Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master Thesis

Analysis of the Impact of Aging on the EXT4 and ZFS Filesystems and Their Countermeasures

by Lars Thoms

Scientific Computing
Department of Informatics
Faculty of Mathematics, Informatics and Natural Sciences
Universität Hamburg

Course of studies:	Informatics
Matriculation number:	6415095
First reviewer:	Prof. Dr. Thomas Ludwig
Second reviewer:	Jun.-Prof. Dr. Michael Kuhn
Supervisor:	Jun.-Prof. Dr. Michael Kuhn

Hamburg, 27.11.2020

Contents

Acknowledgments

I would first like to thank my supervisor, junior professor **Michael Kuhn**, who has guided me with his expertise to formulate the research questions, compose the methodology, and evaluate the experiment. His enriching feedback during our discussions pushed me to expand my perspective and knowledge. Above all, he showed me the right tools that I needed to choose the right direction and complete my thesis.

Also, I would like to acknowledge my colleagues from the working group **DL.MIN** at the office of the dean of the Universität Hamburg. They contributed support for my master thesis by providing server and storage systems. Without this exclusive use, the experiments would have been problematic to perform.

Furthermore, I would like to say «thank you» to two other Michael's. My dear colleague and friend **Michael Heinecke** for his emotional support and assistance to understand and select meaningful statistical tests for the evaluation. The stimulating evening discussions with **Michael Straßberger** about possible measurement methods and reasons for anomalies in the measurement results helped me in my work.

Finally, I could not have completed this thesis without my family, who accompanied me through the difficult time of the SARS-CoV-2 pandemic. I would also like to express my appreciation for having endured my temporarily lousy mood (changing programs because of bugs or missing features or the rerun of failed measurement series often tear up schedules).

Thank you!

Abstract

Filesystems, like almost everything, are subject to an aging process. The effects of this are critical in terms of performance. In most cases, depending on the specific filesystem, long-term use leads to a massive fragmentation of the entire storage system.

This master thesis quantifies these performance losses using the benchmark tool *IOR* and compares common countermeasures. For this purpose, different storage media and filesystems (*EXT4* and *ZFS*) are artificially aged using *Geriatrix*. Afterward, they are defragmented using various methods like the defragmentation tool of *EXT4* or several copy mechanisms: file-based by using *rsync* or streamed with *ZFS* standard tools. After each step, the files' fragmentation is determined, and the respective read and write performance.

For this purpose, a new tool called *fraggy* was developed, which counts the filesystem's file fragments by using *fiemap*. Additionally, a patch extended *ZFS* to offer the *fiemap* interface. Furthermore, *Geriatrix* had to be modified to operate on *ZFS*.

The results of this thesis show that aging effects occur in the form of fragmentation. The direct consequences are losses in the read and write performance of new data sets. The defragmentation tool of *EXT4* is not suitable for long-term use and worsens the result even more. The best solution is actually to copy the data to a new filesystem. However, file-based transfers are not suitable for *ZFS*; the better solution is to transfer a complete snapshot to a new pool via its standard streaming tools.

1 Introduction

The problem of organizing data becomes more and more important with the rapidly increasing amount of information. Information that cannot be retrieved in the near term is considered as lost information.

Compared to the analog world's information, stored data could be texts or keywords on an index card. These are usually located in a filing box as an analogy to storage devices. For finding information quickly, lexicographical sorting is advantageous. Nevertheless, what happens if the information on the index cards needs to be changed or added? What happens if a single filing box is not sufficient? How durable is the writing material (ink, paper, etcetera)?

This analogy can be transferred very well to digital storage management. Filesystems are an integral part of the organization of data. The differences between these systems are significant and have to fit their use case.

1.1 Background

However, what happens if a storage system or a filesystem is used for a long time?

Modern storage systems are usually set up after purchase and used excessively for several years. Even storage systems on a smaller scale, such as personal computers, smartphones, or the *Network Attached Storage (NAS)* at home, are only set up once. As a result, they are usually used unattended until their scheduled end of life.

Furthermore, storage systems are often used at a higher fill level (>80%). If changes, deletions, or rewrites are continually being made over many years, this could impact performance. This level of capacity is required, especially in professional and research data centers, such as *High-Performance Computing (HPC)*. A lower usage level is uneconomical.

For example, the *German Climate Computing Center (Deutsches Klimarechenzentrum, DKRZ)* maintains an extensive *Lustre* storage system for its research purposes. It comprises 54 PiB, which involves an investment of about 6 million Euro. This storage system is divided into two clusters. The current utilization (as of 20.10.2020) is 90% with 19 PiB of 21 PiB and 77% with 25 PiB of 33 PiB.

Furthermore, the system has a power requirement estimated at 200,000 Euro per year [1]. The power requirement is identical for each filling level. Therefore, high storage capacity utilization is more favorable. Otherwise, more storage media would have to be purchased, which in turn converts more electricity.

Most of the world's storage systems have to experience performance limitations after several years of high utilization. Therefore, it is reasonable to assume that there must be some aging process for filesystems as well. What changes occur over time, and do they have an impact on reading and writing performance?

1.2 Problem

Viewed over a long time, two properties change in a filesystem: The used storage capacity and the files' fragmentation.

The amount of utilized storage space usually increases over time. The typical user behavior shows: New data is stored, but older files are rarely deleted. Thus, a storage system often reaches its capacity limits over time.

Due to the constant changes of files in the respective storage system, files have to be split up more. Unfortunately, high fill grades increase this effect massively. Furthermore, modern filesystems use *Copy-on-Write* technique, which includes fragmentation by design.

A possible consequence is a decrease in performance. Besides, there is the question of whether and to what extent this effect occurs with different data carriers. For example, a consumer hard disk behaves differently than a data center hard disk. There is a technological difference to *solid-state drives (SSD)*, which must also be considered.

The evaluation of storage- and filesystems for acceptance tests is problematic concerning real-world operation. Usually, measurements are performed with an empty, freshly initialized filesystem. However, institutions rarely use them in this empty state. For economic reasons, a storage system is never used only partially, but always to its limit.

Thus, it means that these measurements are useless because they do not display the real state. A possible consequence is that a reliable system does not meet the usage requirements after a short time.

1.3 Justification

Because storage systems are rarely operated for a short period, it is essential to investigate the age signs of filesystems. It is not only relevant whether a storage system meets the specified performance data. It should still be productively usable after a few years.

The problem becomes worse with increasing abstraction layers. A distributed storage system like *Lustre* uses separate filesystems for each node (for example, *ZFS*), which is merged transparently for the user. In the worst case, files are not only split within a filesystem but across multiple nodes.

Systems of this size cannot merely be reinitialized. Therefore the evaluation of the performance, including the fragmentation behavior, is essential.

1.4 Research Questions

These problems give rise to the following research question: how does the aging process affect the fragmentation and performance of storage systems? Are there any differences between different storage systems, or, respectively, filesystems? At least *EXT4* is not a *Copy-on-Write* system, unlike *ZFS*.

Furthermore, the question arises whether the user behavior influences the development of the performance. Is writing many small files more problematic than saving large ones? Which performance type changes: reading, writing, or both?

Can an administrator measure the fragmentation of a filesystem? Are there special programs for this purpose, and are they compatible with several filesystems? Finally, the most crucial question for operators: Are there suitable countermeasures that prevent the aging process?

1.5 Research Method

Concerning the research questions' answer, performance measurements are applied in several steps in this master thesis. Consequentially, data is collected on empty, fragmented, and defragmented file systems. Various parameters are tested, such as file size or the file system used. Both *EXT4* and *ZFS* are used, and the measurement results are compared.

In addition to the I/O performance, the number of fragments per file is also recorded. Thus, changes in the system can be observed over the steps and related to the performance. The measurement data are described and evaluated at the end. Thereby the respective characteristics of the constellations are evaluated to make suggestions for improvement.

2 Background

2.1 Filesystem

Generally, a filesystem describes a technique to store information on a data carrier, intended to read it back later. Most of the time, users strive to save their data on a non-volatile medium, not only to read back but also to modify.

Andrew S. Tanenbaum wrote in his standard reference *Modern Operating Systems* that storing information has to pass three main requirements [2, pp. 263]:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

In continuation of this, he raised three questions [2, pp. 264]:

1. How do you find information?
2. How do you keep one user from reading another user's data?
3. How do you know which blocks are free?

The term «filesystem» implies that files loom large. A file is neither more nor less an information bundle; in the first place, it is an unspecific data stream. Semantically, it is interpretable such as a text file or a picture. Notwithstanding, the interpretation of data is not the assignment of a filesystem.

To respond to the first question of Tanenbaum mentioned above, files require addresses to store, read, and modify information on storage systems at a later time. There exist two methods to address data on physical media: *Cylinder Head Sector (CHS)* and *Logical Block Addressing (LBA)*.

2.1.1 Addressing Schema: Cylinder Head Sector

The older solution *Cylinder Head Sector* was used to address data chunks (also called blocks) of hard drive disks. The figure 1 on page 14 shows the context of the addressing schema concerning their variables. The first step is selecting a cylinder, also known as track. Therefore, disks consist of n uniformly wide circular rings, wherein turn n describes the offset to the edge of the circle.

Hard drive disks usually contain more than one spinning disk, each of them with an own read/write magnetic head. These mechanical parts are numbered consecutively. Finally, the third variable specifies the sector of a disk. It corresponds to a circular cutout, as seen in the mentioned figure.

A significant disadvantage of this method is the limitation of addressable storage space. Depending on the device interface and block size, different maximum storage capacities are reachable [21, table 5.11, pp. 142]:

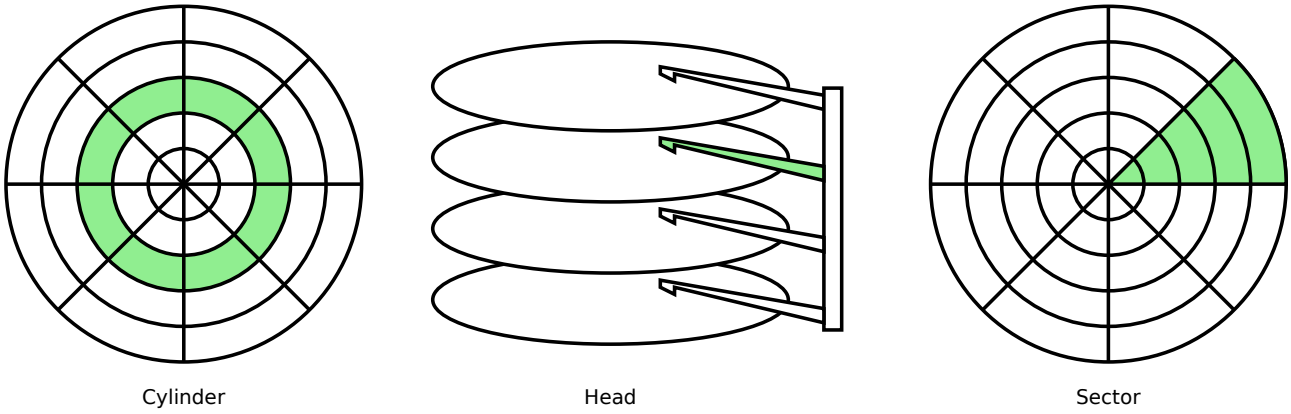


Figure 1: How *Cylinder Head Sector* addressing schema works

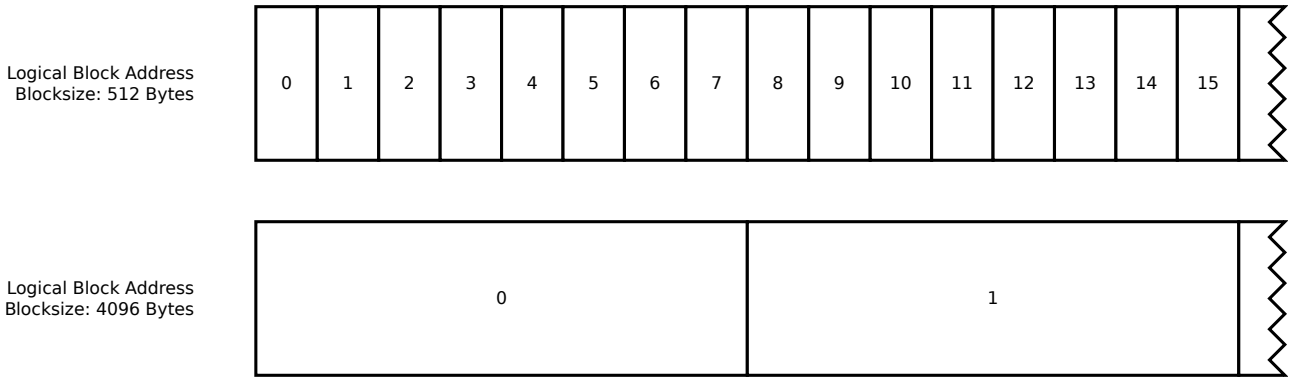


Figure 2: *Logical Block Address* with two different block sizes

	INT 13h BIOS	IDE/ATA
Cylinders	1024	65536
Heads	256	16
Sectors	63	255
Block size	512 B	512 B
Capacity	7.875 GiB	127.5 GiB

$$\text{Capacity} = \text{Cylinders} \times \text{Heads} \times \text{Sectors} \times \text{Block size}$$

2.1.2 Addressing Schema: Logical Block Address

At present, operating systems (OS) and, therefore, filesystems (FS) do not use the *Cylinder Head Sector* algorithm anymore. Through the variety of physical types of storage mediums, they use the addressing schema *Logical Block Address*, since magnetic tapes or solid-state drives (SSD) cannot provide circular cutouts to define sectors, for instance.

This schema counts all blocks linear, beginning from zero. The figure 2 on page 14 shows two different modes of a data drive using *LBA*. The upper one uses an outdated block size of 512 bytes, commonly used in hard drive disks just a few years ago. In contrast, modern drives operate with a size of 4096 bytes, also noted as 4K-HDDs.

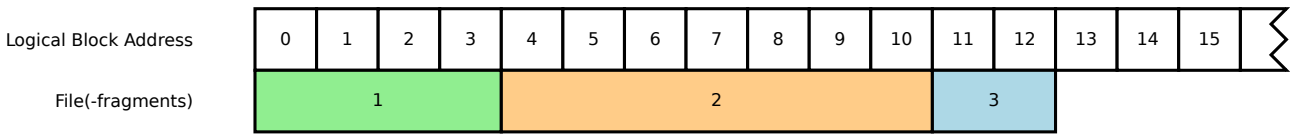


Figure 3: Linear filesystem devoid of vacancy

Since ATA-6[1] addressing of 2^48 blocks is possible [3, pp. 51]. This corresponds to $2^48 \times 4096$ bytes $= 1 \times 10^18$ bytes $= 1$ EiB, for example.

2.1.3 Arrangement of Files on a Physical Storage

As described above, a file containing much information usually requires more than one block on a storage device. Therefore, an arrangement rule is needed.

A simple rule is a linear writeoff, as seen in figure 3 on page 15. On this method, the operating system, more precisely the filesystem driver, write files in one piece without a margin on the storage medium.

For some instances, this method is well-performing: for example, creating backups to a tape archive. Files are written in succession on a linear medium without the intention to change them. Therefore, this is the fastest technique to achieve the shortest writeoff time.

One significant disadvantage of consecutive write is the missing space to grow files. If a user changes its data, a limitation of equal or smaller space demand is pretended. As a result, modifications are hardly possible.

To increase the I/O performance stability over time, it is recommended to set the vacant space between the files to greater than zero. As an example, figure 4 on page 16 uses 25% of its file size as a gap. Some modern filesystems use heuristics to define an individual space demand. Possible influential factors are:

- **Filesize**
  Small files have an enormous potentiality to grow in a short period for more than 100%, in contrast to huge files.

- **Data type**
  A different type, a different use case. Binary files like programs rarely evolve in space by users' hands. By contrast, users often modify their textfiles or spreadsheets.

- **Location**
  In most instances, operating systems use predefined paths for a particular usage. For example, a Linux operating system and programs append logs to files in `/var/log/`. They grow without modification of the older datasets.

