defined as the maximum, similarly to the modification time.

4.3.4. Separate metadata B$^\varepsilon$-tree

```
# Metadata tree
- Foo|object-id -> 42

# Data tree
- 42|0 -> [...]
```

Listing 4.8: Multiple trees

```
# Single tree with prefixed subtrees
- Meta|Foo|object-id -> 42
- Data|42|0 -> [...]
```

Listing 4.9: Single tree with prefixes

The data and metadata subtrees can be split into two separate trees (Listing 4.8), or merged together into a single tree (Listing 4.9). This choice affects performance properties of the object store, as well as future extensions and optimisations:

Disjoint subtrees
: The possible key sets of each merged subtree must be disjoint to maintain unambiguity regarding which subtree a key is a member of. Two methods of ensuring disjointness are prepending unique fixed values to each key of each subtree, or defining gaps of invalid keys in one subtree, which are then inhabited by the other subtrees.

Space usage of prefixes
: With prefixed subtrees, every key-value pair of the subtree stores a redundant copy of the prefix. For large chunks, this overhead is negligible, amounting to only 8 MiB overhead in key-value pairs for 1 TiB of object data for a chunk size of 128 KiB and a single-byte prefix.

Mixed message types
: Unfortunately, different message types can currently not be mixed in a single B$^\varepsilon$-tree, which prevents the mixed usage of message types optimised for a specific subtree.

Different storage pools per tree
: By backing the different trees of an object store with different storage pools, different storage media could be configured, so that the object store uses one set of media for metadata, and another for data.

[5] Where the modify step compares the current value, and only conditionally modifies it

Per-tree overhead
: Cache pressure would be slightly reduced if a large amount of concurrent object stores with only one tree are used, because the amount of root nodes is reduced from two/three to one.

Consistency
: With a future extension for per-dataset synchronisation, the assumption that every tree is written back to disk at the same time may no longer hold. If the entire state of an object store is contained in a single tree, it will remain consistent, but if it is spread across multiple trees, and the system crashes after writing back only one of multiple trees, the resulting object store may be inconsistent.

Snapshots
: Because the existing snapshot mechanism of the B^ε-tree storage stack operates on single datasets/trees, offering snapshots for single-tree object stores would be easier than having to coordinate snapshots of multiple trees.

Two separate B^ε-trees are chosen to allow for the different message types described in Section 4.3.3, which are essential in providing consistency under concurrent operation, and implementing metadata updates without read-modify-write cycles.

4.3.5. Key-value encoding format

The previous key-value listings used a symbolic notation for readability, with abstract key elements separated by pipe characters, but the B^ε-tree storage stack uses variable-length byte sequences internally, which entails an encoding scheme for each symbolic key and value.

Fixed metadata keys are indexed simply by the object's name, whereas custom metadata is indexed by concatenating the object's name, a zero byte, and the metadata key. As a result, object and metadata names are restricted to not contain bytes with a value of zero. Object data chunks are always indexed by a 12-byte key, where the first 8 bytes constitute a 64-bit unsigned object identifier in big-endian byte order, and the remaining 4 bytes represent the chunk identifier, which is similarly a 32-bit big-endian unsigned integer. The byte order is essential in ensuring that logically sequential chunks are also sequential in the B^ε-tree (and thus iteration order).

4.4. API design

The functionality described in Section 4.1 needs to be exposed to the library user in a manner that is simultaneously convenient and performant, while favouring safety by design over options which depend on the user's careful usage for safe operation.

4.4.1. ObjectStore API

The primary entrypoint of the object storage extension is the `ObjectStore` structure, which is opened from a `Database`, as shown in Listing 4.10. A single B^{ε}-tree storage stack database can hold multiple object stores, each with a different unique user-provided name.

Once constructed, an `ObjectStore` can be used to list ranges of objects, or to work with individual objects. There is no need to explicitly create an object store before opening it, and although it is possible to close a previously opened object store, this is usually unnecessary.

```
struct ObjectStore {
  data: Dataset,
  metadata: Dataset,
  ...
}

let store = database.open_named_object_store(b"images")?;
```

Listing 4.10: ObjectStore API

4.4.2. Object API

An object is uniquely identified by its original key. As most operations involve the data subtree, which is indexed by the object identifier, it is cached to avoid an unnecessary lookup per operation. A simplified version of the `Object` type can be found in Listing 4.11.

```
pub struct Object {
    key: Vec<u8>,
    id: ObjectId,
    // ...
}
```

Listing 4.11: Definition of Object

Operations on an `Object` can only be performed in conjunction with an `ObjectStore`, because the datasets making up an object store are held by the `ObjectStore` struct. Multiple variants of association between `ObjectStore` and `Object` were considered and implemented, as each comes with its own advantages and drawbacks:

4.4.2.1. Unchecked keys

The `Object` key could be passed directly to an `ObjectStore` for every operation, shifting the responsibility of matching up keys with the object stores in which the object exists to the user. An example of this approach can be found in Listing 4.12.

Although some mismatches can be caught at runtime, e.g. on deletion of a non-existing object or when reading from an object located in another object store, other operations will lead

```
// Create an `Object` key for the object `foo` in `object_store`
let foo = object_store.open(b"foo")?;

// Write "bar" to byte offset 42
object_store.write_at(foo, b"bar", 42)?;

// Object-ObjectStore mismatch is possible, with undefined consequences,
// as `foo` only exists in `object_store`, not in `wrong_object_store`
wrong_object_store.write_at(foo, b"this will break something", 21)?;
```

Listing 4.12: Example usage of unchecked object keys

to unintuitive consequences, such as attempts to write to an object which only exists in a different object store. While it would effectively prevent this issue, an additional check for object existence on every write operation would contradict the focus on write optimisation. A mismatch between object key and object store is always considered a logic error, and there are no supported usecases where a mismatch is useful or necessary.

4.4.2.2. Checked keys

Instead of validating e.g. object existence on every write operation, object keys could hold a tag corresponding to the object store. This tag could be the full name, a hash of it for fixed space usage, or simply an incrementing session-unique counter. On every operation, the object store's tag would then be compared to the tag of the key, and the operation is aborted with an error or a panic if there is a mismatch, potentially causing the entire process to halt if the panic is unhandled.

These runtime checks must execute on every operation while enabled, adding extra cost even to correct usage, while burdening the user with having to decide if and how to react to these logic errors. If implemented with conditional compilation, it would be possible to keep the safety feature on during development, but disable the additional memory and time cost for deployment, without having to alter the user's code.

If manual construction of object keys is permitted, these precautions do not necessarily prevent undefined behaviour, as the user could still construct a key with a mismatched tag. Further, this variant can at best throw an error during runtime, and may still miss certain conditional code paths, potentially resulting in logic errors during deployment after the safety feature was disabled, if those code paths are only executed under very specific circumstances. Static guarantees should be preferred where possible, so that invalid associations are either not possible by design, or detected at compile-time.

4.4.2.3. Object handles

An object handle carries a unique identifier for the referenced object, as well as a reference to the `ObjectStore` it was created from. Future operations are then invoked from the `ObjectHandle` itself, which implements them by using the stored references, as demonstrated in Listing 4.13.

This neatly maps onto many usecases of an object, statically ensures the object handles can not outlive the store, and prevents operating on an object with a mismatched object store.

In this variant, object handles are only obtainable from the object store the object exists in.

```
struct ObjectHandle<'store> {
  store: &'store ObjectStore,
  object: Object
}

let (obj, info) = object_store
    .open_object(b"foo", StoragePreference::NONE)?;

// info contains modification time and size

obj.write_at(b"bar", 42)?; // Write "bar" to byte offset 42
```

Listing 4.13: Example usage of object handles

4.4.2.4. Object handles with unchecked construction

As a combination of both previous approaches, a function for unchecked construction of object handles is provided, affording the flexibility of unchecked keys when required, while retaining the convenience and safety of object handles when constructed directly during opening or creation of an object.

Because this unchecked construction mechanism allows the creation of an `ObjectHandle` to an object which exists in a different `ObjectStore`, and subsequent use of that handle may result in data corruption, the user still needs to be careful when mixing objects from different `ObjectStores`. However, this increased vigilance is only required when manually constructing `ObjectHandles`, not for automatically associated handles. As mistakes can only be made during creation instead of on every use, this approach limits the code sections which require additional auditing, when compared to the previous unchecked approach.

4.4.2.5. Discussion and Decision

Object handles are more convenient to use than unchecked keys, by avoiding the need of repeatedly passing the object information on every operation, but additionally prevent dangerous logic errors involving mismatched `ObjectStores` with objects from other stores, and simplify the caching of intermediate information to reduce lookups. Checked keys add avoidable overhead to catch logic errors, but never as early as the object handle variants, which avoid the mismatch by design.

The `ObjectHandle` construct is most suited for cases with clear-cut lifecycles of objects, where an object is opened or created, a number of operations are made, and the object is then deleted or closed. It is less compatible with more complicated lifecycles, especially externally defined

ones such as within a FUSE[6] filesystem. FUSE demands a stateful interface, where open objects are represented to the calling party as file descriptors. This requires keeping a mapping of open objects, to be closed at the client's convenience, which would require holding the inner `ObjectStore` reference of an `ObjectHandle` for an externally defined period. This makes it non-trivial or impossible to prove to Rust's borrow checking mechanism that an `ObjectHandle` is never used after its `ObjectStore` is destroyed. As the implicit lifecycle of unchecked keys is not managed or checked by the Rust compiler, they are better suited for such complicated lifecycles.

Safely-created object handles would be unacceptably restrictive as the only available interface, unnecessarily hindering the implementation of plausible usecases involving complicated lifetimes. With the addition of unchecked handle construction as an escape mechanism, the advantages of both approaches can be combined, letting the user trade safety for flexibility when required.

4.4.3. Cursor

Many external APIs are built around a streaming IO interface, instead of the batched **Write** and **Read** interface provided by the B^{ε}-tree storage stack. A streaming interface for object handles is presented, to facilitate interoperation with other streaming interfaces, without requiring the user to manage a buffer manually at every call-site.

Constructed via `ObjectHandle::cursor`, the `ObjectCursor` maintains an internal position and dispatches calls to the standard library's `Read` and `Write` interfaces to the underlying object via **Read** and **Write**. The internal position can be modified via Rust's `Seek` interface. Because it is built on top of **Read**, read operations via the cursor perform zero-filling if used with sparse objects. If the cursor is used primarily with small operations, e.g. reads of a single byte, usage of Rust's IO buffering facilities (`BufReader` and `BufWriter`) is essential to performance.

The naming is adopted from database terminology and matches Rust's `std::io::Cursor`[7], which provides similar functionality for otherwise random-access byte sequences.

```
let obj = store.open_object(b"example")?;
// create a new cursor, initialised to the beginning of the "example" object
let mut cursor = obj.cursor();

io::copy(&mut input, &mut cursor)?;

// jump back to the beginning of the stream
cursor.seek(SeekFrom::Start(0));
```

Listing 4.14: Example usage of `ObjectCursors`

[6] Filesystem in userspace (FUSE) is an interface which allows the implementation of custom filesystems without editing operating system code.

[7] https://doc.rust-lang.org/std/io/struct.Cursor.html

4.5. Summary

The presented object store supports the essential operations **Create**, **Read**, **Write**, and **Delete**, and a handle-based interface with **Open** and **Close**. Additionally, a user can wait until all changes have been written to persistent storage with **Sync**, list objects with **List**, change the key of an object with **Rename**, and query fixed object metadata with the **Status** operation. Custom metadata can be queried, listed, and set or deleted by name.

Two optional operations have not been implemented: **Link/Unlink** and **Copy**. The implementation of **Link/Unlink** would require only minor changes to the metadata tree. A naive full data copy can be implemented on top of the exposed API without any changes, even for sparse objects, but an efficient implementation of **Copy** with data sharing between all instances of an object, and chunk-level copy-on-write semantics is left for future work.

In conclusion, the object store's data and metadata is organised as follows: Data is split into chunks, which are potentially non-contiguous (sparse) and indexed by an object identifier via a level of indirection, instead of using the full object name. The object identifier is packed into a single key-value pair alongside other fixed metadata: the modification time and object size. Fixed metadata is interleaved with custom metadata, which are user-provided per-object key-value pairs. Each object store uses two datasets (B^ε-trees) to organise metadata separately from object data, in order to improve metadata locality.

The final layout and a subset of the API (sparse object writes, metadata, cursor writes) are demonstrated below: Listing 4.15 uses the public object API to create the two datasets shown in Listing 4.16. Matching colours are used to mark the origin of key-value data in the function arguments.

```
let os = db.open_named_object_store(b"example", StoragePreference::NONE)?;

// Create a new object and write to two non-contiguous locations
let obj1 = os.create_object(&[10, 10, 10])?;
obj1.write_at(&[42, 43, 44], 10);
obj1.write_at(&[1, 2, 3, 4, 5], 500 * 128 * 1024);

// Add a key-value metadata pair, with key "foo" and value "bar"
obj1.set_metadata(b"foo", b"bar")?;

// Use the cursor API on another object
let obj2 = os.create_object(&[20, 20])?;
let mut cursor = obj2.cursor();
// Each write advances the internal position ...
cursor.write_all(&[1, 4, 3, 2])?;
cursor.write_all(&[5, 6, 7, 8])?;
// ... which is also exposed via the standard library's Seek interface
cursor.seek(SeekFrom::Start(1))?;
cursor.write_all(&[2, 3, 4])?;
```

Listing 4.15: Demonstration of object creation, and (meta-)data write operations

```
// Metadata dataset:
// Fixed metadata for an object with the key [10, 10, 10]
[10, 10, 10] -> [0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 232, 3, 0, 0, 0, 0, 41, 248, 251, 96,
    ↪ 0, 0, 0, 0, 144, 186, 171, 49]
// Custom metadata for the same object, for metadata key [102, 111, 111] (foo),
//                                     with value [98, 97, 114] (bar)
[10, 10, 10, 0, 102, 111, 111] -> [98, 97, 114]
// Fixed metadata for the object [20, 20], whose object id (first 8 bytes)
// is one larger than that of the previous object
[20, 20] -> [1, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 41, 248, 251, 96, 0, 0,
    ↪ 0, 0, 48, 202, 171, 49]

// Data dataset:
// Two chunks of the first object, both with an object id (first 8 bytes) of 0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 42, 43, 44]
// Chunk 500 (1 * 256 + 244 = 500), far after the first chunk but sparsely allocated
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 244] -> [1, 2, 3, 4, 5]
// First chunk of the object with an id of 1
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0] -> [1, 2, 3, 4, 5, 6, 7, 8]
// Object id counter (\x00oid), storing the most recent id of 1 in little-endian
[0, 111, 105, 100] -> [1, 0, 0, 0, 0, 0, 0, 0]
```

Listing 4.16: List of key-value pairs from the metadata and data datasets, after the execution of Listing 4.15

Chapter 5.

Tiered storage

As described in Section 2.2, tiered storage systems can combine the advantages of heterogeneous storage devices without data duplication (which would constitute caching), and the decision to move data between storage tiers can originate from an automated policy, or the user/client application.