[1] AT Attachment: a filetransfer standard for parallel communication between storage devices and mainboard

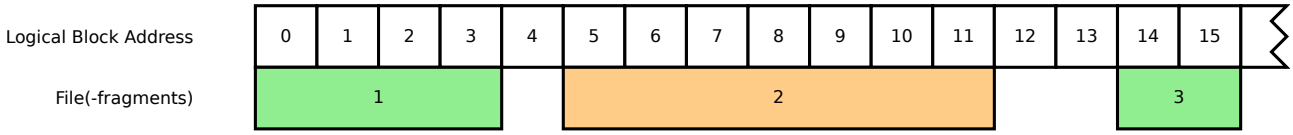


Figure 4: Linear filesystem with margin

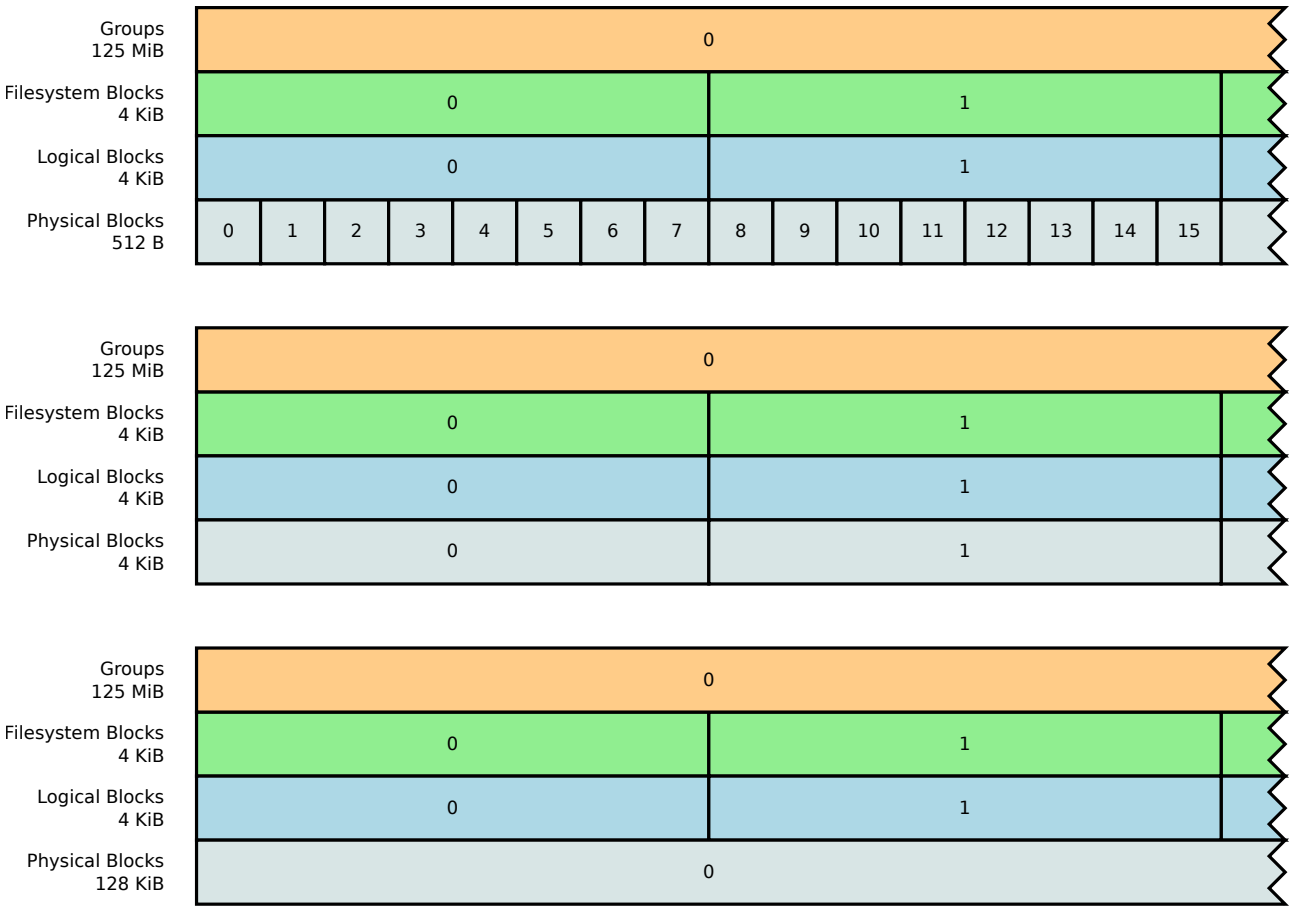


Figure 5: Abstraction of filesystem blocks

2.1.4 Addressing Hierarchy of Filesystems

Filesystems, in general, do not map their addresses to *Logical Blocks*. There exist several layers of abstraction. The figure 5 on page 16 illustrates the four common layers, starting from the bottom:

1. **Physical Blocks**
   The size of the lowest level is not configurable but is specified by the manufacturer of the device. The electric wiring is designed in such a way that writing or reading must always be performed on the entire block. Regardless of the logical block size.

2. **Logical Blocks**
   The reading and writing of storage areas are always based on logical segmentation. It is used for addressing stored data using the *LBA* system, which numbers the blocks linearly. These units are not divisible.

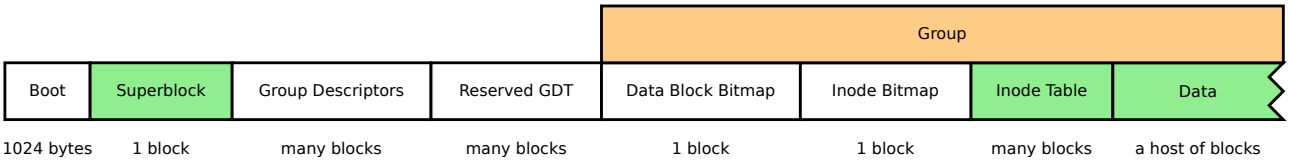


Figure 6: EXT4 filesystem structure

In some circumstances, logical blocks are smaller than physical blocks (as seen in figure 5 on page 16, the third example). In this case, the contiguous/adjacent logical blocks must always be read or (re)written during I/O operations. This behavior usually occurs with *SSD*s, which often have a greater physical block size like 128 KiB.

3. **Filesystem Blocks**
   They are an abstraction to facilitate larger block sizes than an underlying layer provides. In the first example, the device only supports a physical block size of 512 bytes, which leads to administrative overhead and performing issues on large disks with huge files.

   Therefore, small filesystem blocks are suitable for storing small files. Also, the page sizes of databases are usually quite small and should be adjusted to this block size.

   Large blocks are useful for managing large files. The sequential read and write performance is higher due to the reduced overhead. On the other hand, small files waste much space because of the lessened degree of filling the blocks. Consequently, an administrator has to configure this parameter according to its final usage.

   ZFS, for example, allows different block sizes (termed as *record sizes*) for datasets. This feature enables the administrator to select the optimal sizes depending on the application.

4. **Groups**
   Due to performance tunings, most filesystems clusters their blocks to larger chunks with their management blocks. The size is configurable, as well.

2.1.5 Fundamental Structure of a Filesystem

The last paragraph describes how filesystems address their data on a storage device. However, it is not clear what a filesystem structure looks like yet. The next paragraphs illustrate a prevalent setup, based on the filesystem *EXT4*.

In general, a filesystem is segmented in many sections, as seen in figure 6 on page 17. Green filled blocks are described in the next sections more detailed. The denoted sizes refer to *Filesystem Block* sizes, not to be confused with *Logical Block* sizes.

A brief explanation of the other blocks of EXT4:

- **Boot**
  This area is reserved for boot sectors of operating systems or other undocumented proceedings. Therefore, filesystems ignore this block.

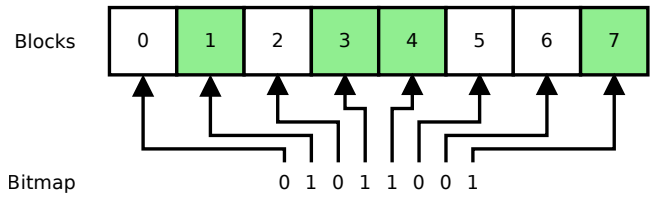


Figure 7: Bitmap functionality

- **Group Descriptors** and **Reserved Group Descriptors Table**
  EXT4 is capable of grouping blocks to optimize I/O throughput. Each of these clusters requires a *Data Block Bitmap*, an *Inode Bitmap*, an *Inode Table*, and a *Data* block. The section *Group Descriptors* contains the addresses of all these metadata, among others.

- **Data Block Bitmap**
  Each bit of this block indicates the usage of a data block within a group, as seen in figure 7 on page 18.

- **Inode Bitmap**
  Likewise, this bitmap refers not blocks but inode slots in their table.

2.1.6 Inodes

Inodes, an abbreviation for index node, allow to find files in a filesystem without scanning the entire disk (worst case $O(n), n :=$ disk size) until the wished file is found an index is needed. Accordingly, an *Inode Table* is available at each block group. These lists consist of tuples containing at least the start position, the length, and an identifier.

In addition to the interfaces and attributes specified by *POSIX*, various filesystems implement their own. For example, *EXT4* stores the date of birth as additional information. By default, *POSIX* specifies that at least the change, modification, and access date have to been stored. Besides saving owner- and group-ID, so-called *Access Control Lists (ACL)* can also be offered.

To address data blocks in a filesystem, older filesystems like *EXT2/3* used a direct/indirect block addressing schema, shown in figure 8 on page 19.

The first 12 of 15 inode address entries contain a 1:1 map to a filesystem block address (charted with a straight arrow). The 13th entry is a reference to an address containing an address block from 0 to n, where $n = \frac{\text{blocksize}}{4}$. Also, these entries are a 1:1 map to a filesystem block address.

Number 14 is a double indirect block, which references a list of indirect blocks, which references address blocks. The 15th entry is similar, except it is a triple indirect block.

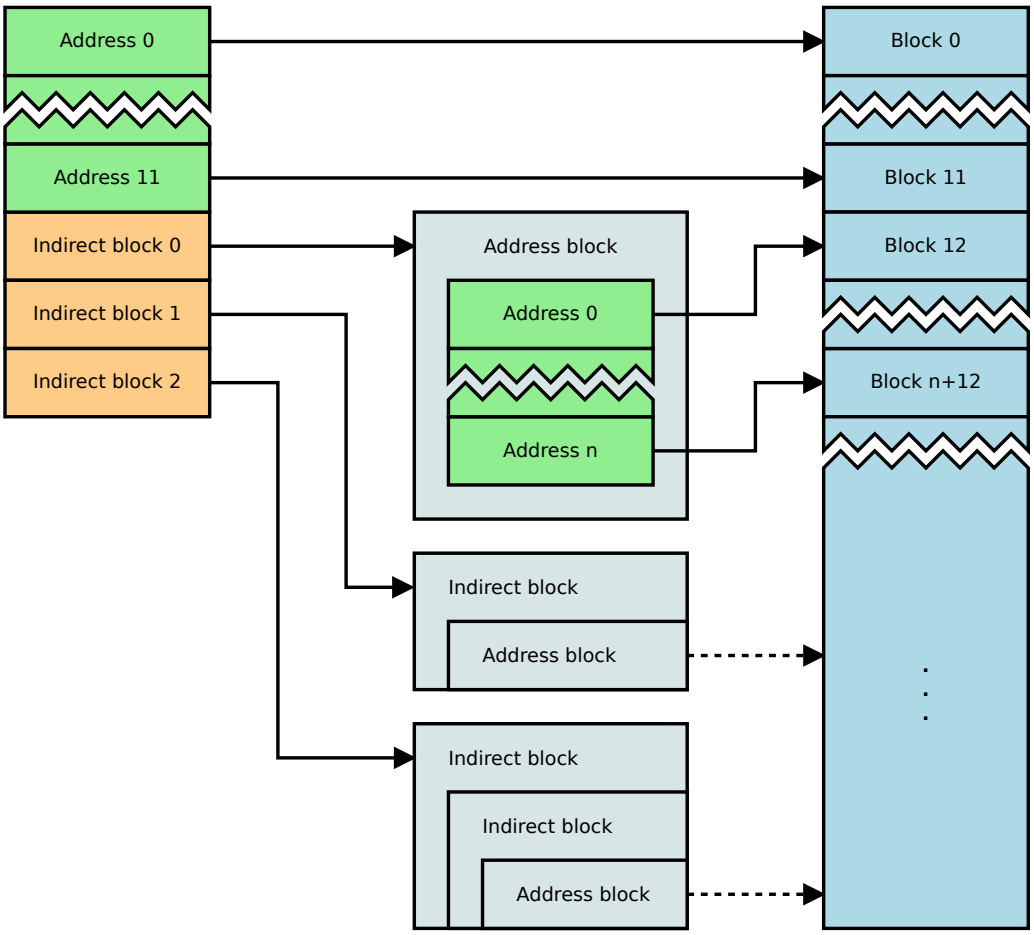


Figure 8: Direct/Indirect block addressing [32, p. 16, figure 9][4]

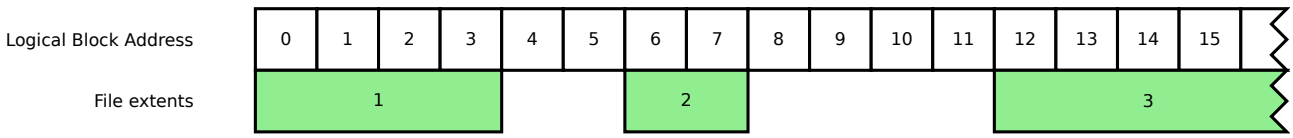


Figure 9: Extents of a file

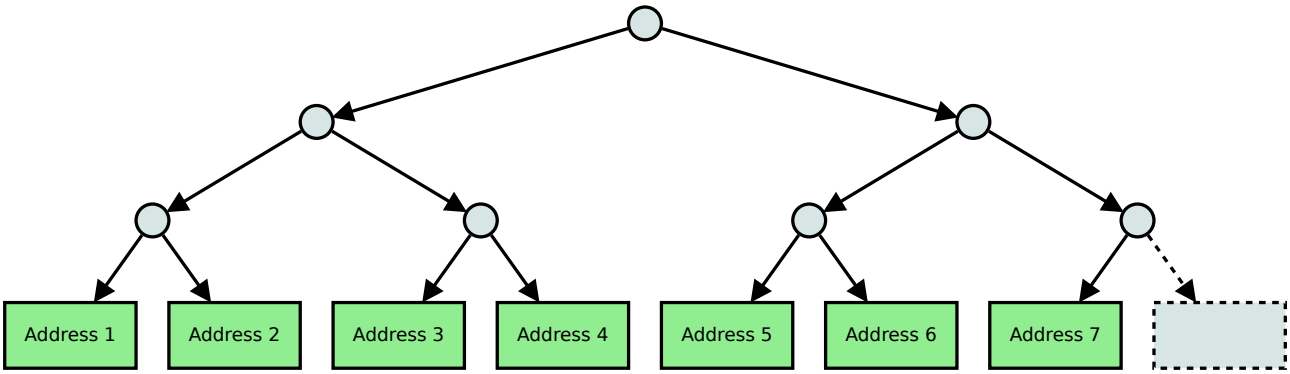


Figure 10: Example of a hash tree used in filesystems

2.1.7 Extents

The solution to split data into blocks is inefficient due to I/O performance, and therefore, the file-size is sharply limited. The calculation of the maximum file size using block mapping is quite simple. The first twelve fields contain direct addresses, and each indirect block maps $\frac{block\_size}{4}$ entries.

$$12 \times block\_size + \sum_{i=1}^{3} \frac{block\_size}{4}^{i} \times block\_size$$

Assuming that the block size is 4 KiB, the maximum file size is 48 KiB + 4 MiB + 4 GiB + 4 TiB. Consequently, modern filesystems use extents: a dynamic amount of coherent, filesystem blocks as symbolized in figure 9 on page 20.

A significant advantage of this technique is to store larger files and higher throughput while reading or writing data. The reason for this lies in the physical functionality of storage devices. The write-read head of an HDD always operates at the same speed. However, if blocks are allocated in succession, the head has not to ignore many blocks while operating.

Similarly, when an SSD is used. Since the controller chips maintain the distribution of data, a meaningful approach is to read/write large chunks to ensure a great dataflow.

Additionally, current filesystems do not use direct and indirect block addressing anymore. Instead, they use tree structures as in figure 10 on page 20. There are different types of implementations.

EXT4 uses a hash tree; therefore, only the leaves contain addressing information, and all the other nodes are links to them. In contrast to *Btrfs*, it is using a B-Tree which nodes contain addresses. Addresses before and after that node value are links to the next layer.

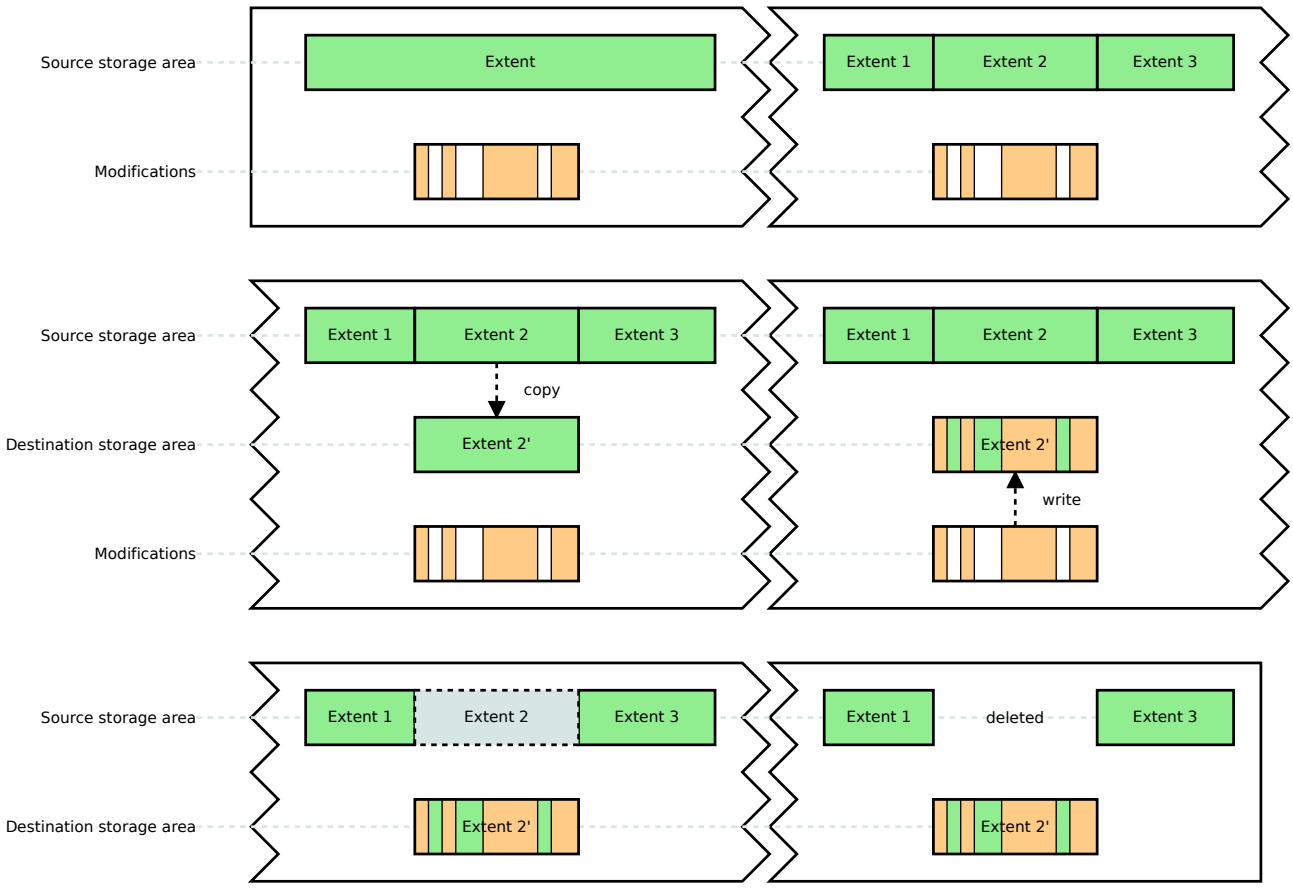


Figure 11: Write operation on a CoW filesystem

2.1.8 Copy-on-write

Some filesystems, like *ZFS* or *Btrfs*, were designed as a so-called *Copy-on-write* filesystem, short *CoW*. These filesystems are robust against failures during write operations. This is due to its design, how data is (over-)written:

1. Should an extent be modified?
   Yes: Proceed
   No: Stop
2. Does the modification affect a small part of the extent?
   Yes: Proceed
   No: Jump to 4.
3. Set virtual crop marks before and after the parts to be modified.
4. Copy (sub-)extents to another storage area
5. Store modifications on the copied extents
6. Commit new address and, if crop marks are available, new extents

A schematic illustration of a file modification offers figure 11 on page 21. The first step consists of an extent and a modification request. Since the extent is large, the filesystem sets two crop marks. In the following step, it copies the middle block. As the fourth step, it writes down the changes to the copied extent. A commit follows it in case of success. Commit means atomically writing down the new addresses into the inode block to minimize failures due to blackouts.

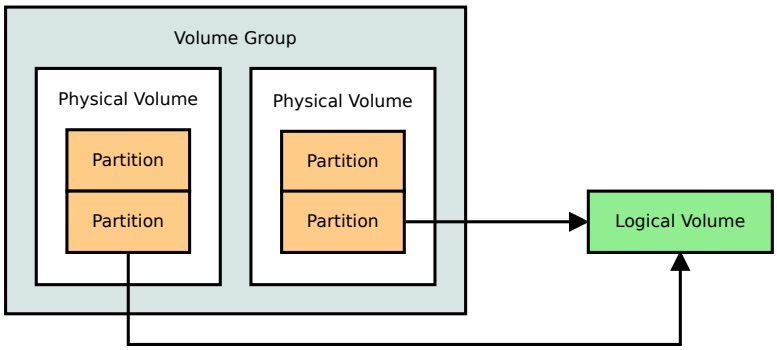


Figure 12: Structure of a LVM

One downside resulting from copying existing data is a reduced overwrite performance. In exchange, a file versioning on a storage level is available. Therefore, filesystems like *ZFS* and *Btrfs* comes with a toolchain for managing snapshots.

2.2 Storage Systems

Due to the higher demand for storage capacity, a single storage medium does not satisfy these requirements. A current standard market hard drive disc provides approximately 16 TB, which is not equal to the demands of server owners or customers with a significant data emergence.

Therefore, different solutions for individual needs exist:

- Logical Volume Manager
- Redundant Array of Independent Disks
- Clustered and Distributed Filesystems

All these techniques are currently in use and fulfill various use-cases.

2.2.1 Logical Volume Manager

A *Logical Volume* is an amalgamation of different physical partitions and reveals one virtual block storage. The figure 12 on page 22 shows a simple LVM setup with two disks *(Physical Volumes [PV])* in a *Logical Group (LG)*. Each of them contains two partitions *(Physical Partitions [PP])*. For this example, one *PP* of each *PV* is used to form a *Logical Volume (LV)*.

Importantly, this virtual block device does not provide any information, how this storage is aligned or replicated. The first, but simple configuration mode is to put the storage cells in a row. Above these merged storage capacities, the user could configure any desired filesystem and mount it as a single device.

Furthermore, it is possible to configure an LVM to operates like a RAID (see next paragraph), with the difference that an LVM intentionally not provides any redundancy.

2.2.2 Redundant Array of Independent Disks

In contrast to *LVM*, the primary function of a *Redundant Array of Independent Disks (RAID)* is providing redundancy of stored data with the help of multiple storage devices.

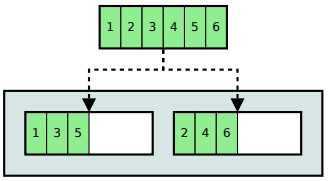
Redundancy must not be confused with data protection or backup! It aims to reduce the possibility of I/O failures by replicating and creating parities. To be more accurate: increasing the *Mean Time to Data Loss (MTTDL)* is the prime objective. A further advantage of some modes of operations is a boost of I/O performance. The next subsections describe different modes, advantages, disadvantages, MTTDL, and application areas.

Variable	Definition
n	Number of involved disks
s	Storage capacity (per disk)
MTBF	Mean Time Between Failures
MTTR	Mean Time to Repair
MTTDL	Mean Time to Data Loss

RAID 0 Data is striped to all available disks.

Strictly speaking, this is not a proper *RAID* configuration, because there is no redundancy. Thus, it acts more like an *LVM* than a *RAID*.

- MTTDL: $\frac{\text{MTBF}}{n}$
- Storage capacity: $n \times s_{min}$
- Minimum number of disks: 2

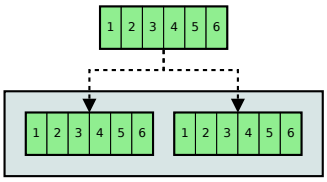


One of the reasons why *RAID 0* is commonly used, is because of the higher I/O performance (approximately up to n times), even though it takes no data security measures. A performance increase exists only if a file is distributed over all disks.

Its main operation fields are combinations with other RAID levels or consumer storage systems.

RAID 1 Data is mirrored to all available disks.

- MTTDL: MTBF^n
- Storage capacity: s_{min}
- Minimum number of disks: 2



The n-times redundancy and the approximately n-times higher read performance of this method are attractive in regards to the security objective «availability.» It is highly suitable to *write fewer read many* systems, as there is no speedup during write operations. Even worse: the writing cycle of the worst disk is decisive.

RAID 2 *(deprecated)* – bit-level striping of data with dedicated Hamming-code parity.

- MTTDL: corresponds to RAID 5
- Storage capacity: $2^{\log_2(n+1)} - \log_2(n+1) - 1$
- Minimum number of disks: 3

This method was mainly developed and used for old mainframes. The use of a Hamming-code as parity resulted in the possibility of an *Error Checking and Correction (ECC)* procedure. The reading rate of mainframes increased as a result.

A notable disadvantage of this system is that the number of disks should be an integer multiple of the Hamming codeword length to detect bit errors. This condition leads to inflexible pool sizes.

RAID 3 *(deprecated)* – byte-level striping of data with a dedicated parity disk.

- MTTDL: corresponds to RAID 5
- Storage capacity: $(n-1) \times s_{min}$
- Minimum number of disks: 3

Like to the following *RAID 4* and *5* procedures, the system calculates parity and writes it to a dedicated device. The calculation of parity usually uses simple, fast operations, such as `XOR`:

$$11101101_2 \oplus 00101100_2 = 11000001_2$$

Furthermore, the performance of this algorithm is similar to *RAID 4* and *5*. In theory, it achieves a speedup of $n-1$. Besides, this and these systems ensure data integrity in the event of a single device failure.

Unfortunately, *RAID 3* and *4* have the notable problem that there is an unbalanced load on the number of I/O operations per disk. Optimally, the file size should be an integer multiple of $n-1$ so that all disks are written equally often.

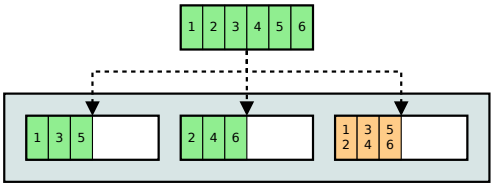
While reading, the parity disk is not used, so the data disks are subjected to a comparatively more significant load. On top of that, this system places more strain on hard drive disks due to the mechanical construction.

Furthermore, write operations cannot adequately be executed in parallel, since the parity disk must always be involved. As a consequence, the write rate is quite limited.

Especially with this RAID, the small chunk size of a single byte is a performance killer. Due to the large discrepancy between logical and physical block sizes, the RAID does not achieve an adequate IOPS rate.

RAID 4 Data is striped on a block-level with a dedicated parity disk.

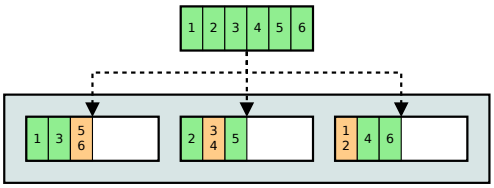
- MTTDL: corresponds to RAID 5
- Storage capacity: $(n-1) \times s_{min}$
- Minimum number of disks: 3



The only difference to *RAID 3* is the chunk size, which corrects the I/O speed losses. Otherwise, the same problems exist.

RAID 5 Data is striped on a block-level with parity.

- MTTDL: $\frac{\text{MTTF}^2}{n \times (n-1) \times \text{MTTR}}$ [22, p.111]
- Storage capacity: $(n-1) \times s_{min}$
- Minimum number of disks: 3



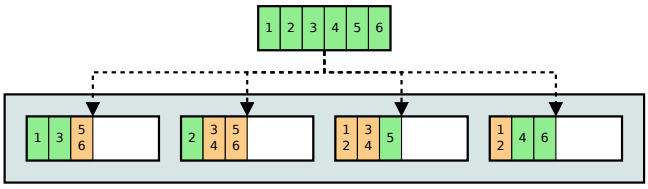
In contrast to *RAID 3/4*, in which the parity disks are fixed, RAID 5 distributes the parities to all available disks according to round-robin. The I/O performance speedup shall be at least $n-1$, as described in *RAID 3*.

By correcting the disadvantages of *RAID 3* and *4*, this configuration is quite standard in servers and NASes. It allows the failure of a single device and is comparatively the cheapest method to build redundancies.

Notwithstanding, if a hard disk fails and the administrator starts a resilvering process, the probability of another device failure is significantly higher. Therefore, to reduce this kind of likelihood, it is necessary to use disks of different brands, generations, and batches.

RAID 6 Data is striped on a block-level with double parity.

- MTTDL: $\frac{\text{MTTF}^3}{n\times(n-1)\times(n-2)\times\text{MTTR}^2}$ [26, p.163]
- Storage capacity: $(n-2)\times s_{min}$
- Minimum number of disks: 3



RAID 6 shares most of the elements of *RAID 5*; the number of parities doubles to 2; therefore, the speedup is $n-2$. Furthermore, the failure rate during the resilvering of a single disk is lower than *RAID 5*.

The calculation of two parities is slightly complicated, in contrast to single parity. In the following equation, the parities are referred to as P and Q, and n defines the number of disks.

$$P = D_0 + D_1 + \ldots + D_{n-1}$$
$$Q = g^0 \times D_0 + g^1 \times D_1 + \ldots + g^{n-1} \cdots D_{n-1}$$

This equation uses a finite field to calculate two isometric parities. Isometric means that the original size corresponds to the resulting size. Since these fields based on a prime number M and vector sizes (blocksize) usually are not, it is necessary to use a Galois field $GF(M^m), m \in \mathbb{N}^+, m :=$ blocksize.

Consequently, P is an `XOR` parity, considering addition as `XOR`. Q, also referred to as Reed-Solomon code, used g^n as a simple generator.

2.2.3 Distributed Filesystem

The systems described in the previous sections are only suitable for local systems, such as a NAS, PC, or server.

In order to enable high-capacity storage solutions, such as those offered by cloud infrastructure providers, a combination of several storage systems is required. These should behave similarly to a *RAID* and offer further functions for an extended area of application.

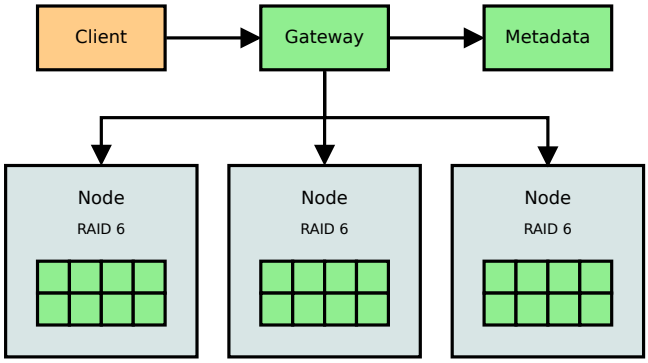


Figure 13: A simple distributed filesystem

For these reasons, distributed filesystems come into play. By definition, they are a combination of servers that store and read data on local data storage devices via a network medium according to a defined scheme.

The local storage systems are mostly *RAID* systems with a conventional filesystem, such as *ZFS* or *EXT4*. Many distributed filesystems use a gateway or management server to which the user connects. Therefore, the user can not see its topology; consequently, the storage system is transparent.

The figure 13 on page 27 shows an example of how a storage system could be structured. It includes a gateway, three storage nodes, each with eight hard drive disks in a *RAID 6*, and a metadata memory.

Initially, the user connects to the gateway. During a read operation, the user asks the metadata server, containing the corresponding resource's location, which can be loaded now. During write access, the user deposits a new entry on the metadata server and stores data on the system.

Distribution Mechanism The exact distribution of the data depends immensely on the systems used and its tasks. One possibility would be that the algorithm always writes files to a storage node in a contiguous manner.

The advantage here is that if a node fails, at least some of the files are entirely readable. On the other hand, the load can be extremely skewed if the data sizes diverge greatly.

In this case, it makes sense to stripe the files to all or some nodes, similar to a *RAID 0*. The figure 14 on page 28 shows last option.

Different Philosophies In the case of distributed systems, a distinction is also made between hierarchical and object-based storage systems.

Hierarchical filesystems offer the classical abstraction of folders and files, where a path is assigned to a file. This practice originates from the analog paper world.

Object storage systems, on the other hand, regard a file as an object that has specific searchable attributes. Therefore, there is an object ID, but no classic file name. These are grouped on the first level by pools.

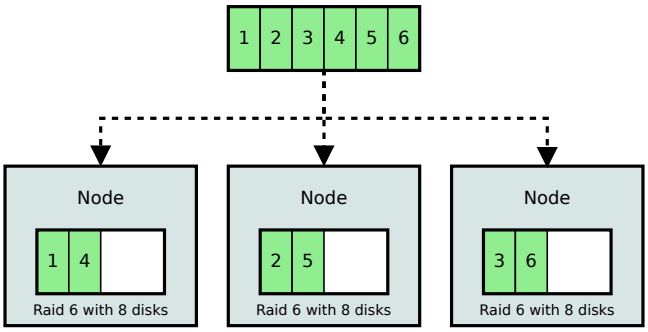


Figure 14: The mechanism of a distributed filesystem

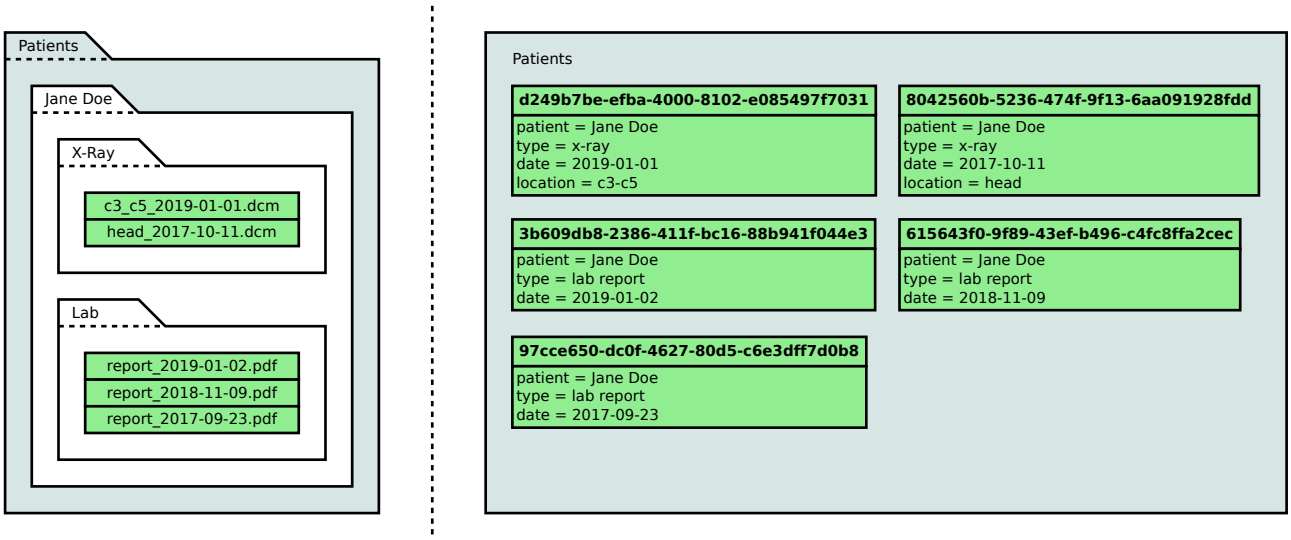


Figure 15: Hierarchical filesystem (left) versus objectstorage (right)

The figure 15 on page 28 shows the two different philosophies in the context of patient files. In the hierarchical model, the files are grouped and sorted thematically. On the other hand, the other model attributes each object so that it can be displayed filtered.