Considering that the design and evaluation of migration policies is a research problem on its own, as described in Section 3.2, this work focuses on statically-assigned tiered storage. More sophisticated allocation and migration processes can still be built on top of a static system, and per-object custom metadata can be used to track application-level usage. An application background process could slowly migrate old objects based on their age and most recent access.

5.1. Division into storage classes

There are different properties in which storage media can differ from another, often with inter-dependencies. Among them are:

Capacity
: The amount of data which can be stored on a medium. With a fixed dataset, this directly determines how many devices are necessary.

Throughput
: How much data can be read or written per device over a limited timespan. This is often strongly dependent on the access mode, e.g. block size and access pattern.

Access latency
: The time between requesting a storage operation and its fulfilment. Similarly to throughput, this can vary greatly with the access mode used.

Optimal access pattern
: The order of access, e.g. sequential block access or apparently random[1], will have different impacts on throughput depending on the device type (SSD, HDD, tape, etc.).

[1] here: too unpredictable for the disk controller/operating system to correctly detect a pattern

Optimal granularity of access
: Due to the internal structure of a device, it may be particularly suited to requests of certain sizes. Some SSDs can be reformatted to use differently sized logical blocksizes, with different performance characteristics.

Concurrency
: The number of simultaneous operations a device can concurrently process. To fully take advantage of a device with concurrency, an application may need to submit many operations at once, instead of waiting for the previous operation to finish.

Backing storage type
: A flash-based storage device may need special procedures, such as communicating unused areas of storage to the device (trimming). Other peculiarities might be lack of wear levelling on less sophisticated flash storage devices, or the use of shingled magnetic recording on a hard drive. Although these circumstances will be reflected in the other properties, knowledge of the underlying cause can allow for special handling by a storage system.

On-device cache
: The type, capacity, and speed of any caches on the storage medium affect burst latencies. Some SSDs use a portion of their capacity in a faster mode as a write cache.

On-device storage controller
: Different storage controllers may exhibit different behaviour when confronted with the same operations, resulting in a different overall system behaviour. Differences could include garbage collection strategies, thermal throttling behaviour, or readahead parameters.

Redundancy
: Although loss of a single storage medium is often handled by the host system, a hardware RAID setup might qualify as a single logical device with some data redundancy.

Chance of successful disaster-recovery
: In the case of a failed storage device without redundancy, recovery chances and their cost can factor into their suitability for storage of important data.

Data durability
: The durability of data at rest is of particular importance for long-term archival. Error correction in controller memory and write caches could also reduce the risk of corruption during recording.

Software aggregations of storage media, such as mirrors, stripes, and parity constructs, also have these properties, which are partially influenced by the properties of their constituent devices, but also determined by implementation tradeoffs.

By measuring the properties of each storage medium, and then every aggregation, they could be placed in a high-dimensional space of tradeoffs, and then picked by a sophisticated selection algorithm for their strengths and avoided for their weaknesses. This would require reliable (preferably automated) measurements, a well-tuned tradeoff function, and knowledge of the requirements of each user. Some of these properties can change by usage, and would need to be tracked or re-measured by the storage system. For example, the throughput of a hard disk

can be influenced by the addressed cylinder, and the allocation strategies of a storage controller can change as the stored data approaches capacity.

While such a complex system would have the potential of outperforming one with less sophistication, its implementation is not realistic in this context, and perhaps not even desirable, if most of the potential can be achieved with less complexity.

Despite the speed of a storage device being very multi-faceted, a small and discrete one-dimensional trade-off space was chosen due to its simplicity, both in implementation, client development, and deployment configuration. Specifically, the user or system administrator is expected to manually form sensible aggregations, and then classify devices and aggregations into at most one out of **four storage classes**. Devices in lower storage classes are generally expected to exhibit less latency, higher throughput, and diminished capacity, when compared to devices from higher storage classes. This ordering expectation is not verified by the database, and its fulfilment is not required for correctness, only for performance. Furthermore, the devices of each class are expected to be homogeneous, as the database does not differentiate between devices within a single class.

This reduction of several attributes into one ordering makes the assignment into classes unclear if only one or a few of these ordering expectations are fulfilled, e.g. with a device exhibiting higher throughput but also higher latency than the devices of an adjacent class.

This shifts the effort and complexity of measurement and selection into the user's or administrator's responsibilities, who are assumed to be more competent at reasoning about the combined performance of the system and application than a naive implementation of the complex automatic tradeoff would be.

Applications can then specify a one-dimensional preference for a particular storage class, and the storage system will try to fulfil that preference or fall back to a default if no preference was given.

The choice of four different storage classes is a simplification in many ways, and assumes falsely that the other properties remain uniform in each storage class, unable to differentiate based on other properties not included in the coarse ranking.

5.2. Cross-class disk addressing

Tree nodes are referred to by their disk offsets, previously comprised of a 12-bit disk identifier, and a 52-bit block offset relative to the start of that disk. With the introduction of storage classes, disk offsets also need to identify the storage class in which the disk resides. 2 bits are repurposed from the disk identifier to identify the referenced storage class, resulting in a new disk offset layout as shown in Figure 5.1.

An alternative representation might retain the 12-bit disk identifier, mixing different storage classes and resolving the storage class implicitly. This would allow moving a disk between classes without breaking existing references, but it would require per-disk lookups in the storage pool configuration to determine the storage class of a block reference.

Shrinking the disk identifier from 12 to 10 bits reduces the upper limit on top-level vdevs in a single storage class from $2^{12} = 4096$ to $2^{10} = 1024$, which should still far exceed the requirements of any realistic deployment, as the risk of data loss increases with larger stripe

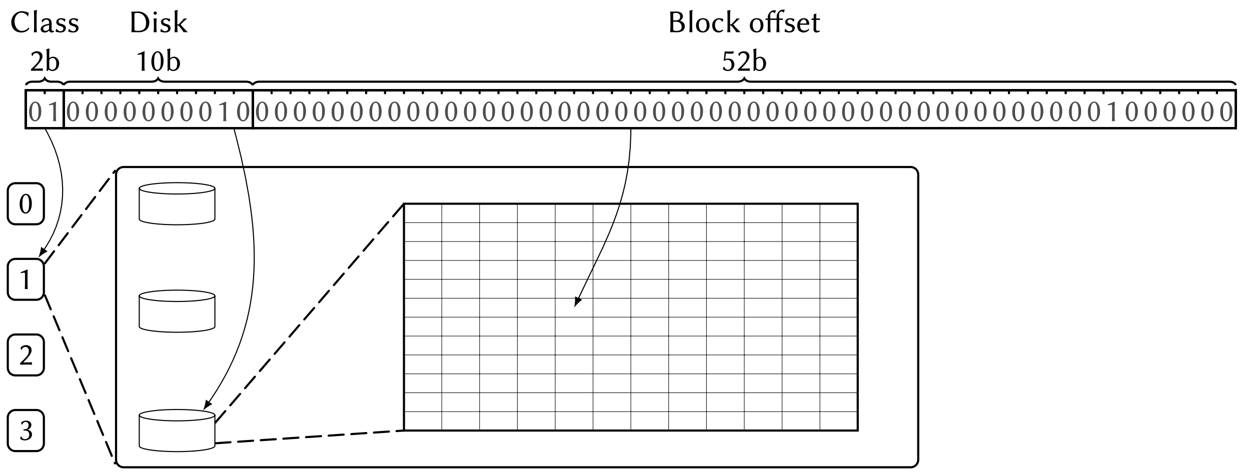


Figure 5.1.: Layout of block offsets

configurations, and each referenced top-level vdev can itself be a mirror or parity vdev instead of a bare disk.

The B^{ε}-tree storage stack currently has no support for any form of storage pool reconfiguration: once a database has been created, the topology of the backing vdevs can not be altered safely. This lack of flexibility manifests in the tiered storage extension in the form of fixed disk identifiers. Simply swapping two top-level vdevs in a storage class will invalidate all existing references to the contained blocks, resulting in unreadable data or corruption. This is not a permanent disadvantage, as a redirection table could be added into the root tree, keeping stable identifiers for all previously seen devices.

5.3. Configurable allocation strategy

A client application might attempt to use more storage classes than are available on the host system, or assume conflicting properties about a given storage class (e.g. that class 1 is always backed by SSDs). The used storage classes should be configurable, because the client developers may make different tradeoffs than the deploying parties.

Another adjacent problem can be found in the handling of allocation failure: If a client application requests an allocation in an already filled storage class, the storage stack could immediately deny the request, or fulfil the request by allocating on a different storage class than requested. Unfortunately, the B^{ε}-tree storage stack currently performs no space accounting, i.e. there is no information about which vdevs (and thus storage classes) still have free space remaining. As a result, allocation attempts on full storage classes result in the futile traversal of all allocation bitmaps, which is a very expensive operation. Falling back to other storage classes on an allocation failure would only amplify this cost, at least until space accounting is implemented. By encoding the fallback behaviour into the allocation strategy, this tradeoff can be made by the user, if necessary.

Both of these issues are addressed with the introduction of a configurable allocation strategy, which takes the form of a list containing a maximum of four nested lists, each with at most four integers in the range of zero to three (inclusively). Each position in this list, and each

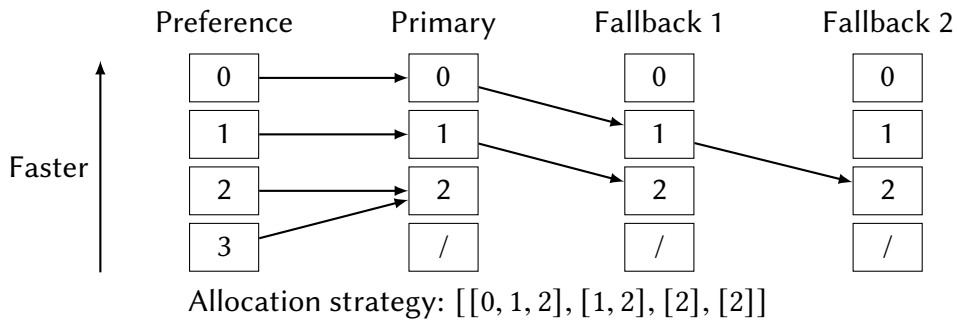


Figure 5.2.: An example of class remapping and fallback configuration by specifying an allocation strategy

integer in the nested lists, corresponds to one of the four storage classes. When the B^ε-tree storage stack needs to allocate a tree node on a given storage class, the nested list at the corresponding position in the allocation strategy is used to inform the order of allocation attempts. For example, the allocation strategy `[[0], [1], [2], [3]]` could be called an identity strategy without allocation fallback, because each nested list contains contains only the class corresponding to its position.

If an application only uses a fixed set of storage classes, the allocation strategy can map those fixed classes onto deployment-specific classes. Similarly, if the application expects four storage classes to be available, but the host system only has two different types of storage device, multiple application-level classes can be aliased to use the same host-level class.

In Figure 5.2, a hypothetical client application uses all four storage classes, but the host system only has three classes available, and classes `0` and `1` are of comparatively low capacity. The administrator could reassign a few devices from class `2` to class `3`, but that might result in unbalanced utilisation, if the application does not access both classes equally. One of the classes might fill up before the other, potentially halting the entire system even though there still is free space available. Assuming striped access, the application would additionally forfeit the benefit of increased concurrent access provided by having more devices to distribute operations across.

By using the allocation strategy `[[0, 1, 2], [1, 2], [2], [2]]`, allocation requests for class `0` will be fulfilled by trying to allocate on class `0` first, but falling back to classes `1` and `2` if class `0` was full. Similarly, failed requests for class `1` will fall back to class `2`. Requests for classes `2` and `3` will be allocated on class `2`, without any fallback.

This remapping/resolution occurs when a block is allocated, and only the resolved class is encoded into the storage address to ensure that changes to the remapping table do not break block references.

5.4. Block allocation

With a storage layer capable of addressing multiple storage classes, and a tree type supporting cross-class and cross-device block references, the allocator needs to choose among all devices of all storage classes when a node is about to be written to disk. The goal of each allocation

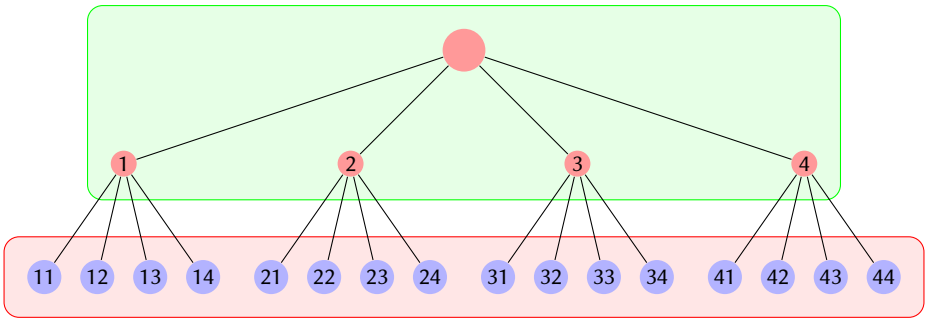


Figure 5.3.: A fully populated tree with a fanout of four. The internal nodes are grouped in green, and the leaf nodes are grouped in red.

heuristic is to optimise future accesses to the allocated blocks, based on predictions about future storage layer requests.

A block allocation heuristic determines multiple important characteristics of the whole system:

Immediate effects

Allocations are triggered during write operations, so naturally the choice of storage class to allocate blocks from influences the completion time of that write operation. Choice of a class with lower access latencies benefits small writes, whereas larger writes benefit from higher throughput, but these are not necessarily constant in a multi-user system such as a storage server. In the event of resource contention, an operation may complete sooner even if the allocator selects a lower class than requested, if that class is currently underutilised.

There is also a risk of uneven block allocation causing allocation failures due to one layer being out of free space, when other classes might be able to accommodate the allocation. Falling back to lower or higher classes in such an event allows continued operation, at the cost of potential performance degradation later.

Delayed consequences

The time required to read back a block from storage depends on the storage medium it was written to, which is influenced by the storage configuration and allocation strategy.

With trees as the primary block organisation method, few block requests are entirely isolated from another. For read access to infrequently used values, a traversal from root to leaf may require fetching multiple blocks. If this path happens to cross storage class boundaries, the total duration of the critical path depends not only the storage configuration of the final node, but also that of every intermediary node. If, for example, fetching a value residing on fast solid-state storage required fetching an internal node from a suspended spinning disk, the total access time would not conform to the access latency expectations of solid-state storage. It is thus important to keep the entire access path as fast or faster than requested, instead of only the final node.

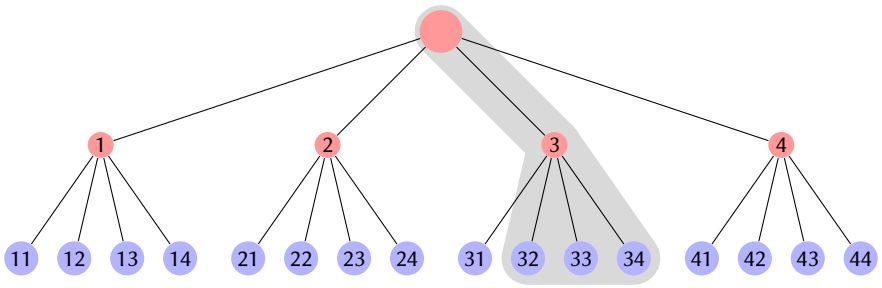


Figure 5.4.: Distribution by subtree

5.4.1. Allocation by tree layer

This heuristic predicts that future accesses are more likely to require an internal tree node, than a leaf node. As the internal tree nodes are needed during every access of any leaf node[2], and written back frequently when buffering messages, allocating the internal nodes on a faster storage class, and the leaf nodes on a slower storage class, can reduce the duration of background **Sync** operations and accelerate the traversal of uncached tree paths.

The average read completion time for point queries is expected to improve over exclusive use of the slower class, but the throughput of larger queries is still limited to that of the slower storage devices.

The reading benefits can only be realised for uncached nodes, but the internal nodes are also likely to be present in the object cache, partially negating the advantages of this distribution scheme with usecases where most of the internal nodes are cached. Additionally, the ratio of leaf nodes to internal nodes follows from the fanout (number of child nodes for internal nodes): Figure 5.3 depicts a fully populated tree with a fanout of four, which contains 16 leaf nodes and five internal nodes at a ratio of $\frac{16}{5} = 3.2$. As the fanout is increased to optimise the performance of uncached random point queries, which depend on a high fanout to ensure a low tree depth, this ratio approaches the fanout itself. As an example, given a fanout of 100, the leaf nodes would outnumber the internal nodes by a factor of nearly 100. If the capacities of both storage classes do not match this ratio, under-utilisation will occur. The user would have no means of increasing utilisation, as the fanout is not configurable.

5.4.2. Allocation by key priority

The previous method treated all keys (and by extension: objects) equally, making a prediction based only on the type of node being allocated. In contrast, this heuristic assumes that future access will not be uniform across the key space, but rather that subsets of keys will be accessed more often than others. Instead of attempting to automatically determine these subsets, the prioritisation is left to the invoking software. The developers of client software are already aware of the domain-specific meaning of objects and their keys, and can make a prediction regarding their access patterns.

This user-provided prioritisation could be useful to e.g. allocate media objects based on expectations about access frequency from similar objects, or to allocate more frequently accessed sections of an object, e.g. the index of a publicly served zip archive, on an SSD, while keeping

[2]Except when the tree consists only of the root node

the bulk of the archive on cheaper storage. Even without keeping statistics about past accesses, an image thumbnail in a gallery could be expected to be accessed more frequently than the corresponding full image.

Because each node may contain multiple keys with different priorities, honouring each priority exactly is infeasible if nodes are to be allocated in sequential blocks. If the conflicting priorities of each key can not be fulfilled exactly, the allocator instead selects a single class to satisfy all requirements, by assuming that an upgrade to a faster class is always acceptable for the client. If no preference was specified for any of the keys, the allocator falls back to a global default storage class.

Additionally to the leaf nodes, data pertaining to a specific key may also be found in the form of buffered messages in the internal nodes, resulting in key-value pairs potentially being distributed across nodes in the entire access path. Because every read operation requires traversal from the root, allocating only the leaf node on the requested storage class would be contradictory to the goal of accelerating the entire access, when the internal nodes are not cached and need to fetched from a slower class.

To prevent such situations, the storage preferences of an access path must be monotonically slower, if traversed from the root. By accounting for the storage preferences of child nodes, this prevents a transition from a slower storage class to a faster storage class during traversal.

Compared to allocating all nodes on a slower class, this might result in some operations slowing down if the access path crosses storage class boundaries and the overhead of querying different classes (e.g. wakeup time, loss of request-global IO scheduling, etc.) outweighs the cost of fetching more data from the slower class.

Due to the greater flexibility and larger predicted performance potential of this allocation heuristic compared to allocation by tree layer, the remainder of this chapter is concerned with defining and implementing node allocation by key priority.

5.5. Storage preferences

Due to the mixture of differently prioritised insertions in nodes, and the fallible nature of allocation from exhaustible resources, requests for specific storage classes can not always be fulfilled. This is represented internally, as well as in the public API, by using **storage preferences** instead of specific storage classes.

Each of the four storage classes can be specified as a storage preference, indicating an application request that the associated key-value pair should be allocated on the corresponding storage class or a faster class.

To indicate a lack of preference, storage preferences allow an additional value, here called ***None***. In contrast to falling back to a global default storage class whenever the user fails to provide a preference, a distinct *None* value can losslessly represent that non-decision. This detail is relevant when changing the default storage class, as otherwise the database would not be able

to distinguish between data allocated intentionally on the previously default storage class, and data allocated without any preference[3].

The resulting possible values of a storage preference are 0, 1, 2, 3, and *None*.

$$\mathcal{P} = \{0, 1, 2, 3, None\} \tag{5.1}$$

To resolve a conflict between multiple preferences, a total order of **strictness** is defined over $\mathcal{P}$ with the binary relation $a \prec b$, where a is a stricter preference than b. The integer preferences retain their integer ordering, and every other preference is stricter than *None*.

$$\prec \subset \mathcal{P} \times \mathcal{P} \tag{5.2}$$

$$\prec = \{(a, b) \mid a, b \in [0, 3] \land a < b\} \cup \{(x, None) \mid x \in [0, 3]\} \tag{5.3}$$

Akin to $min(a, b)$, $a \curlywedge b$ is defined as the **stricter** of the two preferences. As an example of using the $\curlywedge$ operator, a conflict between 1 and 3 is resolved to $1 \curlywedge 3 = 1$, a conflict between *None* and 3 is resolved to $None \curlywedge 3 = 3$. Only a conflict between *None* and *None* results in *None*.

$$a \curlywedge b = \begin{cases} a & \text{if } a \prec b \\ b & \text{otherwise} \end{cases} \tag{5.4}$$

Similarly, the $\curlywedge$ operator is an equivalent to $\min_{x \in S} x$. $\curlywedge S$ is recursively defined as the strictest preference in the set of preferences S, with the preference of an empty set defined as *None*. The selection of $x \in S$, and thus the order of reduction, does not affect the final result, as $\curlywedge$ is both associative and commutative.

$$\curlywedge S = \begin{cases} None & \text{if } S = \emptyset \\ (\curlywedge S \backslash \{x\}) \curlywedge x & \text{if } \exists x \in S \end{cases} \tag{5.5}$$

If the final conflict resolution is *None*, a configurable global default storage class is chosen, otherwise the corresponding storage preference is selected. The remapping and fallback process described in Section 5.3 is applied to the final preference during block allocation.

Explicit storage classes are not exposed to the user, and are only used internally after allocation has succeeded.

5.6. Preference associations

When inserting a message for a particular key into a tree, the caller additionally passes a storage preference for that key-message pair, which is stored alongside it. When a message application results in the creation of a key-value pair in a leaf node, that storage preference is preserved alongside the new key-value pair.

While this representation technically allows for a single key to have different preferences when spread into multiple buffered messages, and the interface can not prevent this from occurring,

[3]For example, if the administrator changes the default storage class due to a lack of free capacity on the previous default class, reallocations of a tree node filled with *None* preferences would be using the new default, whereas they would continue using the old default if *None*s were instead eagerly resolved.

mixed storage classes for a single key are deemed unnecessarily difficult to predict, and keys (but not objects, as will be explained in Section 5.8) should be treated by users as having only one storage preference.

To allocate the blocks for storing a tree node, the data management layer queries the storage preference of that node. Although the actual implementation avoids excessive recalculations, at an abstract level each node can be thought of as not storing a storage preference itself, but rather recalculating the preference recursively from its contents.

Depending on a node's location inside the B^{ε}-tree, it can either be an internal node, or a leaf node:

$$P_{node}(n) = \begin{cases} P_{leaf}(n) & \text{if n is a leaf node} \\ P_{internal}(n) & \text{if n is an internal node} \end{cases} \tag{5.6}$$

A leaf node l consists of key-value pairs ($entries(l)$), each of which carries a storage preference (queried with $P_{entry}(e)$). The overall preference of the leaf node is defined to be the strictest preference of all entries.

$$P_{leaf}(l) = \bigwedge \left\{ P_{entry}(e) \mid e \in entries(l) \right\} \tag{5.7}$$

An internal node i consists of one child buffer for each of its child nodes ($children(i)$). Each child buffer b contains a reference to its corresponding child node ($child(b)$), and a variable amount of buffered messages intended for that child node ($messages(b)$). Each child buffer's preference is determined by the strictest buffered message, and the preference of its corresponding child node, as a node must not have a preference which is less strict than that of any of its children.

$$P_{childbuffer}(b) = \bigwedge \left\{ P_{message}(m) \mid m \in messages(b) \right\} \wedge P_{node}(child(b)) \tag{5.8}$$

The overall preference of an internal node is then defined as the strictest preference of all contained child buffers.

$$P_{internal}(i) = \bigwedge \left\{ P_{childbuffer}(b) \mid b \in children(i) \right\} \tag{5.9}$$

5.7. Incremental tracking of preference propagation

The computation of $P_{node}(n)$ would involve all transitive children of n, a prohibitive cost if any children are not already cached. Instead of repeatedly calculating the preferences of nodes, they can be incrementally computed, adjusting a stored preference as necessary for each operation. Each tree node hierarchically keeps track of its current storage preference. In the context of storage preferences, **tracking** by a tree node refers to storing and maintaining a storage preference appropriate for the contents of that node. Furthermore, **upgrading** and **downgrading** of a storage preference designate a transition to a preference with greater or lesser strictness, respectively.

Unfortunately, certain operations complicate the tracking process. For example, if a leaf node l contains three key-value pairs with the storage preferences 2, 1, and 2, then $P_{leaf}(l) = 1$ in accordance with Equation (5.7). Upon the removal of a key-value pair however, no certain statements can be made regarding the exact new value of $P_{leaf}(l)$ without inspecting the

remaining pairs: if the removed entry had a preference of 1, the new overall preference of l will be 2, whereas it would continue to be 1 if one of the entries with a preference of 2 had been removed. With more realistic leaf node sizes, iterating the internal data structure of a leaf node on every removal to determine the new storage preference becomes undesirably costly.

To remedy this flaw, a fifth preference state is added internally: **unknown**, indicating that the relevant preference is currently not known. The transition into the *unknown* state is also referred to as preference invalidation. Instead of eagerly causing an immediate recalculation of the node's preference after a removal, the cached preference can instead be set to *unknown*, which defers the recalculation until the storage preference is next queried. This mechanism also avoids unnecessary recalculations, when an invalidating operation is performed repeatedly. Due to the hierarchical design, the recalculation step of an invalidated node does not have to recursively scan all of its transitive children, instead inspecting only itself and the preferences of its direct children.

As the correctness of the implementation of incremental tracking is important for the system's performance, the following invariant must hold for any reachable tree state: whenever the tracked preference of any given node n is queried, it must either equal $P_{node}(n)$, or *unknown*.

The tracking mechanism is explained next, along with an inductive argument that it upholds the invariant for all reachable tree states, by starting with the creation of empty components and discussing every operation that could result in a change to the tracked storage preferences. The argumentation of each operation's correctness can assume that the tree state before the operation fulfils the invariant, and only reasons about the newly reachable tree states.

5.7.1. Node types

As previously discussed in Section 2.3.1, there are two node types present in a B^ε-tree: leaf nodes, and internal nodes. A leaf node contains entries consisting of a key, value, and a storage preference.

An internal node contains pivots, and one child buffer for each child node. A child buffer contains a reference to the corresponding child node, and the buffered messages intended for the key range of that child. Whereas the leaf node directly tracks its storage preference, the child buffers of an internal node each individually track one preference for the buffered messages intended for their child node.

When the internal node is queried for its storage preference, it has to iterate over its child buffers to select the strictest preference. This is necessary because the node references of a child buffer can be updated without notification to the internal node, which could result in temporarily incorrect preferences if the internal node would cache the overall preference and the new node pointer necessitates an up- or downgrade of the internal node's preference. The effects of the following operations in Section 5.7.2 are not discussed for internal nodes, as the stateful tracking occurs only in their child buffers.

Preference propagation The child buffers of internal nodes hold references to their respective child nodes. These references allow (synchronised) replacement, and will be replaced e.g. when a node is modified in-memory, and then replaced again when that node is written to disk. Each such reference can be either *unmodified*, *modified*, or *in writeback*.

An *unmodified* reference contains a disk address, which encodes the storage class as described in Section 5.2. This can violate the invariant established in Section 5.7 if the encoded storage class differs from the original storage preference due to class remapping or allocation fallback. Instead of additionally storing the original preference of each node, this is considered an acceptable deviation because attempts to allocate on previously full storage classes are likely to fail again, and allocation failures are excessively expensive until space accounting is implemented.

References to nodes which are *modified* or *in writeback* have been altered to additionally hold a storage preference, which is calculated when inserting new nodes into the data management layer, and before writing them back to persistent storage.

The B^ε-tree storage stack already ensures that child nodes are queued for writing before their respective parent nodes, so that parent nodes can refer to the newly allocated positions of child nodes instead of their previous locations. This process is reused for incremental preference calculation, to ensure that the preferences of each child have already been calculated before the preference of their parent node is queried.

5.7.2. Node operations

5.7.2.1. Creation and loading from disk

In terms of storage preference tracking, empty node creation is the simplest operation.

An empty leaf node is defined to have a known storage preference of *None*, by the empty-set case of 人. When a leaf node is loaded from disk, its entries are already iterated over to construct its internal data structure, during which the strictest preference is determined, and then used as the cached preference of the new leaf node.

A child buffer can only be created for an existing child node, whose storage preference can be retrieved from the corresponding cache reference. As there can be no buffered messages yet, the queried preference can be cached.

When an internal node is read from disk, its child buffers retain the cached storage preference they were originally written to disk with, obviating the need for any recalculation after loading.

5.7.2.2. Message insertion

A message is inserted into a leaf node by querying for a value associated with the message's key, and then applying the message to the query result (even if it is empty). There are four resulting combinations:

- There was no value for the key, and the message application did not create a new one: No change.
- There was no value for the key, but the message application resulted in a value: Entry insertion.
- There was a value for the key, but the message application removed it: Entry deletion.

- There was a value for the key, and after message application there is still a value: Entry modification.

On creation or modification of an entry, the node preference is upgraded to that of the inserted message if the message preference is stricter, so that there is no entry with a stricter preference than the node itself.

When a message causes the removal of a key-value pair, the preference of the removed entry is compared to the current node preference. Because the previous tree state can be assumed to uphold the invariant, it follows that the preference of the removed entry can not be stricter than the node preference, leaving equal strictness and lesser strictness as possible comparison results. If the removed entry had a preference that is less strict than the node preference, there must be another entry with a stricter preference, and the cached node preference does not need to be invalidated. Only on removal of an entry with equal strictness is the cached preference invalidated, because it is impossible to determine with the available information whether the removed entry was originally responsible for the upgrade to the current storage preference. Similarly, if the new preference is less strict than the previous preference following the modification of an entry, the overall preference must be invalidated to allow for downgrades by overwriting to occur.