Improvements In order not to send all data through the gateway, which leads to a bottleneck effect, optimization possibilities exist. For example, the gateway could only function as the first entry point. As a result, the gateway transmits the address to the metadata server and the client queries and loads or saves the data independently on the assigned nodes, as seen in figure 16 on page 28.

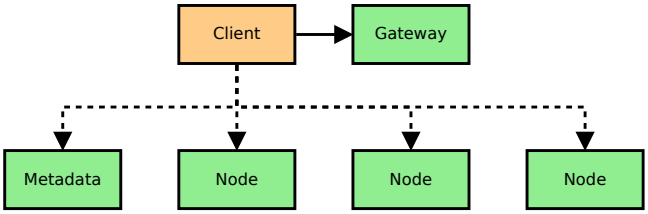


Figure 16: Network optimization in distributed filesystems

In some systems, metadata servers are not in use. The storage location is derivable from the given resource name. These systems require a global view of the used topology. As a reference, *Ceph* uses a so-called *CRUSH algorithm* to determine how to store and read data.

Such systems use an adaptation of the hash tree (or Merkle tree) to map to the given topology. The storage nodes form the leaves. For example, if there is no (sub)grouping of nodes, the tree has a depth of one with n edges.

Of course, to further optimize the system omitting the gateway server is also possible. Therefore, every node knows the current topology and is reachable under the same domain.

2.2.4 Clustered Filesystem

By definition, distributed filesystems are accessible from anywhere over the network. Usually, the user applies known protocols like *NFS* or *SMB*. Product-specific clients are also possible, which also upload and download files via the network.

The main difference to clustered filesystems is that they are integrated differently. Here, people use technologies such as *FibreChannel* or *iSCSI*, which make it possible to mount the storage system directly as a block device in the user system.

Thus, a cluster filesystem is integrated as a virtual block device on several clients and is seemingly a local hard disk. Therefore the implication is: Clustered Fileystem → Distributed Filesystem.

2.3 Reading and Writing Behavior

How operating systems store or read their data in detail on a data carrier depends on the hardware and the degree of optimization used. In the end, however, it is always the controller chip that decides which the operating system is addressing with drivers.

The first rough distinction is the use of rotating magnetic disks or flash memories.

2.3.1 Hard Drive Disks

The main problem with hard drive disks regarding I/O operations is the so-called *seek time*. It describes the time between the start of the task request and reaching the correct read/write position. Another problem is the desired storage density of a disk without changing the standardized design (3.5" or 2.5") or interface (*SATA* or *SAS*).

Increase of Storage Capacity The first method to improve the storage density is to switch from *longitudinal recording (LR)* to *perpendicular recording (PR)*. As shown in figure 17 on page 30 and figure 18 on page 30, each dipole represents one bit. The magnetic moment $\vec{m}$ is no longer horizontal but vertical and allows a higher density due to the space-saving.

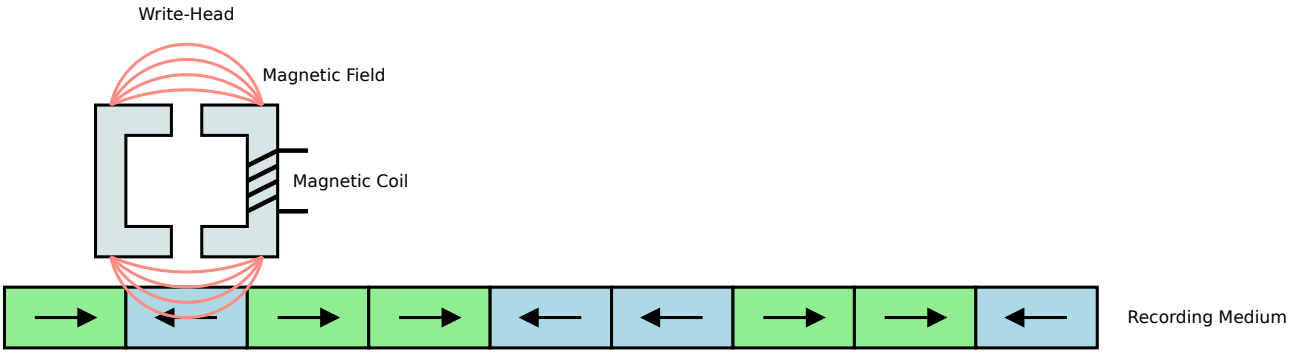


Figure 17: Longitudinal Recording

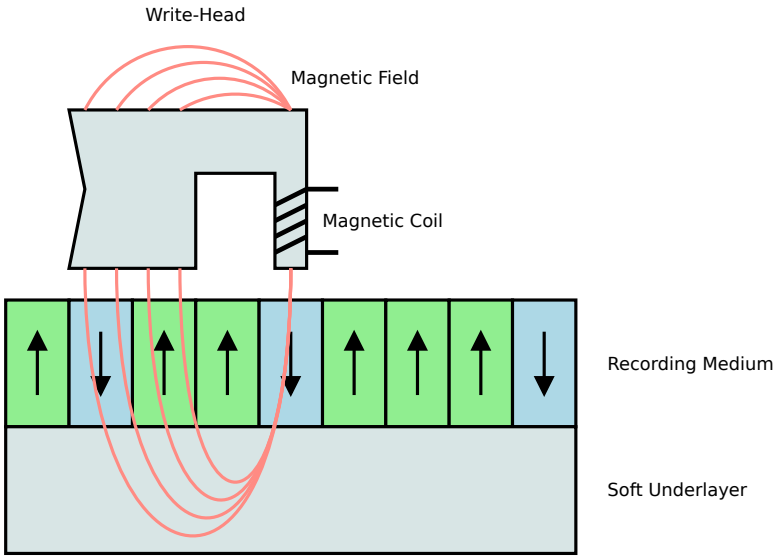


Figure 18: Perpendicular Recording

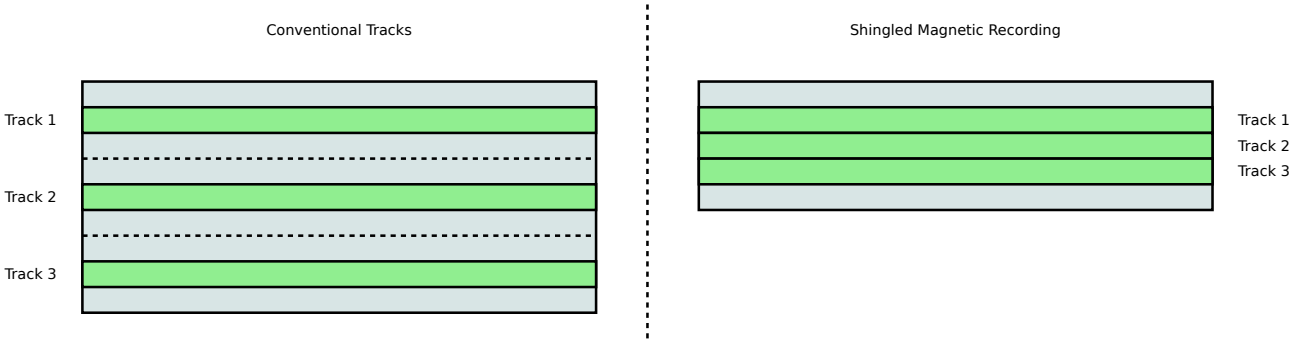


Figure 19: Shingled Magnetic Recording

Another technique for reducing track spacing is called *shingled magnetic recording (SMR)*. As shown in figure 19 on page 31, the conventional tracks are merged with their distances in groups. During a write operation, the write head overwrites the adjacent tracks due to its overlapping size. As a consequence, it has to rewrite all tracks of a group spirally. Further on, there is a distance after each group again.

During the constant overwriting of an entire spiral group, *SMR* reduces the write performance in regards to data density.

A rather new and sophisticated method is the so-called *heat-assisted magnetic recording (HAMR)* process. The size of the individual dipoles of the magnetic layer is smaller than before. A laser heats the environment to be described to its Curie temperature T_C, making it paramagnetic instead of ferromagnetic. The dipole size ensures that even a superparamagnet is formed and reacts more sensitively to external magnetic effects. Consequently, a smaller magnetic field strengths $\vec{H}$ is needed to write on disk, which allows higher precision.

Replacing the gas medium air inside the hard drives with a less dense one, such as helium, is also an approved method of increasing storage capacity. The lower density reduces the flow effects of the read/write heads between disks. It follows that seven instead of five rotating disks are in use.

Position of the Data The use of several rotating disks within a hard drive disk increases the potential read and write rate. The read/write heads are coupled to each other, so they are fully synchronous on every disk level.

For the best-case scenario, given data has to be written/read on the same track on all layers instead of asymmetrical data distribution. The controller does this exact allocation of data; the operating system has hardly any influence here.

The data is further provided with *forward error correction*. It is stored on the disk using a modulation, such as *partial-response maximum-likelihood (PRML)* or *extended PRML (EPRML)*. As a result, the stored data is not isometric compared to the original data.

Because *PRML* uses a dynamic threshold to digitize a signal, these checksums are necessary. The thresholds are corrected to ensure that the calculations are correct. Thus the magnetic variance of a carrier material, influenced by composition and temperature, can be counteracted. Furthermore, the noise problem is reduced.

Furthermore, the controller conceals defective areas *(bad blocks)* with blocks from the *hot fix redirection area*. The position of the data block is elsewhere and transparent to the operating system. The controller uses a translation table to address them. This total failure prevention method slows down the whole system since the read/write head has to be realigned for these redirected blocks.

2.3.2 Solid-State Drives

In contrast to the hard disk, the *SSD* is not subject to mechanical wear due to actuators. An *SSD* consists of many small memory cells, which are electrical semiconductors. States are encoded accordingly by the electrical charge, usually more than one state per cell.

The main problem of an *SSD* is that these cells lose capacity with an increasing redescription, and thus the discrete distances to distinguish the states decreases.

As a result, an *SSD* can no longer use a cell after a certain number of write operations. Conventional cells can be written 1000 times before the cells do not permit any more differentiable states.

Longevity of an SSD The rewritability of *SSD* cells is limited before the differentiation of electrical signals is no longer possible. Commercially available *SSDs* can be rewritten about 1000 times. To avoid rapid obsolescence, *SSD* controllers use some quite sophisticated techniques.

The first technique is the so-called *wear-leveling*. The controller distributes all write operations pseudo-randomly to all cells to achieve a homogeneous write load per cell. In return, the controller most likely will store a rewritten file fragment in another cell.

To avoid a counter and an allocation table addressing each cell, the cells are grouped into pages. They resemble the physical blocks of a hard disk.

By hosting *spare areas*, the controller reserves a significant memory area of an *SSD* for later use. Therefore, the controller can replace defective cells by remapping. So this area supports *wear-leveling* to be able to use an *SSD* for quite a long time.

Position of the Data Due to the two algorithms mentioned above, the operating system cannot get the exact position of data or enforce a specific address for write operations. The distribution of the data over the pages is also completely transparent. The controller only returns addresses which it may remap itself with a translation table.

2.4 Aging Process of a Filesystem

Everything ages, why shouldn't this apply to a filesystem? An interesting aspect here is how a system ages and what effects it has. There are several questions for the analysis of the aging process:

1. What are the tasks of this computer?
2. Which I/O operations do a computer perform throughout time?
3. How long is the period?
4. What kind of data is stored?
5. What storage solution is used?
6. What is the ratio of used and available storage capacity?

The first question is very characteristic for the intended use. A home computer on which a user mainly plays games has an entirely different I/O behavior than a bank's mainframe on which databases are operated.

Therefore, typical I/O operations are resulting. A gaming PC will mostly read some large files and update them occasionally. Furthermore, such systems are unlikely to run day and night for 5-10 years without reinstallation or retirement.

In contrast, the database server in question runs for more than a decade without reinstallation. Besides, it writes a lot of small changes at different places on a storage medium.

Moreover, both systems do not necessarily use a similar storage solution. A PC probably uses a single *SSD*. The mainframe probably uses a *RAID* as long-term storage with intermediate storage units, consisting of *SSDs*, which are used in a *RAID*.

On top of that, the fill level of a system is very relevant, since specific effects only occur with a more extended period while operating with a high load.

2.4.1 Arrangement of Files in the Storage System

There are several ways in which a filesystem can store files on a storage system. One possibility has already been mentioned in section 2.1.2: The linear arrangement with and without spacing.

The main problem with contiguous storing of data arises with non-isometric changes. Considering that most filesystems are used in situations where most of the transactions consist of change requests, this method is not suitable for storage systems in production mode, and a different algorithm is required.

Therefore, the file usually needs to be split, since the following memory area is either already occupied or insufficient.

The decision about the size of a fragment does not have to be made statically. Instead, it is a trade-off between various factors. These include, among others

- the total size of the file and the number of fragments calculated from it,
- the storage availability of the filesystem and
- the writing behavior.

The writing behavior can be a criterion for decision making. The following questions are conceivable:

- Is data only appended, like in a log file? Then it is worthwhile to use nearly linear storage with a large reserved area after the file.
- How many changes are made (database versus text document)? Are they isometric? For isometric changes, large fragments are useful, otherwise small ones.
- How extensive are the changes? They should be reflected in the fragment size.

The final decision on the size of a large file fragment is preceded by an assessment of the following advantages and disadvantages:

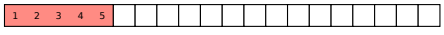
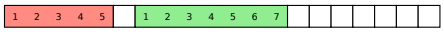
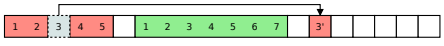
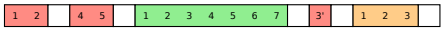
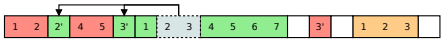
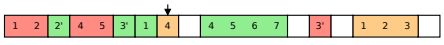
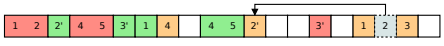
Criterion	Small Fragments	Large Fragments
Number of address entries in the metadata area	High (-)	Low (+)
Number of jumps during I/O operations	High (-)	Low (+)
Data to be copied on *Copy-on-Write* systems	Little (+)	Much (-)
Storage availability of forced fragment size	High (+)	Low (-)

However, the biggest problem with fragmentation arises with distributed storage systems, such as *RAIDs*, *Logical Volumes*, or *Distributed Filesystems*. Especially in a heterogeneous storage structure in terms of physical and logical block size, the number of fragments can vary substantially. Keeping control over a homogeneous distribution of fragments in a distributed system is also rather sparse.

The controller of a distributed filesystem does not necessarily have to possess the information about the detailed storage status. Most of the time, it is sufficient for him to know about the fill levels of the respective nodes and their availabilities. It is not essential to tell the controller the exact distribution between the disks, the number of faulty memory areas, etcetera.

2.4.2 Behavior over Time

In the following, a small filesystem is simulated to get an impression of how a filesystem might behave over time. Files are written, modified, and deleted. It is a *Copy-on-Write* system.

Step	Operation	Filesystem
1	Write file «red»	
2	Write file «green»	
3	Update a block of file «red»	
4	Write file «orange»	
5	Update two blocks of file «green»	
6	Add more data to file «orange»	
7	Delete two blocks of file «green»	
8	Update a block of file «orange»	
9	Add more data to file «red»	

The simplified simulation shows a filesystem with a high degree of occupancy. As seen in figure 20 on page 36, from step four, the level is 75% and increases to 90% by step nine. Due to this load, changes are almost only possible by splitting into fragments.

Step five illustrates this very well, where a part of the existing fragment is broken up, duplicated into two fragments due to *Copy-on-Write* and lack of space and extended with new data contained in a third fragment. As a consequence, a file consisting of one fragment splits into five parts. More generally, by looking at figure 21 on page 36, it can be seen that the number of fragments increases for all files as soon as a high fill level is reached.

2.4.3 Counteractive Measures

Various approaches are conceivable to counteract a high degree of fragmentation. This problem is probably the reason for the decreasing performance of a full and old filesystem.

The most straightforward and non-technical solution is to force a fill level below $n\%$ (for example, 80%). This goal can be determined by rules that block the additional storing of data. Reserving storage areas on filesystem level or quotas, enforced by the operating system, are also conceivable. Also, the procurement of additional storage elements in order to grow the total capacity is imaginable. These possibilities are technically easy to implement but require a partial uselessness of the hardware.

Another solution, which would exhaust the fill level to some extent, is *planned fragmentation*. Each larger file is divided into several fragments in predefined sizes and distributed on the data storage. This method causes the user to accept a loss of performance at the beginning. On the other hand, the problem of massive fragmentation is reduced, or more specifically, it requires more time to emerge. The last option considered in this thesis is *defragmentation*. It is an active procedure for re-sorting and merging logically related data blocks.