The process is simpler for internal nodes and child buffers, as insertions can never remove a message from these node types, and thus only increase the strictness of the node's preference. When inserting a message into a child buffer, its preference is upgraded if the message's preference is of greater strictness than the current preference.

5.7.2.3. Split

When splitting a leaf node, a range of its entries is removed and added to a new leaf node. As the new node is constructed iteratively, its cached preference is tracked during the insertion of each entry. The cache of the original node, from which those entries were removed, must be invalidated, because the removed keys may have increased the strictness of the tracked preference during their original insertion.

When a node is split into two new nodes, its parent node must also split the corresponding child buffer into two new buffers along the same key ranges. Both resulting buffers have an unknown preference, requiring a scan of their buffered messages on the next query for the internal node's preferences.

5.7.2.4. Merge

To merge a leaf node with another, the entries of the node with lexicographically greater keys are appended to the other node. If the preferences of both nodes are known, that of the growing node can be upgraded if necessary without inspecting each individual message.

When two nodes are merged, their parent node must also merge their corresponding child buffers into one. If the preferences of both buffers are known, the resulting preference can be determined without inspection of the stored messages by selecting the stricter of the two preferences. Otherwise, the newly merged buffer has an unknown preference.

5.7.2.5. Flush

Flushing of an internal node selects a child buffer, and propagates its messages to the corresponding child node. After this operation, the cached preference of the flushed child buffer needs to be invalidated, because the child buffer no longer contains any messages, and the preference of the child node may have been altered by the flushed messages.

5.7.3. Invariant checking

From the definitions of P_{node}, $P_{internal}$, and P_{leaf} follow these two derivative invariants:

1. During path traversal from the root, the encountered nodes have storage preferences of monotonically decreasing strictness. In other words: the preference of a parent node is as strict as or stricter than those of its child nodes.
2. If a cached node preference is available, the node may contain no entries or message buffers with a stricter preference.

They must similarly be upheld for every reachable tree state. As each tree component implements an uncached and an incremental variant of preference calculation, their results can be compared and asserted to be equal as a debugging measure, which can help detect the point of divergence if compared frequently enough.

5.8. Object store API changes

As the B^{ε}-tree storage stack is intended to be usable without taking advantage of tiered storage, the public API has been partially duplicated, adding function variants which allow overriding the used storage preference. Because objects consist of multiple key-value pairs, individual chunks of an object can be stored with different storage preferences than the rest, allowing for the prioritisation of object ranges.

The object handle and cursor API introduced in Section 4.4 have been extended to allow the specification and overriding of storage preferences without forcing the user to provide the active storage preference during every operation, as demonstrated in Listing 5.1.

```
let (obj, _info) = object_store.open_object(b"foo")?;
// An `ObjectCursor`, which implements the standard traits Read, Write, Seek
let mut cursor = obj.cursor_with_pref(StoragePreference::FASTEST);

writeln!(&mut cursor, "these changes will be allocated onto the fastest
    ↪ possible class");

cursor.set_storage_preference(StoragePreference::NONE);

// these writes will be performed with the `ObjectStore`s default preference
writeln!(&mut cursor, ...)?;
io::copy(&mut stdin, &mut cursor)?;
```

Listing 5.1: Example of object usage on multiple tiers

Chapter 6.

Database improvements

Numerous other changes to the B^{ε}-tree storage stack are described here, which are not related to either object storage or tiered storage.

6.1. Leaf node on-disk representation

Although other node types are encoded automatically with bincode[1] before being written to disk, leaf nodes are encoded manually into a format specifically designed to avoid a deserialisation step, so that the on-disk data of a leaf node can be efficiently queried without copying [Wiedemann, 2018, p.78].

Previously, the format of a packed leaf node matched the structure outlined in Listing 6.1, storing the number of entries contained in the node, and fixed-size positions and lengths specifying where the actual key data and value data could be found in the trailing `data` byte array. Importantly, the entries in `keys` and `values` are ordered identically to their unpacked counterparts, so that corresponding key and value positions have the same index in their respective arrays. To retrieve a value for a key, the index of the value location in `values` can be located by using binary search for the key via `keys`.

The following changes have been made, resulting in the new layout described in Listing 6.2:

1. Because keys and values are written into `data` sequentially, the positions in `keys` and `values` are monotonically increasing, and each key position is immediately followed by the position of the next entry. Instead of separately storing the length, it can be calculated by subtraction of two adjacent offsets. As the last such offset in the `values` array has no succeeding offset, `data_end` is added to allow calculation of the last length.
2. 32-bit offsets and lengths are unnecessarily wasteful considering the maximum node size of 4 MiB. Offsets are now encoded as 24-bit unsigned integers, which still allows for 16 MiB node sizes.
3. Instead of iterating the internal `BTreeMap` of a leaf node four times to create separate ordered sections of `keys`, `values`, and then writing all key data and value data into `data` in two passes, the current version only needs two iterations of the `BTreeMap` to write interleaved entry information, and fill the `data` array.

[1] A compact binary format for the serde (de-)serialisation framework: https://github.com/bincode-org/bincode

4. For the addition of tiered storage described in Section 5.6, a single byte of per-key information is added for every `entry`.

Despite the added functionality, this new representation is more compact: the metadata cost per entry has been reduced from $2 * (4 + 4) = 16$ bytes to $3 + 1 + 3 = 7$ bytes[2], resulting in per-entry savings of 9 bytes.

```
Layout:
    entry_count: u32,
    keys: [Data; entry_count],
    values: [Data; entry_count],
    data: [u8]

Data:
    pos: u32,
    len: u32,
```

Listing 6.1: Previous version, using 32-bit offsets and lengths

```
Layout:
    entry_count: u32,
    entries: [Entry; entry_count],
    data_end: u24,
    data: [u8]

Entry:
    key_pos: u24,
    key_info: KeyInfo,
    data_pos: u24

KeyInfo: storage_preference: u8
```

Listing 6.2: Modified version, using 24-bit integers and additionally storing per-key information

6.2. Direct IO

When files are not specifically opened with the `O_DIRECT` flag[3], the Linux kernel routes IO operations through its own page cache [Bovet and Cesati, 2005, p. 668-671]. When enabled, subsequent file operations are no longer performed in *buffered* mode, but as *direct IO*, which reduces the amount of buffer copies on the path from and to the underlying storage device, and bypasses the page cache logic.

If the application performs additional caching of the contents of a *buffered* file, as was the case with the B^{ε}-tree storage stack, this can result in data being cache twice. Not only does this double caching lead to unfair comparisons with single-cached systems, but the higher utilisation of the Linux page cache could even negatively affect other applications running on the system, which might be more reliant on the page cache. An internally managed application cache can take advantage of domain knowledge of cache entries, unlike the Linux kernel, which can not interpret the cached data and is unable to incorporate application-level information into its decisions.

In order to utilise direct IO, the following rules must be followed whenever the application interacts with a file with the `O_DIRECT` flag [Mendez and Lührs, 2019, p. 33-34]:

1. The disk offset of read/write operations must be an integer multiple of the configured block size, i.e. the application can not read/write from/to any arbitrary location of a file.

[2]Except for the last entry, where an additional 3 bytes are required for `data_end`
[3]Or that flag is afterwards set via `fcntl(2)`

2. The buffer length of read/write operations must be an integer multiple of the configured block size, i.e. the application can not read/write an arbitrary amount of bytes.
3. The source/destination buffer of an operation must be allocated at a memory location with an address that is an integer multiple of the configured block size.

The memory alignment requirement has been solved with a custom buffer type, which ensures correct alignment and only allows buffer sizes of integer multiples of the internal block size. The alignment block size is dependent on the used storage devices, and can be configured in some cases[4]. The only currently tested block size of the B^ε-tree storage stack is 4096 B, which would cause insufficient alignment for storage devices with a native block size of 8192 B. As the block size is only defined in a single compile-time constant, it appears likely that an adjustment could add compatibility with 8192 B blocks sizes, but this has not been tested. Until the B^ε-tree storage stack gains support for runtime configurable block sizes (like ZFS' ashift parameter), another workaround could be to disable direct IO on a per-vdev level, although the cost of aligning in-memory buffers is not elided when direct IO is disabled for every vdev.

6.3. Compression

Compression in the B^ε-tree storage stack occurs for each tree node, by running the encoded form of a node through a configurable compression algorithm. Compression is conceptually mandatory, but a compression algorithm called `None` simply returns the input data and thus performs no compression.

Whereas an object pointer previously encoded the compression algorithm into its type, every object pointer now carries information on the decompression step necessary to retrieve its original data, called a *decompression tag*. Currently, the decompression tag only encodes the originally used algorithm, the compression level is not required during decompression.

The Zstandard[5] compression algorithm has been integrated as a new compression algorithm with a configurable compression level, and is benchmarked in Section 8.2.3.4.

6.4. Synchronisation

Synchronisation refers to the action of ensuring that the logical content of a database corresponds to its physical counterpart on persistent storage, *synchronising* the blocks on disk to the value they should have. It is of particular importance in the presence of write caching, which can occur at multiple levels with the B^ε-tree storage stack, including the following:

1. The B^ε-tree storage stack will keep decoded and modified versions of database nodes in memory.
2. The operating system kernel or filesystem may separately cache write operations if used in an asynchronous mode (often the default).

[4] At least with certain NVMe SSDs, which support namespace reformatting with multiple logical block addressing formats.
[5] https://github.com/facebook/zstd

3. The underlying storage device can have further internal caching.

These layered caches are meant to improve system performance, and allow for short bursts of operations, with a higher throughput than would be possible for an extended duration. Simultaneously, because these caches often reside in volatile memory, they represent a risk of data loss, if e.g. a power interruption leads to a host shutdown. The tradeoff between safety and performance can be shifted by disabling one or more of these write caches.

As discussed in Section 6.2, the backing database files are now opened with the `O_DIRECT` flag, bypassing the operating system page cache in favour of the internal caching. Configuration of the internal caching behaviour of storage devices is left to the system administrator.

Tree locations To operate on a dataset, the location of its root node is required. This location is found by first following a block pointer from the most recent superblock to the root node of the root tree, and then retrieving the block address to the root node of the desired tree. It follows, that a newly opened database can only access changes of a dataset as old as or older than the most recent superblock.

During extended operation, the user was previously expected to periodically call `Database::sync`, in order to write out all pending changes, and finally refresh a superblock. Otherwise, the database will not be able to locate recent versions of its datasets, and no changes will appear to have been written.

Background synchronisation timer An optional background thread has been added, calling `Database::sync` in a configurable interval. It can be activated by calling `Database::with_sync`, and will use the synchronisation mode of the provided `DatabaseBuilder`. This sets an upper bound for the age of lost data, at the cost of additional IO load and locking parts of the database periodically. Appropriately timed explicit synchronisation calls are still possible, either in conjunction with or instead of the background timer.

Unfortunately, in its current state, the B^{ε}-tree storage stack always reallocates during synchronisation, even when no changes have been made, resulting in unnecessary IO load for the host system. Because this behaviour was only noticed after the development phase, no cause analysis has been conducted. It appears possible that the write-back functions either incorrectly copy an unmodified node, or that some entries of the root tree are indeed being changed, possibly unnecessarily.

6.5. Fuzzing

Fuzzing is a method for testing software by randomly generating input data until the process crashes, or an assertion fails [Liang et al., 2018]. The search for successful inputs is greatly accelerated by taking into account additional information about the executed code. Although fuzzing is not able to prove the correctness of a system, it is a highly useful tool for finding previously unconsidered edge-cases.

libFuzzer[6] is a fuzzing engine which guides input generation by observing changes to code coverage, and recombining promising inputs. With the help of cargo-fuzz[7] and Rust bindings for libFuzzer[8], two fuzzing targets have been added to the B^ε-tree storage stack, fuzzing for sequences of key-value and object operations which lead to a crash or cause an assertion to fail.

Although the original input provided by libFuzzer is a simple byte sequence, structured fuzzing input is supported, with many prebuilt transformations from byte sequences to normal Rust values. An example is shown in Listing 6.3, demonstrating the convenience of automatic structured fuzzing.

```
1  #[derive(arbitrary::Arbitrary)]
2  pub enum ObjOp {
3      Write(ValidKey, Vec<u8>, u64),
4      Read(ValidKey, u32, u64),
5      Delete(ValidKey),
6      ...
7  }
8
9  pub fn run_obj_ops(ops: &[ObjOp]) {
10     let mut db = setup_db(64);
11     let os = db.open_object_store().unwrap();
12
13     for op in ops {
14         use crate::ObjOp::*;
15         match op {
16             Write(ValidKey(key), data, offset) => {
17                 let (obj, _info) = os.open_or_create_object(key).unwrap();
18                 let _ = obj.write_at(data, *offset);
19             },
20             // ...
21         }
22     }
23 }
```

Listing 6.3: An abridged example of object store fuzzing, `ops` contains an arbitrary sequence of operations, which is processed in lines 12-21. If an error is encountered, the responsible input is collected.

Over the course of several days of fuzzing, numerous crashes have been detected, traced to previously unknown bugs and subsequently resolved. The located issues included edge cases such as integer overflows during chunk iteration when trying to iterate object chunks with byte addresses larger than 512 TiB, but also a lack of key-value pair size restriction, which would crash the B^ε-tree storage stack after failing to split a root node with only one entry.

Fuzzing also helped to narrow down the cause of changes to the tree structure that caused crashes later in the execution, by finding minimal sequences of tree operations which would

[6] https://llvm.org/docs/LibFuzzer.html
[7] https://github.com/rust-fuzz/cargo-fuzz
[8] https://github.com/rust-fuzz/libfuzzer

still trigger a crash. This led to the reimplementation of `range_delete` with naive message insertion, after fuzzing revealed that it could leave the tree in invalid states due to lack of rebalancing functionality.

6.6. Memory vdev type

Although a storage system is usually configured to use persistent storage, there are cases where its use is inconvenient: automatic testing and fuzzing. Concurrent access to the same backing storage is not supported in the B^ε-tree storage stack, each instance assumes exclusive access. This prevents the parallel execution of multiple instances with the same configuration, but assigning each concurrent instance a unique storage configuration would solve this problem. However, using physical persistent storage still unnecessarily incurs the overhead of IO with the underlying storage device, and causes additional wearout.

By introducing an in-memory vdev type, the B^ε-tree storage stack can be configured to write to a memory buffer instead of files/block devices. This provides each instance with exclusive storage access, and is faster than actual file IO, allowing for more fuzzing executions over a given timeframe. The size limitation of in-memory vdevs is not a problem in practice, as neither tests nor fuzzing require large capacities in order to function.

The implementation is comparatively simple: a buffer of the configured size is allocated and read/write operations are implemented by copying into or out of buffer regions. To accommodate concurrent access, the entire buffer is read-write-locked on every operation.

6.7. Layered configuration

Previously, only a few parameters of the B^ε-tree storage stack could be configured, by passing separate arguments while constructing a `Database`. This included the storage configuration to specify which top-level vdevs to use, the cache size, and whether to ignore the database's previous content. Parameters like the utilised compression method or level, the capacity of the writeback queue, or the size of the thread pool used for asynchronous IO could not be changed by the user without editing the code.

The database initialisation procedure has been altered to allow user-supplied code to provide different components, such as a configured storage pool, or the root tree. As this flexibility is usually not needed, a default implementation provides the required components based on a nested configuration structure called `DatabaseConfiguration`.

Although this allows an embedding application to configure the B^ε-tree storage stack, the application-side construction of a `DatabaseConfiguration` still needs to be made user-configurable. This is accomplished with the figment[9] library, which supports the construction of a nested key-value document from multiple *providers*, which can be converted into a `DatabaseConfiguration` if the format is correct.

These providers can be ordered, so that values configured via a higher-priority provider override those of lower-priority providers. Two command-line applications developed in the

[9] https://github.com/SergioBenitez/Figment

course of this thesis[10] use the following providers (listed in an ascending order of priority): a provider with a sensible default configuration, a JSON file located via the environment variable `BETREE_CONFIG`, and finally environment variables beginning with `BETREE__` and hierarchically delimited by `__` (double underscore). Examples for both configuration methods are displayed in Listing 6.4 and Listing 6.5. The use of `bectl config print-active` is recommended when using environment variables. The JULEA backend instead relies only on a JSON file specified through the JULEA configuration of the same format, and ignores environment variables for consistency with other JULEA backends.

Because it would improve the developer experience if different embeddings of the B$^\varepsilon$-tree storage stack used similar defaults and environment variables, two preconfigured figment providers for constructing a `DatabaseConfiguration` are available via `DatabaseConfiguration::figment_default` and `DatabaseConfiguration::figment_env` if the library is compiled with the `figment_config` feature.

```
{
  "storage": {
    "tiers": [
      [ "/dev/disk/by-id/nvme-CT500P5SSD8_20512BF90C84" ],
      [ "/dev/disk/by-id/ata-WDC_WD30EFRX-68EUZN0_WD-WMC4N2195306",
        "/dev/disk/by-id/ata-WDC_WD30EFRX-68EUZN0_WD-WCC4N3RS58KK" ]
    ]
  },
  "compression": { "Zstd": { "level": 3 } },
  "access_mode": "OpenIfExists",
  "metrics": { "enabled": true, "interval_ms": 500, "output_path":
    ↪ "/tmp/betree.json" }
}
```

Listing 6.4: Example JSON configuration

```
# Allocate storage preferences 0 and 1 on class 1, and fail for 2 and 3
BETREE__ALLOC_STRATEGY='[[1],[1],[],[]]'
# use only a 4GiB in-memory vdev as storage on class 0
BETREE__STORAGE__TIERS='[ [ { mem = 4294967296 } ] ]'
# use 2GiB for caching tree nodes
BETREE__CACHE_SIZE=2147483648
# enable Zstandard compression at level 3
BETREE__COMPRESSION='{ Zstd = { level = 3 } }'
```

Listing 6.5: Examples of configuration via environment variables

[10] `bectl` and `bectl-perf`

6.8. Cache size accounting

An existing bug in the B^{ε}-tree storage stack was revealed during the initial development of a scenario for the evaluation in Section 8.2.3.2. The scenario involves a large object being written twice, each time with different storage preferences. The metrics displayed in Figure 6.1 (top) indicate that the first write completes at 1:41, while the second rewrite occupies a much larger timespan (up until 51:00, when it was cancelled), writing vastly more data than expected in the process.

Intensive debugging revealed that the act of unpacking a leaf node from its disk representation was not accounted in the cache size, which is a single atomic counter cooperatively managed by every user of the cache, including the tree layer. Soon after beginning the overwrite, the cache size accounting has leaked considerable virtual cache space, eventually exceeding the configured maximum cache size (visible in the second plot as the red line growing unboundedly). When this happens, the cache eviction logic will always decide to evict its entries, even if that would empty the cache completely, leading to every node inserted into the cache to be immediately evicted afterwards. In order to empty the cache, any modified entries must be written back to disk, which triggers a catastrophic feedback loop: if the node was modified during or before write back, it is considered *stolen* and needs to be marked as deleted because it is superseded by the newly modified node. This deletion queues a message into a global buffer which will eventually perform the necessary change in the allocation bitmaps of the root tree during the next *sync*. Unfortunately, due to the misaccounted cache size, the modification of the allocation bitmap is also immediately evicted and written back, only to be fetched again to process the next queued deallocation message.

This caused the cycle of reallocation visible in the first plot after 1:41. Although this behaviour has been eliminated, it could reoccur whenever another cache size accounting bug is introduced.

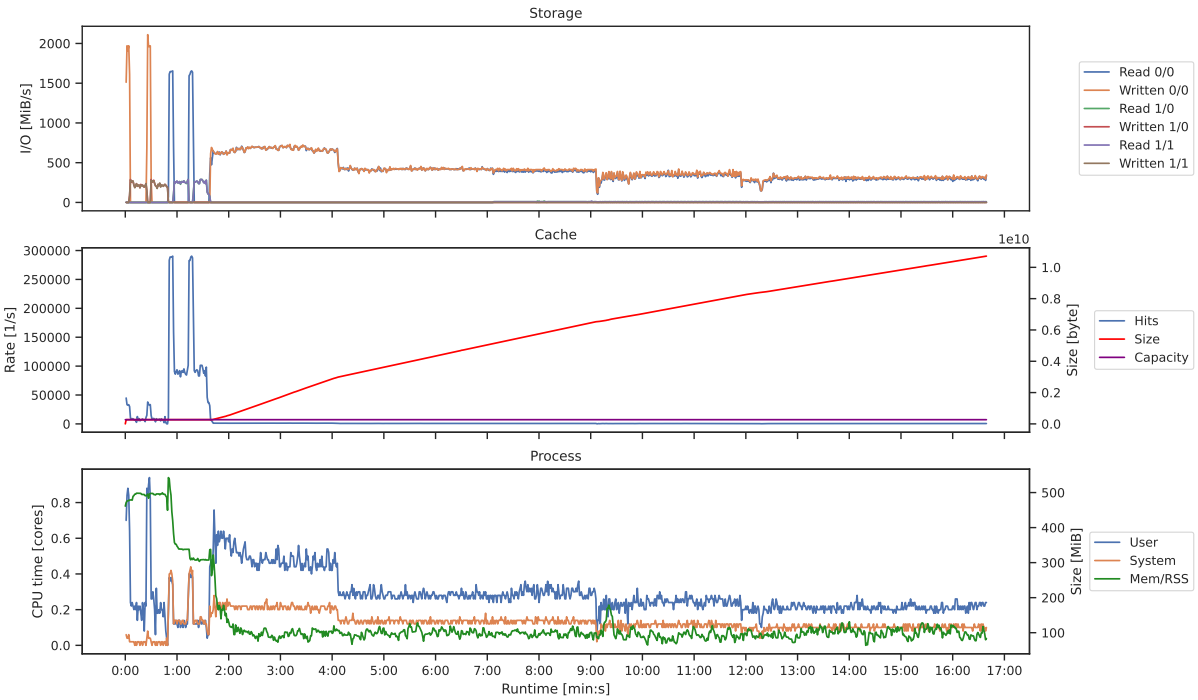


Figure 6.1.: The plotted data has been down-sampled, and reduced to the first 1000s for readability. The actual test ran for 51min, and wrote over 464 GiB before manual cancellation. Individual plots and notation are explained in Section 8.2.

Chapter 7.

Applications

The B^{ε}-tree storage stack is primarily a Rust library, meant to be embedded into applications which can use the provided functionality either internally, or may choose to expose an arbitrary subset of operations to the user.

A subset of the new aspects of the Rust API has already been described in the preceding chapters, and this chapter will instead describe specific integrations of the library which have been implemented as a part of this thesis.

7.1. bectl

In the absence of other integrations, there is no method for the user to directly interact with the B^{ε}-tree storage stack or any databases created with it. Even though realistic usage of the library is likely to use the Rust API or another integration, a more direct access can be useful during development or deployment.

bectl (a shortening of B^{ε}-tree-control) was created early in the familiarisation process to fill this gap, exposing many of the available operations in the form of a command-line application. Even in conjunction with more sophisticated integrations, bectl can remain a useful debugging and diagnosis tool. Its functionality is organised with nested subcommands (discoverable via integrated help output), and an example usage is shown in Listing 7.1.

The commands are grouped into four top-level categories:

Configuration
: The currently active and default database configurations can be listed, which can be useful to verify expectations against the actually used configuration. which is comprised of a configuration file and environment variables, as described in Section 6.7.

Database Management
: Besides forcing a database reinitialisation, resulting in the loss of all stored data, bectl can also display the current database superblock for diagnosis purposes, and list the entries of the root tree.

Key-value interface
: Basic manipulation of key-value pairs is supported, by generating a full listing of the entries of a specified dataset, and allowing the user to query, insert, or delete additional entries. Additionally, there is a subcommand for outputting the abridged tree structure

introduced in Section 8.1.1, to help gain an intuitive understanding of how different operations affect the B^ε-tree.

Object interface
: Similarly to the key-value interface, the object subcommands allow listing of objects (incl. metadata), as well as the overwriting, deletion, renaming, and reading of arbitrary objects.

This approach of creating a new process to accomplish a single purpose is inferior to a client-server structure in terms of efficiency, as the constant overhead of opening and closing the database is incurred for every operation, and in-memory caching is limited to the comparatively short lifetime of only one operation.

7.2. JULEA

As previously described in Section 2.4, JULEA abstracts over three different families of storage interfaces. Although the existing B^ε-tree storage stack functionality could be used to implement a key-value backend, only an object store backend has been implemented. This section introduces a JULEA backend implemented on top of the object-storage extension of the B^ε-tree storage stack.

7.2.1. Choice of implementation language

Because the JULEA libraries and included backends are exclusively written in the C programming language, and the B^ε-tree storage stack already had a C interface, an initial prototype backend was written in C as well. While the resulting backend was functional, there were a few disadvantages to this choice:

1. The JULEA build process would have to either build the B^ε-tree storage stack libraries itself, resulting in a dependency on the Rust toolchain, or the libraries would have to be provided by the user during compilation.
2. The B^ε-tree storage stack C interface results in another layer of error translation
3. Support for multiple object namespaces requires careful locking behaviour, to support parallel access by different JULEA threads.
4. Because the layout of B^ε-tree storage stack type definitions is not guaranteed to be stable, and is not intended to be exposed to the API consumer, the C interface returns pointers to opaque types. This results in otherwise unnecessary allocations and pointer dereferencing.

A JULEA backend has the form of a dynamically loadable library, which must export a function `backend_info`, which returns a `JBackend`. This return value describes the backend capabilities and holds the function pointers corresponding to each backend operation.

This contract can also be fulfilled by a Rust library, if compiled with the crate type `cdylib`[1]. The implementation of the backend in Rust addresses the disadvantages identified above:

[1] For "C dynamic library"

1. Each project only requires a single toolchain, either for C or Rust, but not both. A deployment of JULEA with the B$^\epsilon$-tree storage stack backend still requires both.
2. The backend can directly use the Rust interface, and only needs a single layer of error translation.
3. Rust's standard library, and external libraries are available to avoid implementing error-prone manual locking.
4. A level of indirection is saved, as the Rust interface allows for visibility restriction on structures. The lack of layout stability guarantees are not relevant, because the B$^\epsilon$-tree storage stack is linked into the backend library.

7.2.2. Implementation

In order to conform to the C API expected for backend libraries, the signatures and type definitions must be made available to the Rust compiler. Calling back into JULEA's tracing facilities also requires the declaration of the involved functions. A narrowly selected subset of these definitions is automatically translated from header files by means of `rust-bindgen` and a build script[2]. The resulting crate `julea-sys` is then used by `julea-betree`. Despite the conventional "-sys" suffix, `julea-sys` does not link against any JULEA libraries.

With these bindings, each backend function has been implemented similarly to the example shown in Listing 7.2. Each backend function needs to reinterpret the untyped pointers (lines 7-9), before using them to call the corresponding object storage functions (line 13). Additionally, operation hierarchies and durations are made transparent to JULEA by calling into its tracing facilities (line 11). Finally, the resulting object handle is written into the output pointer, any errors are logged, and the presence of an error is communicated by returning a boolean (line 18).

After the initial implementation, the JULEA project added iteration capabilities to the object backend interface on suggestion[3]. This functionality has been implemented by wrapping the existing Rust iterator API.

7.2.3. Assessment

Once compiled, the `julea-betree` library has to be registered with JULEA by copying to or creating a symbolic link in the backend library directory (e.g. as `libobject-betree.so`). JULEA can then be instructed to use the backend by specifying `backend=betree`, and the backend path argument specifies the location of a JSON file following the format described in Section 6.7. The newly created backend passes the JULEA object tests and benchmarks without errors.

The JULEA backend interface has very limited support for error handling, using only a boolean return value to communicate to the caller whether the backend has encountered an error. Neither JULEA itself, nor the application calling into JULEA, can differentiate between or react to different causes for a negative (false) return value. For example, a read failure due to a non-existing key, database corruption, database misconfiguration, or an operating system error

[2] https://doc.rust-lang.org/cargo/reference/build-scripts.html
[3] https://github.com/julea-io/julea/issues/91

all appear identically to the caller, with no means of reacting to different error conditions. This information is available to `julea-betree`, and could be translated into a richer error type if the JULEA signatures are updated, but currently the error is only logged and reduced to a boolean value.

During shutdown of the JULEA server, or when a client-mode process exits, each JULEA backend is unloaded. This causes complications because `julea-betree` registers an exit handler with glibc to clean up thread-local storage, and attempting to execute this exit handler will result in a segmentation fault if the backend has already been unloaded. There seems to be no official way of deregistering such an exit handler, and the thread-local value is not directly accessible because it is created from within a dependency of the B^{ε}-tree storage stack.

As the operating system will clean up the process' resources after exiting, it is not necessary to unload the modules individually. A mechanism for backend modules to opt-out of being unloaded has been suggested to the JULEA project[4], but the changes are still pending approval at the time of writing.

[4] https://github.com/julea-io/julea/pull/95

```
$ export BETREE_CONFIG=/var/lib/betree/config.json
$ bectl config print-active
DatabaseConfiguration {
  storage: StoragePoolConfiguration {
    tiers: [
      TierConfiguration {
        top_level_vdevs: [
          Leaf(File("/dev/disk/by-id/nvme-CT500P5SSD8_20512BF90C84",),),
        ],
      },
    ],
    queue_depth_factor: 20,
    thread_pool_size: None,
    thread_pool_pinned: false,
  },
  alloc_strategy: [ [ 0, ], [ 1, ], [ 2, ], [ 3, ], ],
  compression: None,
  cache_size: 268435456,
  access_mode: OpenIfExists,
  metrics: None,
}
# double underscore is used as a separator for configuration paths
$ export BETREE__COMPRESSION='{ Zstd = { level = 6 } }'
$ bectl config print-active
  ...
  compression: Zstd(Zstd { level: 6, }, ),
  ...
$ bectl db init # reset the database, deleting all data
# create a new object "large_object" in the namespace "example_namespace"
$ bectl obj example_namespace put large_object < large_file
$ bectl obj example_namespace list
large_object (5368709120 bytes, modified 2021-05-29T20:13:15.259454+00:00)
```

Listing 7.1: Example usage of bectl, showcasing the layered configuration system. In the first display of the currently active configuration, compression is still disabled, but after overriding that particular setting with an environment variable, the compression configuration has changed. Afterwards, an object is created and its metadata displayed.

```
unsafe extern "C" fn backend_create(
    backend_data: gpointer,
    namespace: *const gchar,
    path: *const gchar,
    backend_object: *mut gpointer,
) -> gboolean {
    let backend = &*backend_data.cast::<Backend>();
    let ns = backend.ns(CStr::from_ptr(namespace));
    let key = CStr::from_ptr(path);

    let (obj, _) = jtrace::with(J_TRACE_FILE_CREATE, path, || {
        let obj = ns
            .create_object(key.to_bytes())
            .map(|(handle, _info)| handle);
        (obj, (0, 0))
    });

    return_box(obj, "create object", backend_object)
}
```

Listing 7.2: As an example for the implementation of a backend operation, `backend_create` converts the incoming raw pointers back into Rust references, then calls the corresponding object creation function from the public B^{ε}-tree storage stack API, while hooking into JULEA's tracing system. The result is written into `backend_object` via `return_box` as a heap-allocated `ObjectHandle`, and the absence of an error is indicated by a return value of `true`.