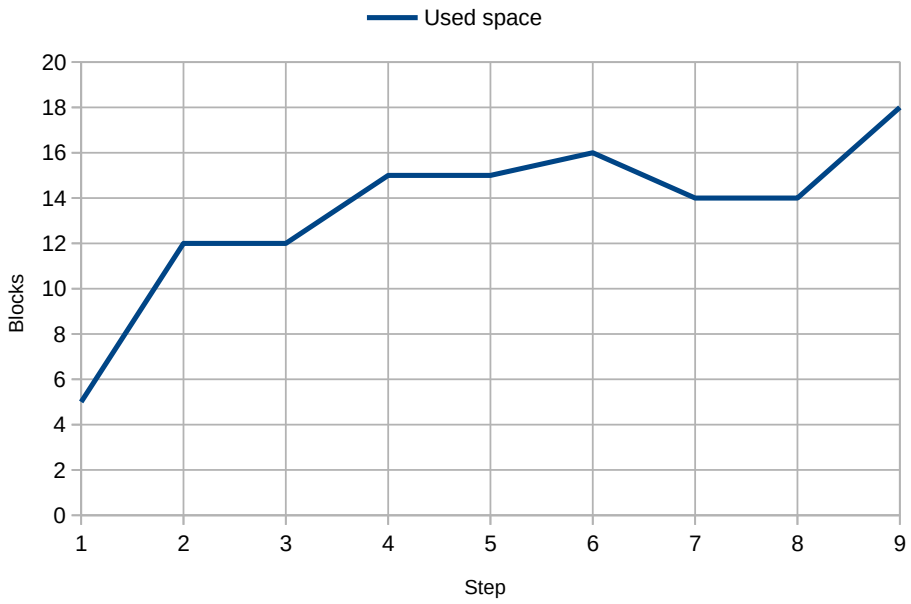


Figure 20: Storage space usage of simulation

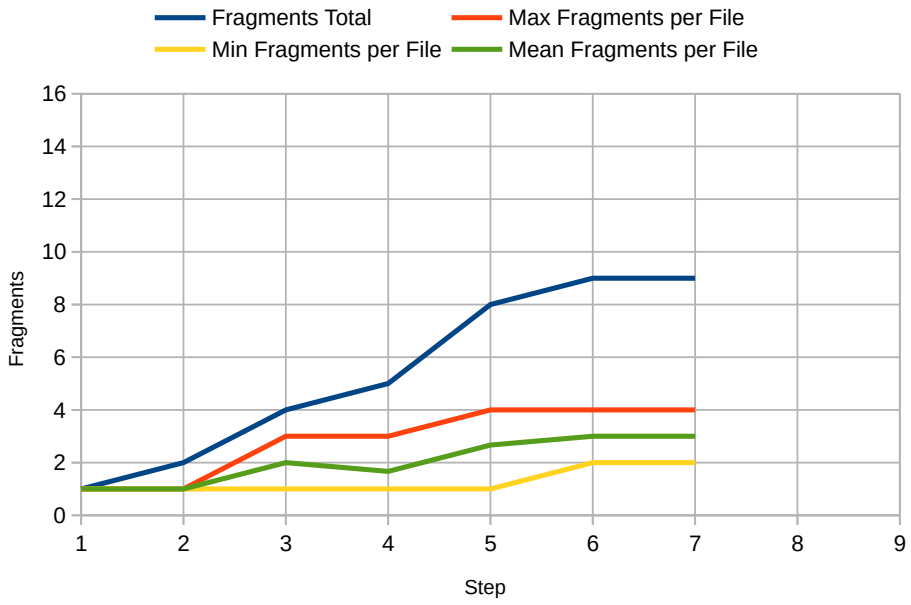


Figure 21: Number of fragments in simulation

2.5 Defragmentation

If there is no other way to improve the storage system, the user must spend time improving performance. Thus, time is invested in a favorable situation to save time on I/O operations later.

There exist a distinction between *online* and *offline* defragmentation. These two modes differ in whether a storage system can remain in production mode during defragmentation. With *offline* defragmentation, the system must be separated from its regular system ('umount'), and a specialized tool starts its optimization process.

The *online* variant processes its algorithms during the normal operating mode. Also, there is a distinction whether a program in the background (*daemon*) automatically reduces the fragmentation incrementally after various write operations or defragments the entire system at selected times (*cron*).

2.5.1 Prerequisites

The defragmentation of a filesystem includes copy and delete operations. Like any I/O operation, these should run atomically and be recoverable in the event of a system failure without any loss of data. As a requirement, the filesystem must provide sufficient spare space.

Oppositely, there is the possibility to use an external data carrier. The defragmenter stores the data pieces temporarily on this medium. In the case of a failure, the filesystem stores a journal for restoring, too.

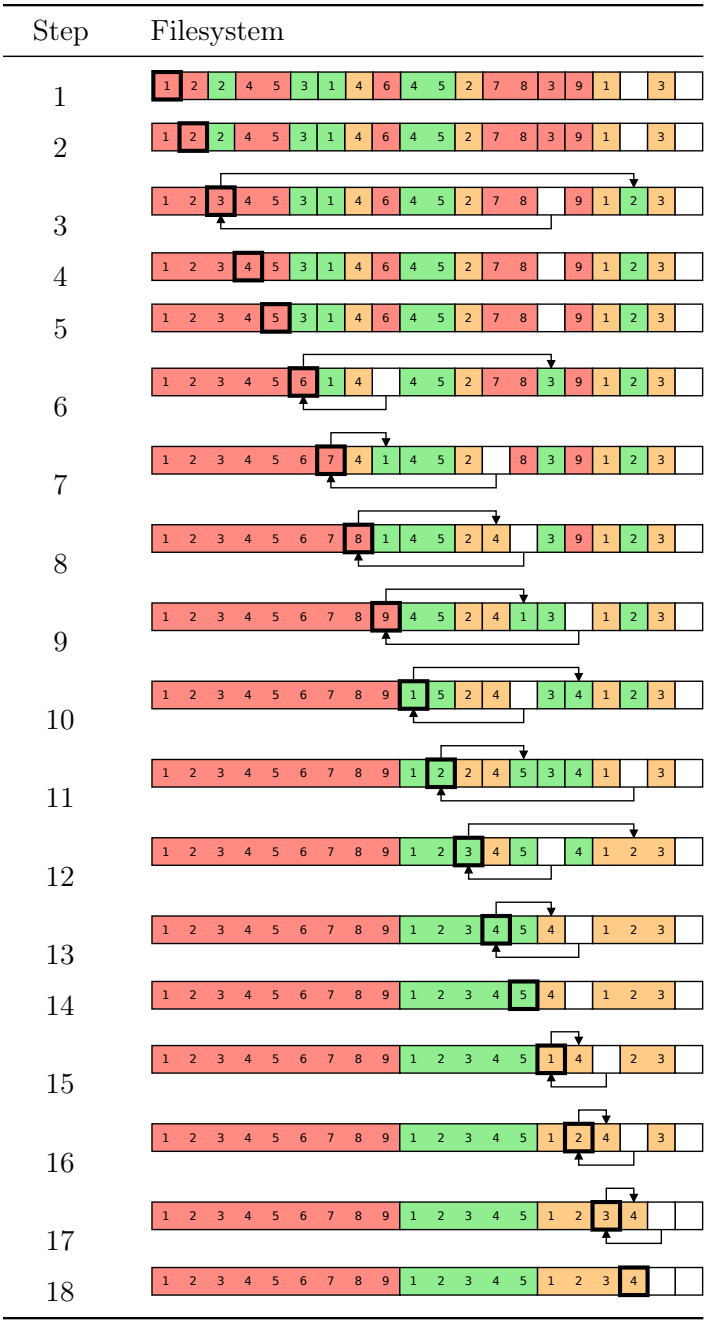
In the worst case, the defragmenter has to temporarily buffer the fragments in the memory to sort the filesystem. This method should only be considered if a backup of the data exists, and an *uninterruptible power supply (UPS)* is available. As a significant benefit, this is the fastest method to defragment files because there is no need to use slow disks.

2.5.2 Defragmentation Procedure

Almost all defragmenters work according to a similar scheme. There is an ascending pointer that addresses the current block. They operate from front to back, i.e., from the low to the high address space. In summary, the steps proceed as follows:

1. The pointer is positioned on block n
2. If block n is not sorted, copy it to the last free block
3. Change the address in the metadata
4. Copy the logical next block to this position
5. Change the address in the metadata
6. Raise the pointer to block $n + 1$

For clarity, the next table shows the defragmentation of the previously simulated data store. The numbers indicate the order of the blocks of a file.



2.5.3 Problems and Optimization Potential

The first problem of the defragmentation process, as shown above, is double copying of nearly all blocks. This process is very time-consuming. Apart from the pending proof of the effectiveness of defragmentation measures on *SSDs*, each flash cell is written twice. Due to the limited number of rewrite cycles, this leads to quicker aging of the disk.

Several proposals to reduce write operations are conceivable. A massively fragmented file does not necessarily have to be completely coherent, as it will probably be split up again during the next update. It makes sense to partially merge the fragments and group them in close physical proximity. As a result, the *seek time* will be sufficiently short.

Moreover, it is unnecessary to copy larger fragments, which drastically reduces the necessitated time for defragmentation.

Another but essential aspect is the insertion of gaps between fragments. This precaution slows down the increase in fragmentation.

However, defragmentation also opens up the possibility of optimizing system performance. By profiling programs and system components, a list of files can be created. Analyzing these track data can reveal groups of files, which are often accessed together.

Besides, some operating systems offer a log file of their prefetch services. By using this data, files can be repositioned on the disk. They are now readable in a single huge read operation with minimal seeking.

3 Research Design and Method

The following chapters focus on the three scientific questions:

1. What specific effects on performance does fragmentation have?
2. How to measure fragmentation?
3. Which practical countermeasures and tools are suitable to address it?

A quantitative study is suitable for clarifying the *first question*. For this purpose, different data carriers, are written using a tool for artificial aging. This program is called *Geriatrix*, and Saurabh Kadekodi published it in *USENIX ATC 2018* [18].

Moreover, the two widely used filesystems *EXT4* and *ZFS* are tested equally. The physical block sizes are matched to the logical ones to achieve higher performance.

Some filesystems provide the degree of fragmentation through their utility tools. To answer the second question, these tools and their values are evaluated to determine their significance.

Since the two filesystems under investigation have different tools and capabilities, different methodologies must be applied to answer the third question.

3.1 Measure Fragmentation

3.1.1 EXT 4

The *EXT4* filesystem offers two different analysis tools: its defragmentation utility `e4defrag` and its free space analyzer `e2freefrag`.

To investigate the current fragmentation, the user can enter a partition or a defined path with privileged access rights. The program `e4defrag` outputs the five most fragmented files. Furthermore, it adds both the actual and meaningful number of fragments in addition to the extent size.

The summary of the analysis includes the complete number of extents and the actual and meaningful quantities. The user also receives the average extent size and the fragmentation score.

The following formula shows the current calculation of the fragmentation score, which is implemented in the toolchain of *EXT4* [29, /misc/e4defrag.c].

$$ratio = \frac{(actual\_extents - meaningful\_extents) \times 100}{files\_block\_count}$$

$$score = \begin{cases} 80 + \frac{20 \times ratio}{100} & \text{if } score > 10 \\ 8 \times ratio & \text{otherwise} \end{cases}$$

The filesystem documentation provides an assessment benchmark against which the user can judge the degree of fragmentation. A high value stands for a high degree of filesystem fragmentation.

The advantage of this metric is that a single scalar value is available at the end of the calculations. This information allows the program to set a threshold value to determine whether a defragmentation is worthwhile. This number also allows the user to get an approximate estimate of the filesystem.

The following code block shows an exemplary output of the tool `e4defrag` with the data just described.

```
$ e4defrag -c /
e4defrag 1.45.5 (07-Jan-2020)
<Fragmented files>                              now/best       size/ext
1. /var/log/boot.log                            36/1              4 KB
2. /var/log/dnf.log                             14/1              4 KB
3. /home/user/.cache/app/cache                  10/1              4 KB
4. /home/user/.var/app/index                    10/1              4 KB
5. /home/user/.mozilla/firefox/a/addons.json    16/1              4 KB

 Total/best extents             644149/597796
 Average size per extent            239 KB
 Fragmentation score                1
 [0-30 no problem: 31-55 a little bit fragmented: 56- needs defrag]
 This directory (/) does not need defragmentation.
 Done.
```

However, the scalar says nothing about the actual amount and distribution of file fragments. Furthermore, the distribution of free disk space is unclear. A meaningful analysis of the current filesystem state, which supports the performance measurements, requires several dimensions.

Therefore, the filesystem provides another tool for evaluation. The program `e2freefrag` analyzes a partition and generates a report about its current free space distribution. For this purpose, it outputs a frequency distribution of free extents with their sizes and quantity.

Additionally, it shows the user some other statistical characteristics, like the number of free blocks/extents, the minimum, maximum, and average available extent size.

The following code block contains an exemplary report for visualization purpose:

```
$ e2freefrag /dev/sda1
Device: /dev/sda1
Blocksize: 4096 bytes
Total blocks: 62196992
Free blocks: 21451242 (34.5%)

Min. free extent: 4 KB
Max. free extent: 2064252 KB
Avg. free extent: 2196 KB
Num. free extent: 39058

HISTOGRAM OF FREE EXTENT SIZES:
Extent Size Range :  Free extents   Free Blocks  Percent
    4K...     8K-  :          7533          7533    0.04%
    8K...    16K-  :          5748         13238    0.06%
   16K...    32K-  :          5094         25465    0.12%
   32K...    64K-  :          4109         43785    0.20%
   64K...   128K-  :          3819         80080    0.37%
  128K...   256K-  :          2745        120191    0.56%
  256K...   512K-  :          2304        204825    0.95%
  512K...  1024K-  :          2170        389098    1.81%
    1M...     2M-  :          2305        850783    3.97%
    2M...     4M-  :          1727       1194814    5.57%
    4M...     8M-  :           713       1020181    4.76%
    8M...    16M-  :           343        907157    4.23%
   16M...    32M-  :           206       1095921    5.11%
   32M...    64M-  :           103       1083890    5.05%
   64M...   128M-  :            73       1640404    7.65%
  128M...   256M-  :            31       1454934    6.78%
  256M...   512M-  :             8        698344    3.26%
  512M...  1024M-  :             6       1225096    5.71%
    1G...     2G-  :            21       9395503   43.80%
```

Due to the histogram, a user can make better statements about the existing filesystem. A significant accumulation of small extents usually leads inevitably to a massive performance loss during write operations.

Nevertheless, it is not possible to make a statement about the performance during reading because the existing files do not necessarily have to be heavily fragmented.

It can be generally deduced that a collection of small extents is worse than an accumulation of large extents.

3.1.2 ZFS

The *ZFS* filesystem also provides administrators with utility tools. The required program is called *ZFS Debug* or short `zdb`. The debugger can be used to query and analyze various information about the state of the given filesystem.

The first relevant record is a report about the distribution of free space. As is usual for a filesystem, *ZFS* also maintains a storage allocation list. To generate an output of the distribution of free space per *metaslab*, use the CLI command `zdb -mm $pool`.

A *metaslab* is a management unit in *ZFS* [27]. These are generated per device *(vdev)* when a pool is created. Up to 200 *metaslabs* are possible, which are homogeneously divided. Each management unit also has a so-called *space map*. These maps contain information about the areas of free space. The following code block shows an extract of the debug output.

```
$ zdb -mm $pool
Metaslabs:
        vdev          0
        metaslabs   116   offset                spacemap          free
        ---------------   -------------------   ---------------   -------------
        metaslab      5   offset   a000000000   spacemap    770   free    36.1G
        On-disk histogram:              fragmentation 12
                         10:   2801 ***********
                         11:   3399 *************
                         12:   3941 ***************
                         13:   2475 **********
                         14:   2401 *********
                         15:   2495 **********
                         16:   3376 *************
                         17:   3785 ***************
                         18:  10772 ****************************************
                         19:   3882 ***************
                         20:   2066 ********
                         21:   1190 *****
                         22:    769 ***
                         23:    456 **
                         24:    210 *
                         25:    105 *
                         26:     32 *
                         27:      4 *
                         28:      2 *
                         29:      2 *
```

The extract contains master data, metrics, and free space distribution of a *metaslab*.

This *metaslab* has the $ID = 5$, starts from the memory area $a000000000_{16}$ *(offset)*, and has 36.1 GB free memory.

The metric is a scalar value for the degree of fragmentation, in this case: 12. Further details will follow. Also, the tool prints a longer list, which is a histogram, afterward.

The description of a line structure is $n : amount_n$. Each row is a so-called *bucket*, which groups together contiguous memory areas of one order of magnitude. The order of magnitude specifies the variable n. The size of the memory area is 2^n bytes, and $amount_n$ indicates the respective frequency.

The minimum size of a *bucket* corresponds to one logical block. This must be specified when creating a pool and should be the same as the volume's physical block size. Common values for the so-called `ashift` are either $9 \rightarrow 2^9 = 512$ bytes or $12 \rightarrow 2^{12} = 4096$ bytes. For each *bucket*, the value $free\_space$ can be determined.

The goal is to achieve the scalar of the degree of fragmentation of a *metaslab*. Therefore, several intermediate steps are necessary. The table $FRAG\_TABLE$ is thus taken from the source code of *OpenZFS* and functions as a weighting factor [14, /module/zfs/metaslab.c].

$$free\_space_n = 2^n \times amount_n$$

$$total\_free\_space = \sum_{i=0}^{31} free\_space_{i+ashift}$$

$$FRAG\_TABLE = \begin{Bmatrix} 100 & 100 & 98 & 95 & 90 & 80 & 70 & 60 \\ 50 & 40 & 30 & 20 & 15 & 10 & 5 & 0 \end{Bmatrix}$$

$$fragmentation = \frac{\sum_{i=0}^{31} (free\_space_{i+ashift} \times FRAG\_TABLE[min(i + ashift - 9, 15)])}{total\_free\_space}$$

This $fragmentation$ value is in the range of 0 to 100. A higher value indicates many small buckets, thus small free storage areas. From this, an administrator can deduce that the filesystem is probably fragmented. Like the *EXT4* toolchain, this scalar only indicates administrators a possible grievance. This value is also unsuitable for research and in-depth analysis.

3.1.3 Analysis Suite: Fraggy

It is fundamental for scientific analysis to get usable information of filesystem fragmentation. For this reason, this thesis includes a new analysis suite called *Fraggy* to gather the needed data. Its development is part of this thesis and was written by me.

The primary focus of this application is on the two filesystems to be analyzed. This collection of tools allows an administrator to evaluate the current state of the filesystem. The suite has two types of tasks:

1. Data acquisition: An executable program records the current distribution of fragments per file. During the collection, the tool aggregates all the data to print them in the machine-readable format *CSV*.

2. Data evaluation: Two different R-scripts read the generated *CSV* files for further evaluation. They are capable of creating graphs for better visualization and human-based interpretation.

Program Description The data aggregator, named `fraggy.agg`, is written in the programming language *C* for performance and the possibility of direct access.

To gather the required data from the filesystem to be analyzed, it has to support a particular system call for device-specific I/O operations. The tool sends `ioctl()` with a request for a so-called *fiemap*. The *fiemap* [5] is a data structure that lists all linked extents on a file basis. To retrieve this data makes it comparatively easy and efficient instead of manually parsing a block map.

Additionally, the program requires a path, which is the starting point of the recursively retrieving of all files and folders.

Subsequently, the algorithm pushes the number of extents per entry into a hash table by using the fragment number as the key. If an entry exists, the function increases its value by one; otherwise, it sets it to one. Rather than reinventing the wheel, this project uses *uthash* [20], which provides *C* macros for hash tables.

At the end of the run, the program only has to sort and loop through the hash map to print a *CSV*.

By the way, the only limit of the program is the availability of *fiemap* data structures. *EXT4* supports it innately; on the other hand, *ZFS* needs currently a particular patch to offer this interface.

The patch to support *fiemap* in *OpenZFS* is described later.

The second part of the suite contains the data analyzers, respectively, visualizers, named `fragmentation.r` and `performance.r`. They are less a program than a statistic script, written in the programming language *R*.

They use the output of the data aggregator `fraggy.agg` to generate statistical core values and graphics, listed below:

- **Mean**: a scalar value for simple comparison
- **Standard Deviation**: a scalar value to extent mean by the amount of dispersion
- **Range**: the range between the minimum and maximum number to receive an impression of the dispersion
- **Shapiro-Wilk Test**: a statistical hypothesis test of normality
- **Wilcoxon Signed-Rank Test**: a non-parametric statistical hypothesis test to compare two related fragmentation distribution. Its calculation is only required if a second distribution exists, and both distributions failed *Shapiro-Wilk Test*.
- **Line diagram**: a visual output to manually see a histogram of the distribution(s)

Building Instructions The used version of `fraggy` is attached to the master thesis and exists on GitHub [33], too. 

The compilation of the data aggregator demands *CMake* and a c-compiler like *GCC*. The execution of the following commands starts the build process in the project directory:

```
$ mkdir build/
$ cd build/
$ cmake build ../
$ make
```

A compiling process is not necessary as the data visualizer is a script. Nevertheless, a functional version of *R* is mandatory. Furthermore, two installed packages from the repository for better visualization are necessitated.

```
$ R --no-save <<< "install.packages(c('tidyverse', 'svglite'))"
```

CLI Parameters The aggregator `fraggy.agg` only needs a path as a sole parameter. It prints the *CSV* directly to standard output. Output redirection on the shell is required to write into a file.

```
$ ./fraggy.agg /mnt/filesystem > frag_distribution.csv
```

To run the visualizer, simply change declared variables at the beginning of the two scripts.

```
$ R --no-save < "fragmentation.r"
$ R --no-save < "performance.r"
```

3.2 Experiment Setup

3.2.1 Computer and its Configuration

The experiment hardware is a server with the following hardware components:

Hardware	Quantity	Model
Chassis	1	Supermicro CSE-815TQC-605WB 1U 4x3,5"
Mainboard	1	Supermicro X11SSW-F
CPU	1	Intel Xeon E3-1230V6
RAM	4	16GB DDR4 PC2666 CL19 ECC unb. Samsung
OS Storage	1	Samsung PM981 256GB M.2 NVMe PCIe 3.0

To quickly swap the storage media for the measurement series in a simple way, the server has four 3.5" removable frame bays. These are usable for 2.5" storage devices, too. The operating system *Debian 10* with kernel version *4.19.118-2* is installed on an internal SSD. Thus it does not influence the measurement series.

The systemwide maximum of file handles is set to $2097152 = 2^2 21$ via `systctl fs.file-max 2097152`; therefore, programs can open more concurrent files.

In recent years, more and more vulnerable CPU bugs have been discovered. Software adaptations, including microcode patches, sometimes cause performance losses. For this reason, the following table lists all affected bugs.

Bug	Abbreviation	CVE
Meltdown [19]		CVE-2017-5754
Spectre Variant 1 [37]	Spectre-V1	CVE-2017-5753
Spectre Variant 2 [37]	Spectre-V2	CVE-2017-5715
Speculative Store Bypass [10]	SSB	CVE-2018-3639
L1 Terminal Fault (Foreshadow) [31]	L1TF	CVE-2018-3615
L1 Terminal Fault (Foreshadow-NG) [36]	L1TF	CVE-2018-3620, CVE-2018-3646
Microarchitectural Store Buffer Data Sampling [6]	MSBDS	CVE-2018-12126
Microarchitectural Fill Buffer Data Sampling [8]	MFBDS	CVE-2018-12130
Microarchitectural Load Port Data Sampling [7]	MLPDS	CVE-2018-12127
Microarchitectural Data Sampling Uncacheable Memory [11]	MDSUM	CVE-2019-11091
SWAPGS [13]		CVE-2019-1125
TSX Asynchronous Abort [12]	TAA	CVE-2019-11135
iTLB multihit [9]		CVE-2018-12207

3.2.2 Used Storage Media

The following test series contrast different characteristics of storage media and their configuration. All used hard drive disks are commercially available, have a 3.5" form factor, use *Conventional Magnetic Recording (CMR)* as the recording technology [15], and are connected via SATA III.

By using only hard disks with *CMR*, current problems with *SMR* are avoided. The difference in performance between *SMR* and *CMR* is most apparent in random-write operations. Whenever a change in *SMR* mode occurs, the controller has to overwrite the entire spiral group. As a result, this mode causes a much higher write effort and hits the write-performance.

The following table lists the tested data carriers. Additionally, it contains the used short names, the model, the capacity, and the revolutions per minute (RPM).

Reference	Model	Capacity	RPM
SG500	Seagate BarraCuda *ST500DM009*	500 GB	7,200
SG1000	Seagate BarraCuda *ST1000DM010*	1 TB	7,200
DC1000	Western Digital Gold *WD1005FBYZ*	1 TB	7,200

The first two hard disks are conventional end-user hard disks. The third one is used in data centers as server hardware to show possible differences in the price level.

The following sections describe the differences to be considered and compared.

Number of Drive Layers Low-capacity drives usually have only one disk layer, which is in contrast to high-capacity drives with several layers.

Engaging in this distinction is whether multi-layer drives are slower or faster than solo-layer drives. Since the read/write heads of all layers are coupled, they all have to follow the same track.

This condition raises the question of whether or not multiple heads can read or write simultaneously. In the worst case, only one head can operate at a time, and frequent track changes will increase the total seeking time.

This test includes the disks *SG500*, *SG1000*, and *DC1000*.

File Sizes The used benchmark tool will read/write lots of files with a specific size. This file size matters considering the available contiguous space areas. Small files will fit in small chunks of free space ranges, but large files will probably split into many fragments.

This test use *10M*, *100M*, and *1000M* as file sizes on all other test runs.

Filesystem As already mentioned, this thesis examines the two filesystems *EXT4* and *ZFS*. Both of them will be almost equivalent tested.

RAID Based on the test about the number of layers, a *RAID* can be used to increase the complexity by one. In this experiment, two identical volumes (*Seagate 500 GB*) are combined to form a *RAID 0*. On *EXT4*, a tool called *multiple disk administration (mdadm)* is used, while *ZFS* already provides this functionality.

To create an *EXT4* software RAID 0, the chunks' size is set to 512 KB. This value is decisive for the configuration of the filesystem. The so-called *stride size* depends on both the chunk and the block size. Furthermore, the *stripe width* must be determined.

$$stride\_size = \frac{chunk\_size}{block\_size}$$

$$stripe\_width = amount\_of\_disks \times stride\_size$$

```
$ mdadm --create --level=0        \
                 --metadata=1.2   \
                 --raid-devices=2 \
                 --chunk=512      \
                 /dev/md0         \
                 /dev/sda1        \
                 /dev/sdb1

$ mkfs.ext4 -L sg500-raid -b 4096 -E stride=128,stripe-width=256 /dev/md0
```

3.2.3 Filesystem: ZFS (Patched)

To use the analysis tool *fraggy* on *ZFS*, a patch has to be applied for now. Patching a filesystems leads to recompiling the whole source code of *OpenZFS*.

Building Instructions

1. Clone the source code from the GitHub repository
   At the time of the measurement series, the commit version *e64cc4954c7862db6a6b4dc978a091ebc3f870da* was the most current and therefore taken.

   ```
   $ git clone git@github.com:openzfs/zfs.git
   ```

2. Install the required compiler and libraries

   - *GCC*: version 8.3.0 (Debian 8.3.0-6)
   - *Kernel headers*
   - *DKMS*
   - *Building environment*: `build-essential`, `autoconf`, `automake`, `libtool`, `gawk`, `alien`, and `fakeroot`
   - *Libraries*: `libblkid-dev`, `uuid-dev`, `libudev-dev`, `libssl-dev`, `zlib1g-dev`, `libaio-dev`, `libattr1-dev`, `libelf-dev`, and `libffi-dev`
   - *Python 3*: `python3`, `python3-dev`, `python3-setuptools`, and `python3-cffi`

3. Apply patch. The used patch is attached to the master thesis or can be found on GitHub [35, *fiemap*-branch].

   ```
   $ git apply zfs.patch
   ```

4. Compile *ZFS*

   ```
   $ cd zfs
   $ ./autogen.sh
   $ ./configure
   $ make
   ```

5. Install *ZFS*
   The *Makefile* enables the option to create installer packages with `make deb`.

3.2.4 Software: Geriatrix

The *Geriatrix* program is a simple tool for artificially aging filesystems. It creates an absolute workload on the desired filesystem, which consists of pseudo-random write operations.

Instead of writing a large number of bytes to a storage system, which would take a long time, *Geriatrix* simulates file creations. To achieve this, it uses the function `posix_fallocate()` (a wrapper around the system call `fallocate()`).

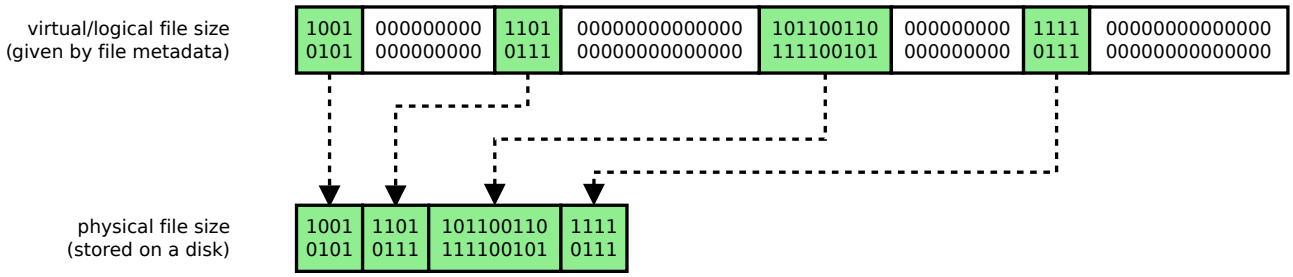


Figure 22: Example of a sparse file

According to the *POSIX* standardization the function `posix_fallocate()` reserves respectively allocates the needed space in a filesystem without writing real content. Therefore, a filesystem would block the wanted file size and handles it as a file without resetting or writing content.

Some filesystems, including *EXT4*, implement this function according to the specification. Both the real and the stored file size (via `stat()` call) correspond to the parameterized file size.

ZFS, on the other hand, implements this system call, but the actual file size corresponds to that of a metablock. It only stores the information about the desired size as metadata. The result of this operation is a so-called sparse file (see figure 22 on page 51).

Therefore *Geriatrix* cannot be used for *ZFS* in its original form. A patch is needed, which converts the function `posix_fallocate()` into a write command. As a consequence, the runtime increases, but the functionality of the simulator remains the same. The used patch is attached to the master thesis.

A summary of the patch: The system call `fallocate()` is replaced by `write()` which repeatedly copies a memory area allocated with zeros.

As a requirement, the target filesystem does not use compression methods such as *run-length encoding (RLE)*. These algorithms have a massive impact on the file size, especially on large blocks containing one symbol, like zero.

The files and folders are created and modified according to a particular pattern. These are manageable by using profiles. The pseudo-random generator used requires a seed, and by using the same one, it enables comparable test runs. The avoiding of multithreading ensures a sequential schedule; otherwise, the operations' sequence will not be identical.

The upcoming experiments target a fill level of 80%. Furthermore, the I/O workload is 100 × partition size.

The suite comes with several profiles. The following test series use the aging profile *Agrawal*. The origin of the modeled profile originates from a five-year metadata study in which the behavior of Windows desktop PCs was investigated [25].

The paper around *Geriatrix* also used *Agrawal* for most of the test series and was thus able to measure significant results concerning *EXT4* in hard disk performance [18, p.698].

Building Instructions

1. Clone the source code from the GitHub repository
   At the time of the measurement series, the commit version *962538f4cb46e86ac3e6f507f8ec718f959373cd* was the most current and was taken, therefore.

   ```
   $ git clone git@github.com:saurabhkadekodi/geriatrix.git
   ```

2. Install the required compiler and libraries

   - *G++*: version 8.3.0 (Debian 8.3.0-6)
   - *Boost*: version 1.67.0

3. Apply patch (only in case of *ZFS*)

   The used patch is attached to the master thesis or can be found on GitHub [34] 

   ```
   $ git apply geriatrix.patch
   ```

4. Compile *Geriatrix*

   ```
   $ cd geriatrix
   $ mkdir build && cd build
   $ cmake -DCMAKE_INSTALL_PREFIX=/opt ../
   $ make && make install
   ```

CLI Parameters According to the README file of *Geriatrix*:

Used Parameters	Description
`-n 2000397868544`	Size of the filesystem in bytes
`-u 0.8`	Level of the filling during a run as a fractional value
`-r 834413475`	Random seed for a Geriatrix run
`-m /mnt/sg500`	Mount point of partition
`-a agrawal/age_distribution.txt`	Path to the age distribution of the aging profile
`-s agrawal/size_distribution.txt`	Path to the size distribution of the aging profile
`-d agrawal/dir_distribution.txt`	Path to the dir-depth distribution of the aging profile
`-x wd2000_age_initial.out`	Path of the output of age distribution
`-y wd2000_size_initial.out`	Path of the output of size distribution
`-z wd2000_dir_initial.out`	Path of the output of dir depth distribution
`-t 1`	Number of threads. Set to `1` to create reproducible disk images, due to the randomness of schedulers during multithreading.
`-i 100`	Workload: $n \times$ partition size
`-f 0`	Fake mode to test configuration
`-p 0`	Idle time between operations
`-c 0`	Confidence interval. A value of 0 implies perfect aging.
`-q 0`	Query before quitting
`-w 10080`	Running time limit in minutes (7 days)
`-b posix`	I/O API backend

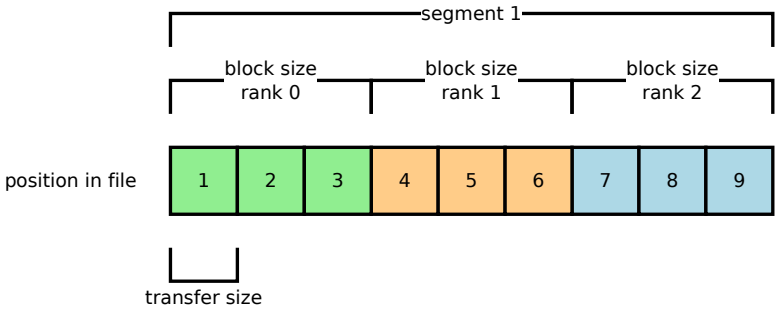


Figure 23: Visualization of *IOR* mechanics [24]

3.2.5 Software: IOR

IOR (Interleaved or Random) is a commonly used benchmark application to evaluate filesystem performance. It includes many interfaces, like the needed *POSIX*. The tool tests both the read and write rates. Moreover, there are two kinds of access strategies: grouped/interleaved and random. Both types of patterns occur very frequently in the context of *High-Performance Computing (HPC)*, usually in numerical applications.