Chapter 8.

Evaluation

In absence of long-term production deployments of the B$^\varepsilon$-tree storage stack, the merit of the previously discussed changes can be difficult to ascertain, including their correctness, performance, and suitability to the implementation of realistic applications. This section evaluates the correctness and standalone performance of the system. A comparison to suitable alternatives has not been conducted, as a fair comparison necessitates familiarisation with the internals of each solution in order to adjust their tuning parameters to the given scenario.

8.1. Object store functionality

During and after the initial implementation of the object store extension, correctness was tested with the following methods, in addition to manual verification with the *bectl* debugging tooling described in Section 7.1.

8.1.1. Integration tests

Although many individual components of the B$^\varepsilon$-tree storage stack are tested in isolated unit tests, there are complicated interactions which can only occur in situations where multiple abstraction layers are used at once. For example, even if the splitting operation of internal nodes is unit-tested, it does not cover the special case of splitting a tree's root node, an operation which relies on the presence of the data management layer.

Integration testing is an approach to software testing where the isolation between each component from unit testing is lifted, and the system is tested in larger groups, or as a whole. In contrast to unit tests, the failure of an integration test does not necessarily give an indication of where the issue is located, but by testing multiple layers at once, it allows fewer tests to check more of the system's behaviour.

Due to the nested nature of a tree data structure, and the complexity of B$^\varepsilon$-trees after even simple operations, accessing and comparing the properties of each tree node to their supposed values would result in lengthy and difficult-to-maintain testcases. Additionally, these testcases could be invalidated by minor changes to the tree logic, e.g. a different splitting threshold could restructure the entire tree. Instead of fetching and comparing each individual property, a hierarchical JSON object is constructed from the B$^\varepsilon$-tree, which can then be compared to a stored version in a separate file, similar to the example shown in Listing 8.1. This workflow

is facilitated by an existing library and command-line utility: insta and cargo-insta[1]. The generated JSON document consists of an abridged representation of the tree shape, which includes node types, storage preferences, and key ranges. This document additionally contains the keys of the data and metadata datasets (but not their values). Although this approach does not address the fragility of testing for the tree structure, it does reduce the effort involved in updating the test suite when testcases are failing incorrectly.

```rust
#[test]
fn insert_single() {
    // a TestDriver with 1 storage class, backed by a 256MiB in-memory vdev
    let mut driver = TestDriver::setup("insert single", 1, 256);

    // each checkpoint compares the tree state to a JSON document on disk
    driver.checkpoint("empty tree");
    // insert 2000 * 8192 bytes of random data as object "foo"
    driver.insert_random(b"foo", 8192, 2000);
    driver.checkpoint("inserted foo");

    for _ in 1..=3 {
        driver.insert_random(b"foo", 8192, 2000);
        // intentionally same key as above, to assert that tree structures is
        // not changed by object rewrites of the same size
        driver.checkpoint("inserted foo");
    }
}
```

Listing 8.1: Example test case, demonstrating a checkpointing system which avoids manual assertions

8.1.2. JULEA test suite

The JULEA project provides its own test suite, and a benchmarking utility. In conjunction with the B$^\varepsilon$-tree storage stack JULEA object backend described in Section 7.2, both of them can be used as an additional means of verifying the basic functionality of the system to some extent.

Covered by the JULEA test suite are basic object operations, such as creation and deletion, reading, writing, and metadata queries, but as JULEA is unaware of the tiered storage functionality, it can only utilise the default storage class, and is unsuited to testing that component. During development, the test suite indicated not only multiple mismatches in the semantics of an operation as understood by JULEA, but an important bug in object opening as well.

After resolving these issues, the B$^\varepsilon$-tree storage stack backend passes both the JULEA test suite and the benchmarking utility. A performance comparison of the new backend according to the JULEA benchmarking suite is provided in Appendix A.

[1] https://github.com/mitsuhiko/insta

8.2. Performance

This section examines the performance of the final B$^\varepsilon$-tree storage stack, with all the previously discussed extensions and modifications. After establishing an approach to data gathering and describing the precautions taken to achieve reliable measurements, multiple usage scenarios are constructed, measured with varying parameters, and the results assessed.

8.2.1. Measuring approach

Measuring the performance of a system is an error-prone process, as the necessary observation of the system could affect the measurements themselves. Precautions must be taken to limit any external influences on the measured application. Instead of gathering all feasible information, only a subset relevant to the analysis is gathered.

An aggregation of metrics in files of newline-delimited JSON has been chosen over alternative modes[2] of exposing metrics, for its flexibility, simplicity and ease of implementation. Individual steps of the data gathering process can be expensive, measurements of up to 20 ms were observed during testing. To limit interference with the measured process, as much information as possible is gathered by separate processes, converted to JSON and written to a file separately.

These individual JSONL (newline-delimited JSON) files are only merged into one file after the sensitive measuring process is completed, and finally presented visually with the Python plotting library Matplotlib.

8.2.1.1. Selected metrics

B$^\varepsilon$-tree storage stack
: The following metrics are reported directly by the B$^\varepsilon$-tree storage stack, with a configurable interval.

  Cache Monotonic counters for cache events, such as hits, misses, evictions, and the current cache capacity and size.

  Storage Grouped by storage class, each vdev reports monotonic counters of the amounts of blocks read and written so far, the number of failed operations, and any checksum error encountered.

Process information
: The reported cache statistics are based on internal accounting, which does not include other uses of memory, such as compression contexts or buffers currently in the writeback queue. To capture these other sources of memory usage, and to obtain a definitive upper bound on memory usage[3], the operating system is queried for process statistics.

  The operating system does not distinguish between memory usage by purpose, and can help confirm the internal statistics. Additionally to the memory usage (resident set size),

[2] A Prometheus monitoring endpoint was briefly considered and implemented, but finally rejected for the additional complexity of serving and gathering its metrics, and the requirement for global state by the selected library.

[3] In case the internal cache statistics are wrong, as in Section 6.8

the gathered process statistics include the CPU time spent in the kernel and in user mode, as well as minor and major page faults.

System information
To account for thermal throttling of storage devices and the CPU, a separate process periodically gathers the current temperatures of all hardware sensors via *libmedium*[4], which reads and interprets the pseudo files exposed by the Linux kernel's `/sys/class/hwmon` interface.

Timestamp
In order to correlate and merge different JSONL streams, the current time is recorded as the number of milliseconds passed since the UNIX epoch (1970-01-01 00:00).

8.2.1.2. Measures taken to improve test conditions

Various factors can result in unreliable or incorrect measurements, such as:

Thermal Throttling CPUs and storage devices can exceed their intended operating temperature, and throttle their performance to avoid an emergency shutdown or hardware damage. The recorded metrics contain the temperatures of all hardware sensors, which can be used to determine if any devices reached their thermal throttling threshold.

Between every test run, the system is allowed to cool down for a period of at least 70s, and the final test runs were all taken with a similar ambient temperature. After observing high temperatures in the SSD, the GPU fan has been set to a constant speed manually, because the SSD is within the generated airflow.

Unrelated system load To prevent other background processes from interfering with the measurements, any active or scheduled system services are temporarily deactivated, and the final measurements are taken with the graphical interface of the workstation deactivated.

SSD flash translation layer (FTL) trimming When using SSD storage, the entire device is marked as discarded with `blkdiscard`, so that each execution begins without any preallocated blocks. Otherwise, the FTL might behave differently between executions, choosing different algorithms or parameters for block allocation, or garbage collection.

Choice of data sources Most conceivable scenarios need data to write into a dataset or object store, but the choice of data can drastically alter the measured results.

When choosing a file from a local file system as a data source, the B^{ε}-tree storage stack might be kept waiting by a slower file system, thus limiting its performance.

If the compression feature is enabled and the chosen data is highly compressible, gigabytes of raw input data could be shrunk to mere megabytes. Additionally, the storage device itself might detect and react differently to specific patterns, such as data consisting only of zeroes [Zuck et al., 2014], or new data matching the current content of other blocks [Kim et al., 2012]. If this is unintended, the measurement would capture an entirely different system bottleneck, and are unlikely to be representative for realistic usecases. The other opposite is uniformly distributed random data, which is statistically

[4] https://gitlab.com/Maldela/libmedium, version 0.5.5

highly unlikely to be compressible, and would result in wasted compression attempts. Not all sources of random data provide sufficient generation speeds[5], but as cryptographic use is not intended, Xoshiro256Plus is chosen as a sufficiently fast algorithm [Blackman and Vigna, 2018].

8.2.2. Hardware information

Each of the following measurements were taken on a local workstation with an AMD Ryzen 3600 CPU, 32 GiB of dual-channel memory[6], running NixOS 21.05 on Linux 5.9.16 with the following storage configuration unless specified otherwise:

1. A new Crucial P5 500 GB SSD (CT500P5SSD8) is used for storage class `0`, attached via NVMe over PCIe3x2. It is running firmware version P4CR324, and only presents a single logical block addressing format option of 512 B to the host system.
2. Two Western Digital Red 3 TB HDDs are used as top-level vdevs of storage class `1`, so that allocations are shared evenly between the two drives and future read/write operations can utilise both at the same time.

   The model of each drive is WD30EFRX-68EUZN0, a 5400 RPM HDD intended for network-attached storage, and they are attached to the ASMedia ASM1061 controller via SATA 3. Both drives report an operating time of approximately five years, and use the firmware versions 80.00A80 and 82.00A82.

8.2.3. Scenarios

The following scenarios exercise both the new tiered storage functionality and the object store abstraction to various degrees. Although they can not replace proper benchmarking with a real application when determining the suitability of the B^{ε}-tree storage stack for a particular purpose, they serve to give a rough understanding of performance with varying parameters, such as cache size, allocation strategies, and parallelism.

When referring to a specific storage device, the following plots use the notation `c/v` to refer to the `v`th top-level vdev of storage class `c`. For example, in the default configuration above, `0/0` refers to the SSD, while `1/0` and `1/1` refer to the two HDDs.

When a figure contains multiple plots, they are horizontally aligned by the elapsed real time. Although the measurements of both HDDs are recorded and plotted separately, their plots often align very closely, overlapping in many places. The total read/write speed of each class is the sum of their individual measurements.

[5] With a kernel configured to use RDRAND for `/dev/urandom`, output speed can be as low as 66 MiB/s on certain Ryzen CPUs.

[6] M391A2K43BB1-CTD, 2Rx8 ECC UDIMM, CAS CL 19

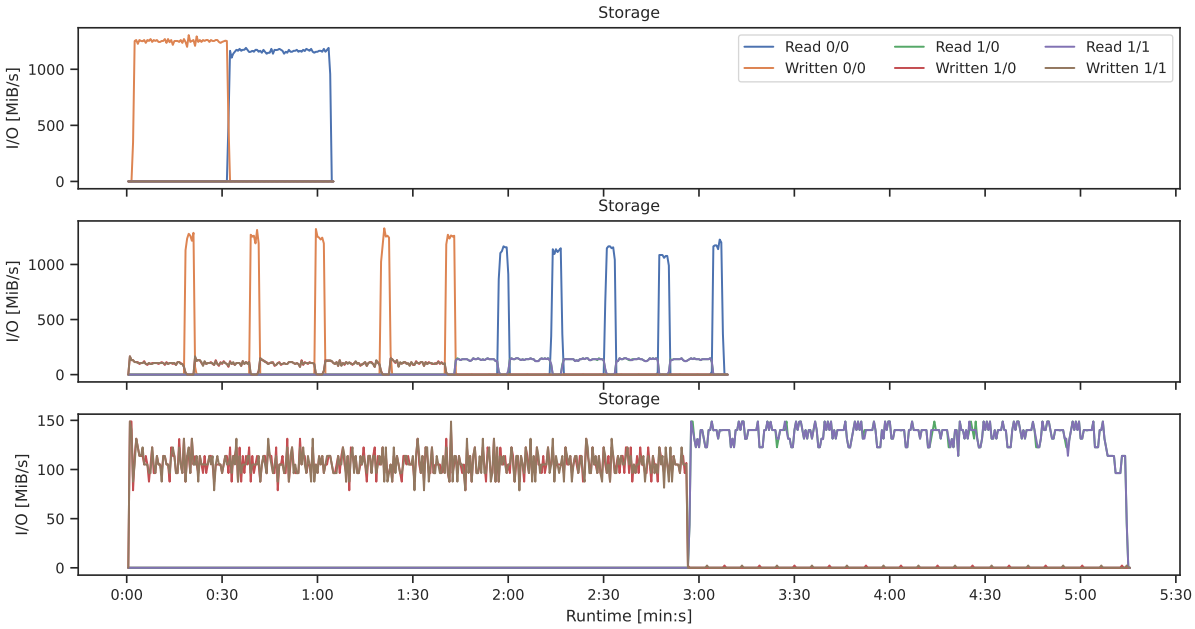


Figure 8.1.: Three different executions of the scenario introduced in Section 8.2.3.1, showing the read/write progress made on each configured storage device: the SSD on class `0` (`0/0`), and the two HDDs on class `1` (`1/0`, `1/1`).

8.2.3.1. Cross-class object

This first scenario serves to establish two baselines for homogeneous object storage on each of both available storage classes, then measure the overhead of heterogeneous allocation in a single object.

A single thread creates five objects in the same object store and sequentially fills each with 5 GiB of random data. The first 2.5 GiB are written with a `StoragePreference` of `1`, whereas `0` is specified for the second half. Afterwards, each object's data is read completely in the original creation order of the objects.

The scenario is executed with the following three different allocation strategies. As explained in Section 5.3, an allocation strategy specifies the storage classes used to fulfil preferences for each class:

1. `[[0],[0],[],[]]` will allocate requests for `0` and `1` on class `0`, and fail requests for `2` or `3`.
2. `[[0],[1],[],[]]` will allocate requests for `0` and `1` on the requested class, and fail other requests.
3. `[[1],[1],[],[]]` will allocate requests for `0` and `1` on class `1`, and fail other requests.