The properties of *IOR*'s I/O behavior are displayed in figure 23 on page 53. The single blocks represent the *transfer size*. In this example, three consecutive blocks represent the *block size* of a *rank*. Since *IOR* can run in parallel, the *ranks* are the respective processes/threads. All nine blocks together form a *segment*.

In the experiment, a single *segment* is written and read sequentially. Therefore the *block size* corresponds to the file size. Different file sizes are used for the test series since fragmentation is related to the free contiguous memory size. 4 KB is a useful transfer size, and the number of repetitions is 100 to obtain enough data points for a meaningful result.

Building Instructions

1. Clone the source code from the GitHub repository. At the time of the measurement series, the commit version *48eb1880e92d8580ad5cab24ecc690526698271e* was the most current and was taken, therefore.

   ```
   $ git clone git@github.com:hpc/ior.git
   ```

2. Install the required compiler

   - *GCC*: version 8.3.0 (Debian 8.3.0-6)

3. Compile *IOR*

   ```
   $ cd ior
   $ ./bootstrap
   $ ./configure --prefix=/opt --with-posix --without-mpiio --without-lustre \
                 --without-hdf5 --without-ncmpi
   $ make && make install
   ```

CLI Parameters According to the usage info of *IOR*:

Used Parameters	Description
`-e`	Perform a `fsync()` operation at the end of each read/write phase
`-a` POSIX	Set I/O API to POSIX
`-b` 10M	Write size per task. This value is adjusted for the different measurement series to `10M`, `100M`, and `1G`.
`-t` 4K	Chunksize per transfer
`-i` 100	Number of repetitions
`--posix.odirect`	Direct I/O Mode to avoid system caches (*EXT4* only)
`-O` summaryFormat=JSON	Output format is JSON (machine-readable)
`-O` summaryFile=sg500_10M_initial.json	Output file. Name contains device name, write size, and the name of the run.
`-o` /mnt/sg500/ior.file	Location of the test file

3.3 Experimental Procedure

The experiments are divided into several steps chronologically, and the experimenter performs two measurement series (*EXT4* and *ZFS*) with each storage medium.

The procedure slightly differs from the used filesystems. In contrast to *ZFS*, the filesystem *EXT4* includes a defragmentation tool. Consequently, *ZFS* uses other strategies to reach the experiments' aim.

3.3.1 Step 1: Device Preparation

Partition Table and Primary Partition Before the experimenter starts the series of measurements, one-time preparation is necessary: Creating the partition table and primary partition.

All disks use GPT as a partition table plus a protected MBR.

```
$ gdisk /dev/sda
> n
> p

Number  Start (sector)    End (sector)  Size       Code  Name
   1            2048      3907029134   1.8 TiB     8300  Linux filesystem

> w
```

Filesystem: EXT4 The creation of an *EXT4* filesystem with default settings is quite simple:

```
$ mkfs.ext4 /dev/sda1
```

Filesystem: ZFS In contrast to that, *ZFS* requires more parameters to build a customized pool.

- `ashift` is a mapping according to the physical block size 2^n; for example, twelve is used on 4K disks and this thesis uses this value by default.
- `normalization` is used to handle special characters in file names. This means: *ZFS* compares the strings by transforming special characters that look alike to avoid user confusion. Usually, the *Normalization Form D (NFD)* is used.
- `compression` must set to *off*. This setting prevents a difference between stored and written file sizes during the performance measurements. The used tool only writes zeros, which could be summarized excellently. However, this would make the measurement series pointless since nothing is written to the target device.

```
$ zpool create -o ashift=12             \
               -O normalization=formD   \
               -O compression=off       \
               -O mountpoint=/mnt/sg500 \
               sg500                    \
               /dev/sda1
```

3.3.2 Step 2: Determine Performance Baseline

To receive the hard disks' raw performance with its filesystem, a measurement in a deflated state is necessary. It is the only way to ensure comparability with all the following measurement data.

All benchmarks are performed by *IOR* with the same parameters to guarantee comparability. Each device and filesystem needs three runs of *IOR* with different file sizes, as mentioned before.

Used variables:

- `$disk` is set to disk name, which is identical to the mountpoint
- `$size` is *10M*, *100M* and *1G*
- `$run` is *initial*

```
$ ior -e -a POSIX -t 4K -i 100                    \
      --posix.odirect                             \
      -b "${size}"                                \
      -O summaryFormat=JSON                       \
      -O "summaryFile=${disk}_${size}_${run}.json" \
      -o "/mnt/${disk}/ior.file"
```

The option `--posix.odirect` enables the *POSIX* `O_DIRECT` flag. This setting instructs filesystems to minimize cache effects [23]. For benchmarks, this option is advantageous to strive to evaluate the real filesystem and not the overall system with caches. Unfortunately, it is not applicable under *ZFS*. The given *POSIX* interface does not implement this property.

Concluding, *IOR* cannot correctly measure the read performance of *ZFS*. The *ZFS* filesystem also has its own *Adaptive Replacement Cache (ARC)*. On the other hand, the measurement tool always reads the same file for a series. Consequently, the cache, i.e. the RAM, is evaluated instead of the desired disk.

To get around this problem, it is unfortunately not sufficient to invalidate the *Page Cache* of the kernel, e.g. by `$ echo 3 > /proc/sys/vm/drop_caches`. Therefore, for every single test series (reading) the *ZFS* pool is exported and imported again. This will invalidate the *ARC* assured.

3.3.3 Step 3: Artificial Aging

Preparation and Execution At this point, the tool *Geriatrix* is ready to use. As stated in the introduction of the program, it creates many directories and files with different sizes according to internal aging profile *Agrawal* until it reaches 80% of the partition size.

Afterward, it modifies all files to perform a workload of 100 × partition size.

Used variables:

- `$disk` is set to disk name, which is identical to the mountpoint
- `$size` is set to the size of the filesystem
- `$filling_degree` is a fractional number, like `0.8`. The used value is determined empirically. Change this value with a workload of `1` to achieve a degree of filling around the targetted 80%. The values that are used: `0.8` at *EXT4* and `0.45` at *ZFS*.

```
$ geriatrix -n "${size}"                    \
            -u "${filling_degree}"          \
            -r 834413475                    \
            -m "/mnt/${disk}"               \
            -a agrawal/age_distribution.txt  \
            -s agrawal/size_distribution.txt \
            -d agrawal/dir_distribution.txt  \
            -x "${disk}_age_initial.out"     \
            -y "${disk}_size_initial.out"    \
            -z "${disk}_dir_initial.out"     \
            -t 1                            \
            -i 100                          \
            -f 0                            \
            -p 0                            \
            -c 0                            \
            -q 0                            \
            -w 10080                        \
            -b posix
```

Evaluation After treating the filesystem chaotically, a fragmentation distribution and a benchmark must be levied. The data aggregation tool `fraggy.agg` determines the distribution of fragments per file.

```
./fraggy.agg /mnt/$disk > $disk_fragmented.csv
```

Repeatedly *IOR* measures the performance of the filesystem in three steps. The variable `$run` from step 2 changes to *fragmented*.

3.3.4 Step 4: Defragmentation

EXT4: Internal Tools This step is only possible with *EXT4* because *ZFS* does not ship a defragmentation tool. Nevertheless, both of the filesystems are defragmented by an alternative method, later.

```
$ e4defrag /mnt/$disk
```

The variable `$run` is defined as *defragmented*.

ZFS: Send Receive An essential advantage of this filesystem is the existence of snapshots. *ZFS* is capable of sending entire snapshots to another dataset, even over a network.

Therefore, an alternative method is worthy of testing: `zfs send` and `recv`. In this test, the tool creates a snapshot, transfers to another disk and back.

```
$ zfs snap $disk1@defrag
$ zfs send -R $disk1@defrag | zfs recv -F $disk2
```

The variable `$run` is defined as *zfs-sent*.

Rsync An excellent alternative to in-place defragmentation is copying files to another storage system (and back). This method operates entirely in userland and is also realizable for a cluster.

During the process, individual nodes or even individual hard disks are locked for writing operations, while *rsync* copies the files to another node or storage drive.

The main advantage of that alternative method is an always freshly initialized filesystem with a low file fragments rate.

```
$ rsync -Pauz /mnt/$disk1/ /mnt/$disk2/
```

The variable `$run` is defined as *rsynced*.

Evaluation The experimenter uses the same methodologies for data collection as in the previous steps. The procedures above include the definition of `$run`.

4 Related Work

Previous scientific papers have also dealt with the aging of filesystems. There are three primary topics: Measurement methods, fragmentation methodologies, and countermeasures.

Measurement Methods The working group of *Alex Conway* introduces a *recursive grep test* in their paper [16]. The functionality of the test is relatively simple. The entire filesystem is read recursively by the command-line tool (CLI) `grep`. Therefore, all directory structures and file contents are read in. This time is measured and normalized by the sum of the read data. This number is determined with the CLI tool `du`. With this methodology, a reading test can be performed quickly and easily on any Linux system.

Furthermore, in their newer paper [17], they add two more measuring methods. The determination of the write performance is done by timing the write operations of the respective applied benchmarks. The sum of the written data also does the normalization. All measurement series or benchmarks omit the flag `O_DIRECT`. Thus the measurement results may have been influenced by the page cache.

As a second addition, they now take a closer look at the free memory space. The authors use the already described program *e2freefrag* on the filesystem *EXT4* to determine the distribution. These measurements were performed regularly during the benchmarks. Thus, the resulting data is mapped on a time curve. The graphs show the changes in the free space gap sizes.

As the last point, there is also the possibility to evaluate the fragmentation of the whole file system. For this purpose, Smith and Seltzer introduced the *layout score* in 1997 [30]. This score evaluates the filesystem by the number of fragments per file (intrafile fragmentation). By definition, a file has a score between 0 and 1. The score consists of the fraction of its blocks, which are optimally allocated, so-called perfectly contiguous.

Conway et al. [16] extend this model. The so-called *dynamic layout score* also evaluates the decision of how and where the files have been fragmented.

This master thesis uses a modification of the *layout score* by Smith and Seltzer. However, it does not plot the temporal course, but the distribution in a snapshot. Unlike the *layout score*, this data is not aggregated but only considers the intrafile fragmentation.

Fragmentation Methodologies A filesystem can be treated differently. All of the listed solutions operate on the application level. Therefore, these programs are executed in userspace.

The first method is taken from Conway et al. [16][17]. They use a version control program called *Git*. Every change within a file is recorded in *Git* in a file in the `.git` folder. This behavior would result in having thousands of files in one folder. To avoid performance issues during directory listing, *Git* creates subfolders that behave like a hash tree.

The idea now is first to copy an existing repository to the disk. After that, the version history is processed while calling many `git pull` requests. Consequentially, the files change so often that they inevitably fragment quickly. According to the paper [16], the reading speed is reduced by a factor of 30 after a thousand operations.

The other presented possibility is based on a realistic scenario. The authors consider a *mail delivery agent (MDA)*, more precisely the program *dovecot*. It is a widely used e-mail server that offers the *IMAP* service. The e-mails are stored in separate text files in the so-called *Maildir* format.

If many e-mails are transported back and forth over time, there is a chance of fragmentation due to the sheer number and fluctuation of small and large files.

Furthermore, they implement a self-made «free-space fragmentation microbenchmark». This microbenchmark creates a lot of small files, deletes them randomly, and then creates new ones. In contrast to the methods mentioned above, it is an independent benchmark in a separate program.

The workgroup of *Alex Conway* studied various filesystems, including *Btrfs* (a *Copy-on-Write* system) and *EXT4* [16][17]. The measurements after aging by «free-space fragmentation microbenchmark» show that read and write performance is significantly higher at a great fill level. Compared to the initial measurement: the writing cost of *EXT4* has increased by 40% and 25% in the case of *Btrfs*. The reading performance is not affected by the high fragmentation of free space because there is no intrafile fragmentation.

However, this master thesis uses a different tool from another paper: *Geriatrix* by Saurabh Kadekodi [18]. The algorithms and profiles used here to age an entire file system are not limited to intrafile fragmentation. In contrast to the application-based benchmarks previously mentioned, the free space is transformed, too.

In order to achieve a realistic aging simulation, the authors have created different profiles. These profiles contain behavioral patterns of how files should be modified over time. For this purpose, they have used different studies dealt with change patterns of real-world storage systems over several years.

Countermeasures The prevention of aging symptoms can be categorized into two categories. The first one considers the various mechanisms that filesystems already implemented by design. Secondly, there is the possibility to optimize an existing system with the help of external tools.

According to [16], there are different design possibilities of filesystems to mitigate aging:

1. **Grouping of cylinders or blocks**
   Several files are often accessed together. The idea implies to store them in a group. For example, continuous files in a directory are read together. Consequently, grouping reduces the *seeking time*.

   ZFS already implements this method. For this purpose, it uses the so-called *metaslabs*. If a new file is created, *ZFS* assigns rank numbers to the different areas. Among other things, this number evaluates the proximity to files at the same hierarchical level. The highest number with the corresponding available storage space is awarded.

2. **Extents**
   Using *extents* is also a useful way to prevent fragmentation. They also reduce the overhead of the address bookkeeping compared to the direct addressing of blocks. Filesystems, such as *EXT4*, can dynamically define the *extents* size during the write operations. Noting the start address and memory length is the only remaining task.

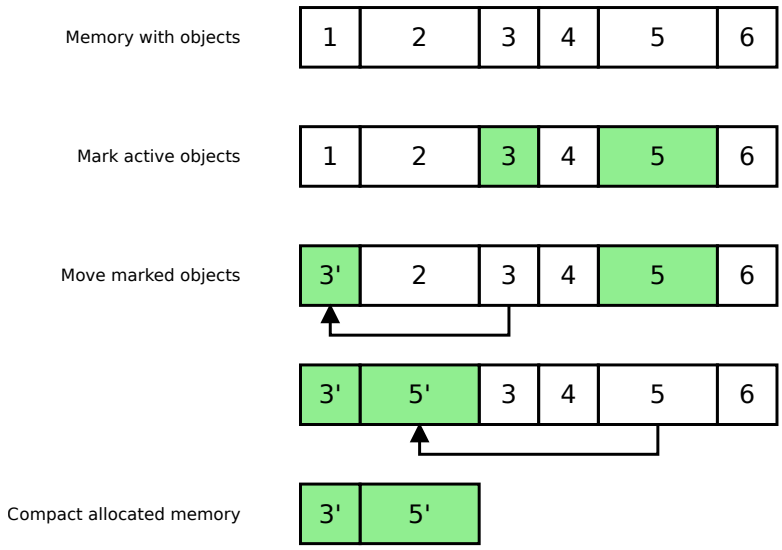


Figure 24: Simple *heap compaction* algorithm

3. **Delayed allocation**
   To determine a reasonable size for the *extents*, write accesses are often buffered in memory. The incoming, continuous data is collected for a few milliseconds to allow the filesystem to reserve larger *extents*. This mechanism is more or less essential for filesystems based on *extents*. Otherwise, only small free memory areas remain after a short time.

4. **Small files and metadata** can be written directly to the administration tree instead of the data area. The leaves of the tree structures usually have a small data area. For example, *ZFS* stores the contents of files up to 512 bytes directly in the tree. This approach makes the address management obsolete and optimizes further access to the storage system. Also, these tree structures usually do not fragment, as they balance and reorder themselves.

However, it is not always possible to prevent aging through the right design decisions. A research group at *IBM* has published a *Research Report* [28], which describes an algorithm for defragmenting *ZFS*.

The paper confirms that defragmentation is a real-world problem in long-term systems. However, they give other reasons besides defragmentation of why data blocks have to be moved: shrinking filesystems or eliminating bad blocks.

Instead of the resource-intensive tracking of back-references, they consider an algorithm class, which is preferably used for garbage collectors. Runtime environments like *C#* or *Java* use *heap compaction* to free the allocated memory area from no longer needed objects. They mark all address ranges of active objects and move them one after the other to the frontmost memory area (simplified in figure 24 on page 61).

The proposed algorithm from *IBM* includes a 3-phase model:

1. **Discovery**
   The file tree is scanned to find all active *extents*. They are stored in an «x-tree» created for this purpose afterward.

2. **Decision**
   The algorithm decides which *extents* should be moved and look for a suitable area in the next step.

3. **Movement**
   In the last phase, the data is transferred, and the pointers in the address-tree are updated.

The algorithm intends to move the data *extent* by *extent* to a new storage device. In contrast to an *in-place* move, there is a much higher chance that more *extents* can be moved and grouped sensibly. The goal is the high availability of large, contiguous, free storage areas.

The paper also describes the possibility of executing the algorithm «online» (while updates are still being written). For this purpose, an «r-x-tree» logs the corresponding new extents and maps the addresses.

The defragmentation of *ZFS* in this master thesis works similarly at the end of the day. The data is also transferred to a new storage system. Unfortunately, only read access is allowed during the process. Also, an in-place solution is not provided.

5 Analysis of the Results

This chapter deals with the measurement results of the described experiments. In the first step, the acquired data are either visualized as graphs or displayed as tables. In the following, a respective description of the data from each measurement series is given. The interpretation and the meaning of the results happen in the following chapter (discussion).

According to the descriptions of the experiments, many different data are generated from the measurement series. Two aspects are important: The filesystems' performance and the number of fragments over the various states.