The read/write performance of each execution is shown in Figure 8.1. The upper and lower plot display the homogeneous operations on the storage classes `0` and `1`, respectively, while the operations in the middle plot alternate after the middle point of each object.

Notably, because of the even split between storage classes, and each logical byte being written and read exactly once[7], the completion time of the second configuration with mixed allocation is very close to the arithmetic mean of the other two configurations: $(5min + 15s + 1min + 5s)/2 = 3min + 10s$, suggesting that the fixed overhead of tiered storage in this scenario is negligible.

8.2.3.2. Storage class inversion

Whereas the previous scenario only involved static class assignment, without any changes after each node was written, this scenario stresses the database by inverting the storage classes of a multi-segmented object: each part of the object is written to classes `0` and `1` alternately, read sequentially, and then overwritten again with an inverted alternation, so that every part switches to the respectively other class.

Among the multiple segment counts and sizes tested, the configuration shown in Figure 8.2 had the longest execution time, with four segments of 8 GiB of randomly generated data each. The first write phase (0:00-2:08) lasts only 128 seconds, as new tree nodes are filled with data without having to wait for input. Conversely, the second write phase first needs to fetch nodes from disk in order to insert messages into them[8], resulting in a 75% longer duration of 225 seconds. The first and second read phase have very similar durations at 119 and 120 seconds respectively, suggesting that there is no residual cost to a storage class change.

From smaller-scale testing, the last node is expected to remain on its previous storage class, because the upper layer may retain the last buffered messages necessary to replace its contents, yielding a non-uniform storage preference until they are flushed.

Whereas the CPU temperature matches the expected lower utilisation during read phases, the development of the SSD temperature reveals that it only cools down significantly while the test is reading from class `1`, but notably not while writing to it. Because the measurements indicate no SSD write activity during that time, and only negligible read activity, the cause for the sustained temperature is unclear. The unexpected heat could potentially be explained by an internal write cache being flushed after the large write bursts before, or other internal flash controller processes.

The third plot contains the memory usage of the entire process as reported by the operating system, as well as the CPU time spent in either kernel or user mode. While the total memory usage remains reasonably close to the configured cache size of 256 MiB during read phases, an increased memory usage is observed in write phases due to the buffers stored in the writeback queue, which can only be freed after they have been written to disk. The rapid insertion of messages during the first write phase is the most computationally intensive section, particularly while writing to the SSD, which can accommodate much higher rates of insertions. Whereas writing system calls can return to the calling thread after the operation has been handed off to the SSD's cache, reading system calls need to wait until the operation is fulfilled before returning control to the caller, resulting in the larger system time usage during SSD read cycles.

[7] Neither condition is necessarily realistic, many workloads are either write- or read focused, and would likely divide the object into uneven pieces due to an underlying structure of the object's data.

[8] Although the cost of such situations is usually lessened by message buffering, a large sequential workload will soon trigger a flush of the message buffer, necessitating the fetching of the corresponding child node for message insertion.

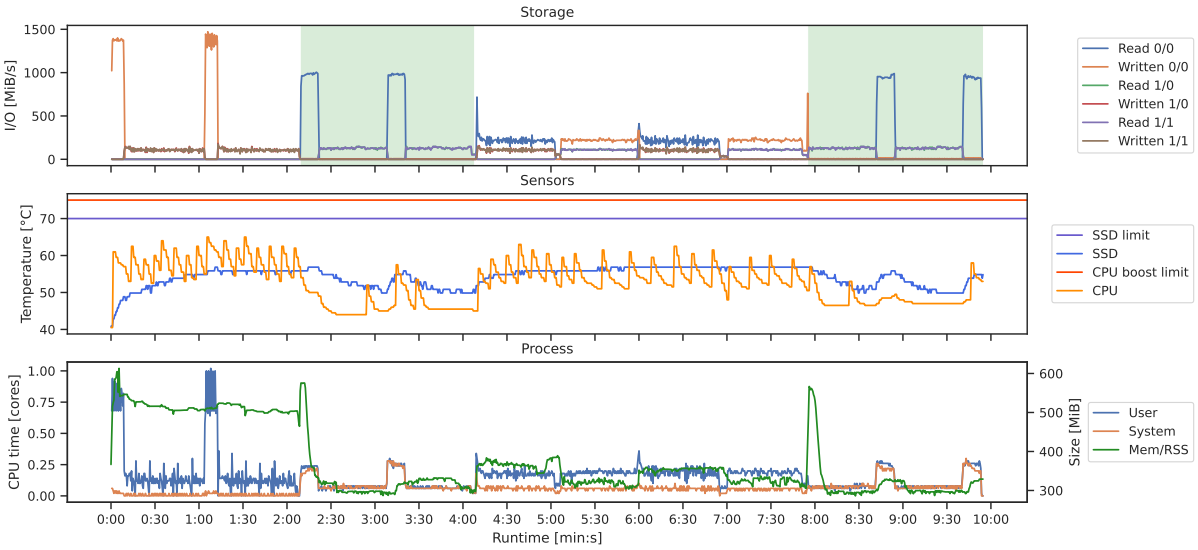


Figure 8.2.: An I/O, temperature, and process information plot, displaying the four phases of the storage class inversion scenario. The two sequential read phases are designated by the green background sections at 2:09-4:08 and 7:55-9:55.

8.2.3.3. Prioritised zip index

In contrast to the previous sequential workloads, this usecase is dominated by (parallel) random reads: a specially constructed zip archive is written to a newly created object, followed by the extraction of 10000 randomly[9] selected files contained in the zip archive. This random read workload is too unpredictable to benefit from readahead, and largely relies on caching as much of the archive as possible.

A zip archive contains a central directory of directory and file entries, which is essential for locating the data of individual files [PKWARE, Inc., 2003]. The starting boundary of this central directory is located manually with `zipinfo`[10] and passed as a command-line argument during scenario invocation. Only the 9.3 MiB section past this boundary is allocated on storage class `0`, the remainder of the archive is allocated on class `1`. This uneven split is intended to accelerate the initial fetching of essential metadata, as the used zip implementation (zip-rs[11]) eagerly reads and decodes metadata for all 80690 entries of the archive before its contents can be queried.

To provide a size distribution and compressibility more realistic than a uniform synthetic workload, and to allow reproduction of these measurements, the sources of the Linux kernel[12] were selected and repacked as a zip archive. This is necessary, as the provided tar file (a tape archive, designed for sequential access) does not store its metadata in a manner suitable for efficient random access. Compression was disabled during creation (`zip -compression-method=store`), to increase the archive size and test the B^{ε}-tree storage stack compression in a later scenario. Additionally, due to the comparatively low size of 1011 MiB, a cache size smaller than the archive is essential in forcing uncached read operations. In a real-world setting featuring significantly larger datasets, the central directory of each archive would not remain cached as reliably as it is with this single archive. Although a cache size of only 32 MiB is

[9] But deterministically between executions
[10] Specifically `zipinfo -v data/linux-5.12.13.zip | grep -A2 'The central directory'`.
[11] `https://github.com/zip-rs/zip`, version 0.5.13
[12] Version 5.12.13, `https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.12.13.tar.xz`

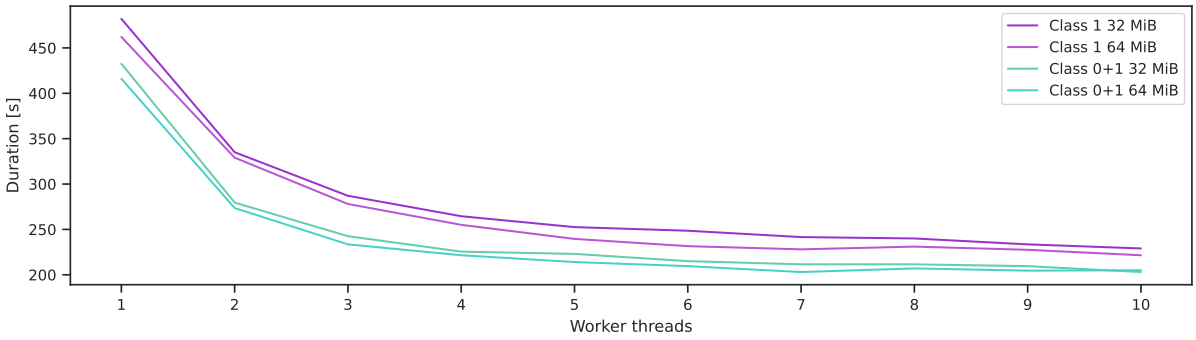


Figure 8.3.: End-to-end completion times of 10000 fetches of randomly selected files from a zip archive object, showing the influence of different allocation strategies, two cache sizes, and added parallelism up to 10 threads.

unreasonably low for any actual deployment, it is used here to force the cache misses that would be present with a larger dataset.

As can be seen in Figure 8.3, the largest end-to-end improvement from tiered storage is a 55s reduction in runtime when using 2 worker threads (regardless of cache size), which amounts to 16.7% of the runtime when tiered storage is not used. This improvement is disproportionately large, considering that only 9.3/1011 = 0.9% of the entire object is set to prefer allocation on the SSD. These results confirm that applications with non-uniform structured access patterns can benefit from setting different storage preferences for different sections of an object.

8.2.3.4. Compressed single-class object

In this scenario, a single file is written from a local filesystem into an object in a newly created object store, then read back by the application. The effects of different compression configurations are measured, starting with no compression, and then using the Zstandard compression algorithm integrated in Section 6.3 with compression levels from 1 to 16.

The zip archive described in Section 8.2.3.3 is used as the input file, due to its reasonable compressibility, and ease of reproducibility. Although it is possible for the reading from the local filesystem to have delayed the measurement, it can be reliably read in manual testing within 0.7s during a continuous read operation, which is far quicker than even the fastest read time measured.

Contrary to the other scenarios, only a single storage class is used here, which is comprised of a single HDD. The comparatively slow speeds of the HDD result in longer scenario execution times, and emphasise the benefits of the high compression ratio of the selected input file: the logical read/write speeds can exceed the maximum physical speeds of the drive, whereas the CPU would likely become a limiting factor with a much faster storage medium.

Figure 8.4 shows the read and write phase completion times of each compression configuration, as well as the total amount of data written to the HDD. Although this total includes the root and metadata B$^\varepsilon$-trees, superblocks, and overwritten data, and thus should not be considered the compressed size of the object[13], it provides an upper bound.

[13]The B$^\varepsilon$-tree storage stack does not currently support the concept of compressed object sizes or compressed key-value value sizes.

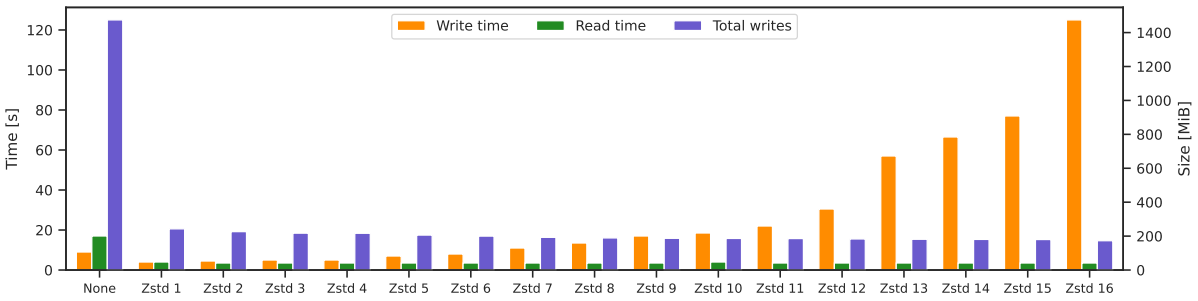


Figure 8.4.: Write/read phase durations and total data written to the storage medium per compression configuration. Zstandard is abbreviated as Zstd, and the following number indicates the compression level.

As expected given the highly compressible input file, the total data written is drastically reduced, which results in faster read/write phases by having to store/fetch less data. For this specific case, compression with Zstandard reduces the total duration up until compression level 11, after which the initial compression takes longer than the uncompressed test run. If it is known in advance that the object will be read multiple times, these higher compression levels could still be beneficial.

Chapter 9.

Summary

The B^{ε}-tree storage stack has been extended with an object storage interface, which maps operations for variably-sized objects onto the underlying key-value interface, and permits the association of metadata attributes with objects. This is accomplished while preserving the benefits of the underlying write-optimised B^{ε}-tree data structure, by matching the object semantics with a separately-defined metadata message type, and made available via a convenient and misuse-resistant object interface.

A tiered storage system has been developed specifically for B^{ε}-trees, which allows the aggregation of varied storage devices into four storage classes. The user can then specify storage preferences for each key-value pair, which translates to chunk-granularity when working with the object interface. When speaking of unstructured data, the referenced data is often not truly unstructured, but instead the storage system is not equipped to take advantage of that structure. While the B^{ε}-tree storage stack does not attempt to detect or exploit any structure within stored data by itself[1], it does allow the user to communicate a very rough structure in the form of storage preferences, which can lead to performance gains when the access patterns are guided by that structure, as demonstrated in Section 8.2.3.3.

In addition to correcting numerous existing issues, multiple improvements to the B^{ε}-tree storage stack include space savings in the on-disk representations of allocation bitmaps and leaf nodes, a new compression algorithm, and the usage of direct IO. The library has been integrated into the storage framework JULEA as a new object backend and passes all object-specific tests. The existing unit testing approach has been complemented with snapshot-based integration tests and two fuzzing targets.

Future Work

In addition to the remaining future work items of [Wiedemann, 2018], four areas for future improvement and extension have been identified, which can be further grouped into changes to the already existing storage stack, and improvements to the newly implemented functionality.

Multiple possible improvements pertaining to the core B^{ε}-tree data structure were discovered during debugging, as well as missing functionality initially assumed to be implemented:

[1] Except via the compression feature

Range delete For example, when deleting large contiguous ranges of key-value pairs, the `range_delete` operation used to efficiently discard entire tree nodes, without the usual insertion of deletion messages. Unfortunately, the necessary rebalancing and tree shrinking functionality was not fully implemented in the original system, and has been replaced with the inefficient insertion of deletion messages. A more efficient reimplementation would greatly improve deletion speeds of objects.

Direct insertion While inserting a message, the message is not always inserted into the root node. Instead, the B^ε-tree storage stack traverses the tree as far as possible using only cached nodes, without performing any disk IO. When inserting a full object chunk, the data is first copied into the message, and then out of the message into the leaf node immediately after, if the leaf node was cached. The copy into the message is necessary if the message outlives the **Write** operation, but an optimisation to the message interface could eliminate the first copy when inserting a newly constructed message directly into a leaf node. As the user-provided buffer is only valid during the **Write** operation, the message type must support the conversion into an independent message, which carries the buffer itself.

Early query abortion Currently, after the B^ε-tree storage stack encounters e.g. a deletion message during a point query, all other messages for that key are still fetched, even though they can not alter the final result. By extending the message interface to allow for a message to indicate that its application will override the effects of any other messages, point queries can be aborted early, and the nodes further down the access path may not need to be fetched from disk.

Opportunistic flush during queries While querying for a key-value pair, all the nodes along the access path must be loaded into memory. This opportunity could be used to gather and flush any messages intended for the final tree node.

Force-flushing Due to the peculiarities of the flushing process, groups of small messages (such as deletion messages) may remain buffered for a long period of time, delaying their effect (here the removal of the corresponding key-value pairs and eventually the containing leaf node) for longer than desired. A background process to gradually force a flush of all message buffers could solve this issue.

Insert batching A batched insert interface, to reduce some per-insert overhead if many new messages are to be inserted at once. Currently, the tree is traversed from the root node for each message insertion, but if many messages share a destination node, this effort could be reused.

Long-term performance The system performance over longer periods of time has not been investigated, tree ageing and fragmentation effects should be measured over extended usage.

The lower layers of the storage stack, the data management layer and the storage pool, can also be improved in several ways:

Space accounting Although the interface of the data management layer indicates that space accounting was planned, it is currently not possible to query the free space available on a given storage device. Proper space accounting is essential in balancing out uneven usage of storage devices, and the knowledge that a vdev no longer has any spare capacity

would greatly accelerate allocation failures (which currently have to scan all allocation bitmaps for that device).

Runtime-configurable compression The compression type used for a given object or namespace should be configurable. Per-object compression poses similar challenges as tiered storage, because compression is specific to a tree node, not a key-value pair.

Background compression Compression currently happens before adding the task to the background worker queue. If compression was offloaded to that thread pool, the **Write** operation could return control to client code without waiting for compression to finish.

Runtime-configurable block size The block size is currently statically configured. Although the B^{ε}-tree storage stack should support its compile-time reconfiguration, more work is necessary to support runtime-configuration, or heterogeneous block sizes.

Certain advanced usecases of the newly implemented object storage layer might become more efficient with the implementation of:

Configurable chunk size A method of specifying an alternate chunk size for an object could be beneficial not only to reduce key-value overhead for large objects, but also when much smaller query/update granularities than the default chunk size are typical. A metadata lookup for the chunk size for every **Write** operation would negate the write optimisation advantages of the B^{ε}-tree. The chunk size could be cached in the object handle if changes to the chunk size of an object are forbidden after its initial creation. Alternatively, the chunk size setting could affect an object storage namespace instead of individual objects.

Copy operation The **Copy** operation has not been implemented due to the complexity of an efficient data sharing approach. Although it is already possible for the user to perform a naive copy of all object chunks and metadata (even for sparse objects), the B^{ε}-tree storage stack could provide an optimised operation, possibly similar to the *copy-on-abundant-write* strategy described in [Zhan et al., 2020].

Applications utilising the storage stack with multiple storage tiers could benefit from the following two features:

Dynamic policy Dynamic migration policies could decide whether to move an object (or key-value pair) to a different storage class based on past access patterns, or statistics about similar objects. Such a policy could complement the user-provided preferences in cases where the user would otherwise reimplement similar access tracking.

Tier-specific node sizes While the large node size of the B^{ε}-tree is well-suited for spinning storage devices with long seek times, modern SSDs do not incur the same cost for random IO. Instead, the read amplification caused by the larger node sizes may outweigh any benefits of sequential IO. A partial solution to this was proposed in [Wiedemann, 2018], by splitting leaf nodes into smaller nodes, which can be fetched and interpreted individually. These smaller nodes could still be allocated contiguously and read completely on HDDs, to maintain the read throughput for sequential access.

Appendix A.

JULEA backend comparison

This comparison has not been performed with the same diligence as the primary evaluation in Chapter 8. As there are many unaccounted sources of unfairness, these results should be interpreted carefully only as a rough orientation, instead of a precise benchmark. The benchmarks are provided by JULEA[1] via the `julea-benchmark` tool. Only object-specific benchmarks are executed, each with a target duration of ten seconds. The batched write benchmark is executed with a custom block size of 128 KiB (the object chunk size) in addition to the default block size of 4 KiB.

All benchmarks were executed using only the SSD described in Section 8.2.2, as the tiered storage feature is not exposed to JULEA. Both the POSIX and GIO backends are used in a client-server configuration, as they do not support being loaded by the client directly, with an ext4 filesystem created using `mke2fs` with the default parameters and options[2]. The B^{ε}-tree storage stack backend was configured to use direct IO with a cache size of 2 GiB to match the availability of the Linux page cache of the other backends. It is tested both in a client-server and client-only configuration, and the compression feature is disabled.

Results The full results for each backend configuration are displayed in Table A.1. In comparison to the filesystem-based backends, the B^{ε}-tree storage stack backend performs well at object creation and fixed metadata queries, but worse at object deletion. This is likely related to the previously described workaround in the implementation of `range_delete`, and should improve drastically with an optimised reimplementation of that operation.

The client-server configuration appears to be limiting the performance of the B^{ε}-tree storage stack backend, as indicated by the much higher throughput measurements if executed in a client-only configurations, which avoids the networking overhead by loading the backend library directly into the client process. The high read throughput largely relies on in-memory caching.

Finally, the cost of small writes is unexpectedly significant, as shown by the difference in the measurements of the batched write benchmark with varying block sizes. Each write will independently traverse from the root towards the tree node closest to the target node, and insert a new message into that node. This process could be optimised with the **Insert batching** and **Direct insertion** future work proposals.

[1] Using commit `115d9e4a8493816c09e51b4c6d37993a9ce8f915`, which contains necessary changes for client-side loading

[2] Version 1.46.2 (28-Feb-2021), active features: `has_journal`, `ext_attr`, `resize_inode`, `dir_index`, `filetype`, `needs_recovery`, `extent`, `64bit`, `flex_bg`, `sparse_super`, `large_file`, `huge_file`, `dir_nlink`, `extra_isize`, `metadata_csum`

Benchmark	POSIX	GIO	B^{ε}-tree (Server)	B^{ε}-tree (Client)
Create	40158/s	31260/s	65891/s	709704/s
Create (Batch)	41168/s	32288/s	65677/s	791475/s
Delete	42677/s	34853/s	13176/s	16739/s
Delete (Batch)	43251/s	35582/s	13447/s	17689/s
Status	59486/s	40422/s	70950/s	1818030/s
Status (Batch)	524123/s	146697/s	2186016/s	2434915/s
Read	209.0 MB/s	183.3 MB/s	262.3 MB/s	5.7 GB/s
Read (Batch)	2.1 GB/s	1.9 GB/s	2.0 GB/s	6.0 GB/s
Write	210.3 MB/s	170.0 MB/s	197.0 MB/s	1.1 GB/s
Write (Batch, 4KiB)	2.1 GB/s	1.8 GB/s	992.9 MB/s	1.2 GB/s
Write (Batch, 128KiB)	2.1 GB/s	1.9 GB/s	3.3 GB/s	7.3 GB/s
Create-delete	41844/s	32635/s	63344/s	448943/s
Create-delete (Batch)	42181/s	32916/s	65426/s	823341/s

Table A.1.: Results of `julea-benchmark` when executed with different backend configurations.

Bibliography

[Aghayev et al., 2019] Aghayev, A., Weil, S. A., Kuchnik, M., Nelson, M., Ganger, G. R., and Amvrosiadis, G. (2019). File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In Brecht, T. and Williamson, C., editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 353–369. ACM.

[Alatorre et al., 2014] Alatorre, G., Singh, A., Mandagere, N., Butler, E., Gopisetty, S., and Song, Y. (2014). Intelligent information lifecycle management in virtualized storage environments. In *2014 Annual SRII Global Conference, San Jose, CA, USA, April 23-25, 2014*, pages 9–18. IEEE Computer Society.

[Asay, 2021] Asay, M. (2021). Why AWS loves Rust, and how we'd like to help. `https://web.archive.org/web/20210713124447/https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/`. Accessed: 2021-07-13.

[Backes et al., 2018] Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K. S., Rungta, N., Tkachuk, O., and Varming, C. (2018). Semantic-based automated reasoning for AWS access policies using SMT. In Bjørner, N. and Gurfinkel, A., editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE.

[Banker et al., 2016] Banker, K., Bakkum, P., Verch, S., Garrett, D., and Hawkins, T. (2016). *MongoDB in Action, Second Edition*. Manning Publications.

[Beaver et al., 2010] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. (2010). Finding a needle in haystack: Facebook's photo storage. In Arpaci-Dusseau, R. H. and Chen, B., editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 47–60. USENIX Association.

[Becker, 1999] Becker, J. (1999). KNBD – a remote kernel block server for linux. Technical report, NASA.

[Bender et al., 2015] Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. (2015). An introduction to bϵ-trees and write-optimization. *login Usenix Mag.*, 40(5).

[Blackman and Vigna, 2018] Blackman, D. and Vigna, S. (2018). Scrambled linear pseudorandom number generators. *CoRR*, abs/1805.01407.

[Bovet and Cesati, 2005] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media.

[Ceph authors, 2017] Ceph authors (2017). BlueStore config reference. `https://docs.ceph.com/en/latest/rados/configuration/bluestore-config-ref/`. Accessed: 2021-07-18.

[Chodorow, 2013] Chodorow, K. (2013). *MongoDB: The Definitive Guide, 2nd Edition.* O'Reilly Media.

[Comer, 1979] Comer, D. (1979). The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137.

[Curry et al., 2010] Curry, M. L., Ward, H. L., Skjellum, A., and Brightwell, R. (2010). A lightweight, GPU-based software RAID system. In *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 565–572. IEEE Computer Society.

[Elmasri and Navathe, 2017] Elmasri, R. and Navathe, S. (2017). *Fundamentals of database systems*, pages 895–896. Pearson.

[Factor et al., 2005] Factor, M., Meth, K., Naor, D., Rodeh, O., and Satran, J. (2005). Object storage: The future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*, pages 119–123. IEEE.

[Herodotou and Kakoulli, 2019] Herodotou, H. and Kakoulli, E. (2019). Automating distributed tiered storage management in cluster computing. *Proc. VLDB Endow.*, 13(1):43–56.

[Kim et al., 2012] Kim, J., Lee, C., Lee, S., Son, I., Choi, J., Yoon, S., Lee, H.-u., Kang, S., Won, Y., and Cha, J. (2012). Deduplication in SSDs: Model and quantitative analysis. In *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, pages 1–12. IEEE Computer Society.

[Klabnik and Nichols, 2019] Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language.* No Starch Press.

[Klastorin et al., 1993] Klastorin, T., Moinzadeh, K., Diehr, G., and Han, B. (1993). Optimal file management in a hybrid storage system. *European Journal of Operational Research*, 64(3):370–383.

[Kuhn, 2017] Kuhn, M. (2017). JULEA: A flexible storage framework for HPC. In Kunkel, J. M., Yokota, R., Taufer, M., and Shalf, J., editors, *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, volume 10524 of *Lecture Notes in Computer Science*, pages 712–723. Springer.

[Lai et al., 2015] Lai, C., Jiang, S., Yang, L., Lin, S., Sun, G., Hou, Z., Cui, C., and Cong, J. (2015). Atlas: Baidu's key-value storage system for cloud data. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–14. IEEE Computer Society.

[Liang et al., 2018] Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218.

[Lu, 2021] Lu, C. (2021). Seaweedfs, tiered storage documentation. `https://github.com/chrislusf/seaweedfs/wiki/Tiered-Storage`.

[Lüttgau et al., 2018] Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J. M., and Ludwig, T. (2018). Survey of storage systems for high-performance computing. *Supercomput. Front. Innov.*, 5(1):31–58.

[McMillen, 2020] McMillen, W. (2020). Trim command – general benefits for hard disk drives. Technical report, Western Digital Corporation.

[Mendez and Lührs, 2019] Mendez, S. and Lührs, S. (2019). Best practice guide - parallel i/o. https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_Parallel-IO.pdf.

[Meunier, 2016] Meunier, P.-E. (Since 2016). Sanakirja – a copy-on-write key-value store based on b-trees. https://nest.pijul.com/pijul/sanakirja. Accessed: 2021-07-08.

[Moinzadeh and Berk, 2000] Moinzadeh, K. and Berk, E. (2000). An archiving model for a hierarchical information storage environment. *European Journal of Operational Research*, 123(1):206–225.

[Neely, 2017] Neely, T. (Since 2017). Sled, a modern embedded database written in rust. https://sled.rs. Accessed: 2021-07-05.

[Noghabi et al., 2016] Noghabi, S. A., Subramanian, S., Narayanan, P., Narayanan, S., Holla, G., Zadeh, M., Li, T., Gupta, I., and Campbell, R. H. (2016). Ambry: Linkedin's scalable geo-distributed object store. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 253–265. ACM.

[OpenZFS project, 2020] OpenZFS project (2020). OpenZFS documentation. https://openzfs.readthedocs.io/en/latest/performance-tuning.html. Accessed: 2021-06-26.

[Pate, 2003] Pate, S. D. (2003). *UNIX filesystems: evolution, design, and implementation*, volume 10. John Wiley & Sons.

[Persy developers, 2017] Persy developers (Since 2017). Persy, transactional persistence engine. https://persy.rs/. Accessed: 2021-07-05.

[PKWARE, Inc., 2003] PKWARE, Inc. (2003). ZIP format specification. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT. Accessed: 2021-07-04.

[POSIX, 2018] POSIX (2018). IEEE standard for information technology – portable operating system interface (POSIX(tm)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951.

[Rust team, 2016] Rust team (2016). Rust: Frequently Asked Questions. https://web.archive.org/web/20160609195720/https://www.rust-lang.org/faq.html.

[Smart et al., 2017] Smart, S. D., Quintino, T., and Raoult, B. (2017). A scalable object store for meteorological and climate data. In *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2017, Lugano, Switzerland, June 26 - 28, 2017*, pages 13:1–13:8. ACM.

[Smart et al., 2019] Smart, S. D., Quintino, T., and Raoult, B. (2019). A high-performance distributed object-store for exascale numerical weather prediction and climate. In *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2019, Zurich, Switzerland, June 12-14, 2019*, pages 16:1–16:11. ACM.

[Tang et al., 2016] Tang, Y., Hu, G., Yuan, X., Weng, L., and Yang, J. (2016). Grandet: A unified, economical object store for web applications. In Aguilera, M. K., Cooper, B., and Diao, Y., editors, *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 196–209. ACM.

[Wiedemann, 2018] Wiedemann, F. (2018). Modern Storage Stack with Key-Value Store Interface and Snapshots Based on Copy-On-Write Bε-Trees. Master's thesis, Universität Hamburg, Hamburg, Germany.

[Zhan et al., 2020] Zhan, Y., Conway, A., Jiao, Y., Mukherjee, N., Groombridge, I., Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Porter, D. E., and Yuan, J. (2020). How to copy files. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 75–89, Santa Clara, CA. USENIX Association.

[Zhang et al., 2010] Zhang, G., Chiu, L., and Liu, L. (2010). Adaptive data migration in multi-tiered storage based cloud environment. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, pages 148–155. IEEE Computer Society.

[Zuck et al., 2014] Zuck, A., Toledo, S., Sotnikov, D., and Harnik, D. (2014). Compression and SSDs: Where and how? In Maghraoui, K. E. and Kandiraju, G. B., editors, *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW '14, Broomfield, CO, USA, October 5, 2014*. USENIX Association.

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, August 12, 2021

_____

Signature