As an overview, the following table shows the variety of runs. There are measurement values for each variation.

Filesystem	Drive	Transfer size	State	IO
EXT4	Seagate 500 GB	10 MB	initial	read
ZFS	Seagate 1000 GB	100 MB	fragmented	write
	WD Gold 1000 GB	1 GB	defragmented (*EXT4* only)	
			zfs-sent (*ZFS* only)	
			rsynced	

5.1 Analytical Methods

The benchmark program *IOR* generates 100 measuring points per survey for each type of IO. The data points are the average data rate per run. The read rate's performance measurement for the filesystem *ZFS* is done with the same file for technical reasons. The *EXT4* filesystem's read and write rates are measured alternately and always with new sample files.

The comparability of the data is achieved by grouping the IO types and states per transfer size, drive, and filesystem. Furthermore, a **box-and-whisker plot** is used for visualization (compare to figure 26 on page 66). These graphs are ideal for showing the dispersion of the data with essential statistical characteristics: the minimum, the maximum, the sample median, and the first and third quartiles.

To better evaluate the results, further statistical variables and tests are added (compare to table 3 on page 108 and table 4 on page 109):

The **mean**, unlike the median, takes into account the outliers. There are technical reasons for their occurrences, and the values based on real measurement data. Therefore, it is essential to take these into account. However, the significance of the average without the **standard deviation** is almost negligible. This value indicates the dispersion of the mean. Additionally, the range of the data between the minimum and the maximum is also added.

Finally, the states are compared in pairs to see if there is a significant difference. The selection of the next test requires the determination of whether the data are normally distributed or not. The probability value of the **Shapiro-Wilk test** indicates the existence of a normal distribution. This test is necessary for further statistical evaluation of the significance. By definition, the available tests require either normally distributed or not normally distributed data.

The hypothesis is that the following data are not normally distributed. Since the measured data are also interdependent (repeated measurements for changed parameters), the **Wilcoxon signed-rank test** is used. Its p-value indicates the significance of a difference between the pair.

As a result, it is possible to determine a change in the data rate, which indicates causality rather than randomness at a very high significance level $p < 0.05$.

The filesystems **fragmentation** is represented by frequency polyline charts (compare to figure 25 on page 65), respectively density graph. The x-axis describes the number of fragments per file. A file consists of at least one fragment. Therefore, the measurement points cannot start from zero. It is important to note that the lines connect the measuring points. Therefore, straight lines between two measuring points do not show any frequency since no data is available for these fragmentation degrees.

The position of the measuring points on the y-axis represents the frequency. As an example, the fictitious measuring point $(50_x, 15000_y)$ means that 15000 files exist, each consisting of exactly 50 fragments. Due to the measurement results, the y-axis scales logarithmically log(10). Consequentially, the visualization of low frequencies is more perceptible.

The diagrams contain all measurement series of a disk per filesystem for direct comparison. The lines are differentiated by color. The *EXT4* filesystem measurement includes three different runs in the given order: «fragmented», «defragmented» and «rsynced». The experimental series with *ZFS* is similar: «fragmented», «zfs-sent» and «rsynced».

In addition to the diagrams, there is a table with statistical data for each filesystem (see table 1 on page 65). The first column defines the used storage medium, and the second one the conducted measurement series. The next columns contain the following statistical values:

- Sum of files consisting of exact one fragment
- Sum of files consisting of more than five fragments
- Sum of files consisting of more than ten fragments
- The highest number of fragments in a file

5.2 Measurement Results

5.2.1 EXT4: Fragmentation

The figure 25 on page 65 describes the fragmentation of the *Seagate 500 GB* with the *EXT4* filesystem. All three runs decrease exponentially from the first measurement point. As expected, the vast majority of the files in all runs consist of a single fragment and form the highest frequency, which is approximately 3.5×10^6.

The decrease of the three runs in the range of 1 to 20 varies: «rsynced» decreases most, followed by «defragmented» and «fragmented». The highest number of fragments for «rsynced» is 35, as can be seen in table 1 on page 65.

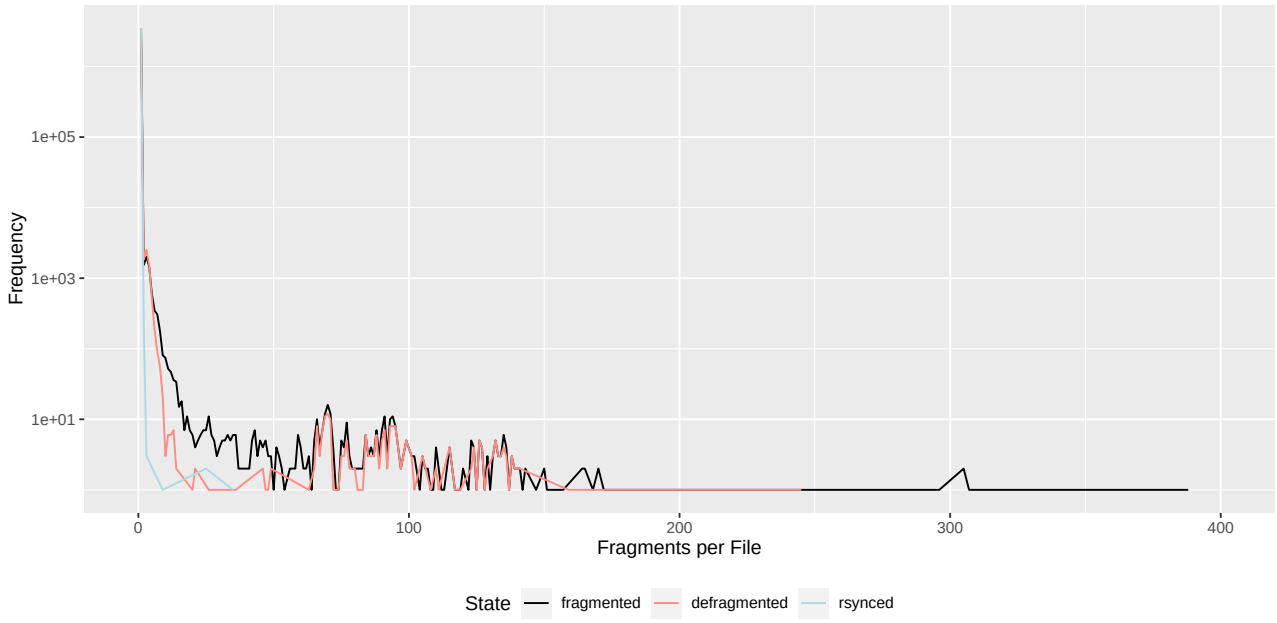


Figure 25: Seagate 500 GB (fragmentation of *EXT4*)

Drive	State	SUM == 1 Frags	SUM >5 Frags	SUM >10 Frags	MAX Frags
sg500	defragmented	3494580	607	253	245
sg500	fragmented	3494253	1733	748	801
sg500	rsynced	3501371	4	3	35
sg1000	defragmented	6990854	281	239	555
sg1000	fragmented	6990675	836	536	1060
sg1000	rsynced	7005415	8	8	69
wdgold	defragmented	6990515	389	230	603
wdgold	fragmented	6990339	1354	605	834
wdgold	rsynced	7005415	8	6	44
sg500-raid	defragmented	6752975	224	224	568
sg500-raid	fragmented	6752495	555	542	983
sg500-raid	rsynced	6761110	5	4	33

Table 1: Statistics about *EXT4* fragmentation

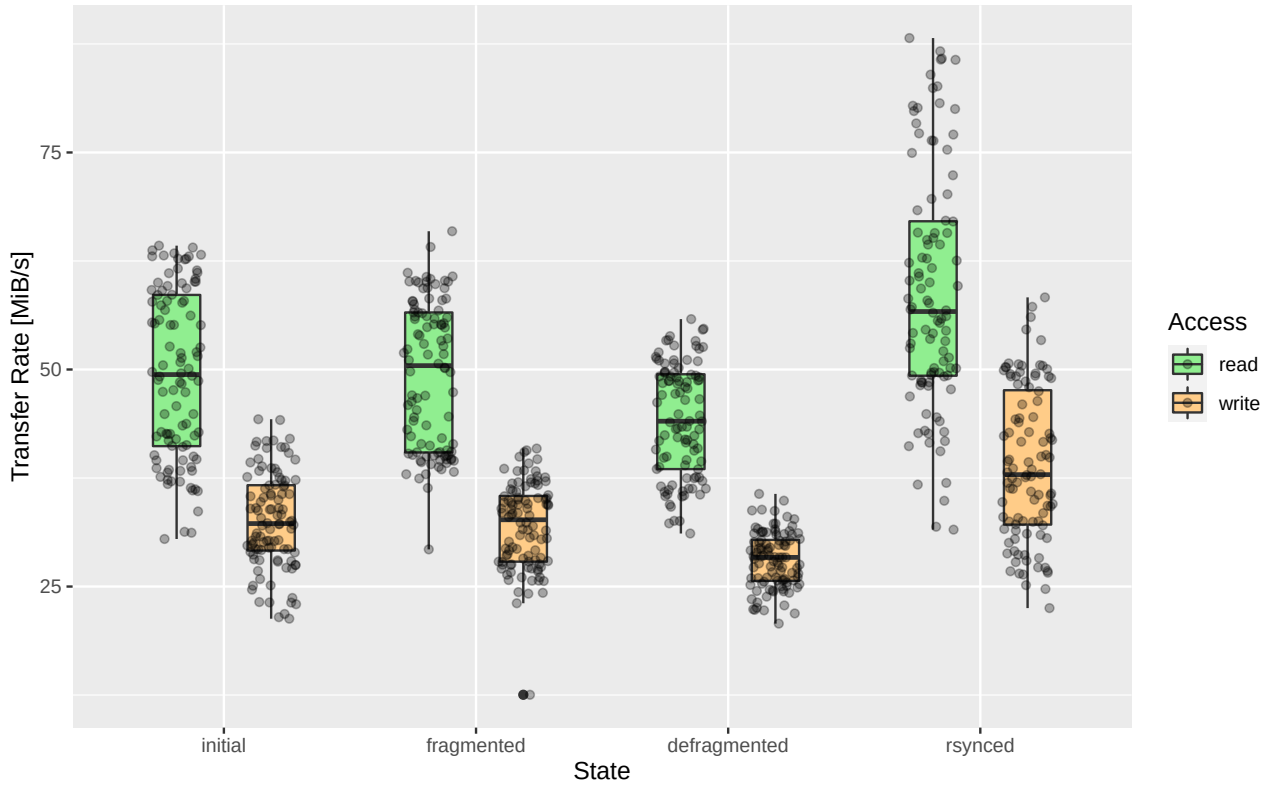


Figure 26: Seagate 500 GB (10 MB file size on *EXT4*)

In the range, 65 to 140, the two runs «fragmented» and «defragmented» are quite similar and are quite erratic. The frequencies range from zero to ten. The frequency of the high fragment degrees is tiny but present. The «defragmented» line has its highest value with 245 and «fragmented» closes with 801 fragments.

In summary, the number of higher fragmentation levels decreases exponentially and decreases even more after defragmentation and even more after using *rsync*.

The measurement results of the other two storage media (*Seagate 1000 GB* and *WD Gold 1000 GB*) are very similar. Therefore, a combined evaluation is appropriate. The referred diagrams are displayed in figures figure 35 on page 96 and figure 36 on page 96. As with the *Seagate 500 GB*, the curves of the other two disks decrease exponentially.

The number of files with a fragment in all six runs is approximately 7×10^6. Furthermore, the runs in the range 1 to 20 fall in the same order. The significant difference is in the remaining runs. In the range 20 to 190 and 260 to 350, there are rarely files in «defragmented». The frequency of the «fragmented» series oscillates irregularly between zero and five files. The range of 190 to 260 is slightly increased for both runs (up to ten files) and is related. There is also a significant accumulation in the range of 350 to 370.

The two hard disks with the higher capacity and the associated number of spinning layers within the storage medium have a significant number of files with a high degree of fragmentation.

5.2.2 EXT4: Performance (10 MB file size)

Figure figure 26 on page 66 shows the measured values of the performance measurement. The tested hard disk is the *Seagate 500 GB* with the filesystem *EXT4*. The four different runs are visually separated after the access operation.

The differences between the first three runs are small. The read and write speeds for «initial» and «fragmented» are considered to be equal. According to the statistical analysis in table 3 on page 108, the average read rate is about 49 MiBps with a deviation between 8.5 MiBps and 9.6 MiBps.

The measurement results of the write rate in the first two cases are also similar. The average is 32.0 MiBps and 32.6 MiBps with a rather small standard deviation (5.0 MiBps and 5.5 MiBps).

The first moderate difference appears in the «defragmented» series. The reading rate is lower (on average at 44 MiBps), but also the scatter of the measuring points (see standard deviation and range). The same behavior is seen in the write rate, which averages 28 MiBps.

During the last run («rsynced») the data obtained shows that the average read and write rates are significantly higher than the other three runs. On the other hand, the different measuring points' dispersion is significantly higher (approximately 57 MiBps for reading accesses and 36 MiBps for writing accesses).

The initial measurement series were always performed with a freshly created filesystem. With a small file size, the overhead becomes noticeable. The filesystem's data structures are probably not very efficient if they have barely been used so far. It is also likely that these measurement files were written in the higher address regions, id est, in the inner tracks with a lower trajectory speed. If the hard disk is full, the inner tracks are already occupied, and the file fragments are probably located in the outer tracks.

The result of a simple read and write benchmark using *GNOME Disk Utility* is shown in figure 27 on page 68. This program divides the complete address range of the disk into 1000 segments, and in each segment, a sample of 100 MiB is written and read. The result shows that both transfer rates decrease with the increasing address. This impressive visualization shows that addressing starts at the outer tracks and moves to the middle while the performance decreases.

The **Shapiro-Wilk test** shows for all data sets (including the other runs not yet described with *EXT4*) that there is no normal distribution of measurement points. Therefore, the **Wilcoxon Signed-Rank Test** can be applied. The test compares each measurement series with its predecessor.

The statistical data on the *Seagate 500 GB* from table 4 on page 109 also show that the difference between «initial» and «fragmented» is virtually non-existent ($p \approx 0.96$ and $p \approx 0.48$). The differences between the next runs are highly significant ($p < 0.025$).

After *Geriatrix* has fragmented both the files and the free space, there will probably be enough gaps for a small 10 MB sample. Therefore the performance after fragmentation is similar for small sample sizes.

The average of «defragmented» is 10% below that of «fragmented» for reading and 12% for writing accesses. The defragmentation tool reduces the fragmentation of the files but increases the fragmentation of the free space. Only small gaps seem to exist, which leads to fragmentation of the sample and a decrease in performance.

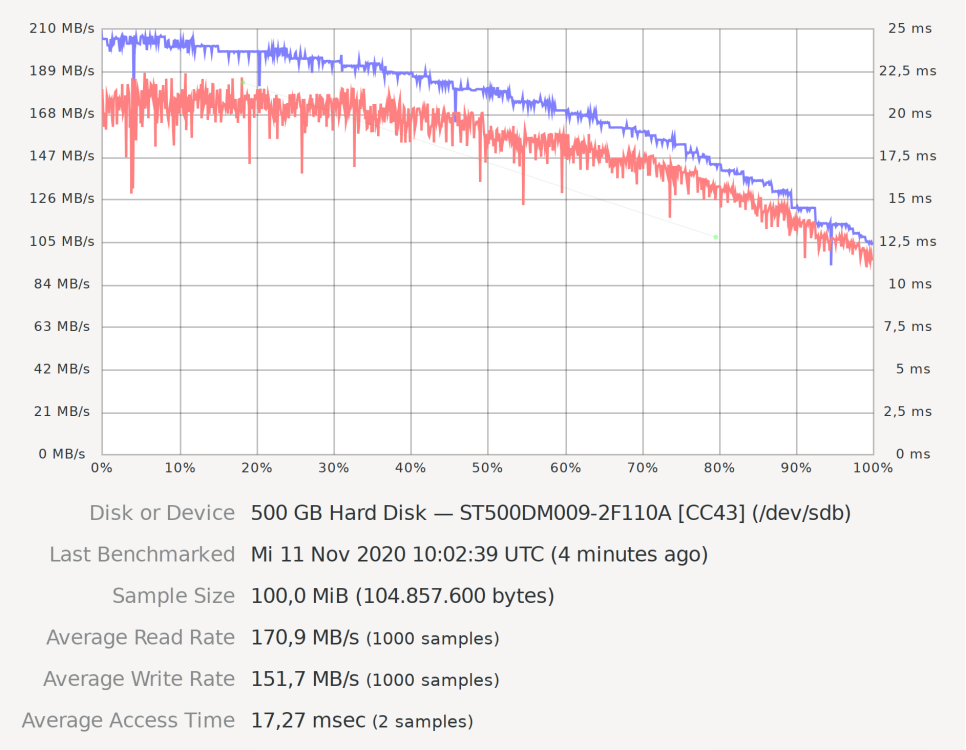


Figure 27: Benchmark of Seagate 500 GB with *GNOME Disk Utility*

The difference between «defragmented» and «rsynced» is confirmed here with an increase of 33% (reading) and 39% (writing). Since the data has been written to a clean filesystem, this performance increase is not surprising.

In summary, the difference is not measurable in the case of intense fragmentation, but after defragmenting. The best performance with dispersion shows up after copying all data to the hard disk.

5.2.3 EXT4: Performance (100 MB file size)

The measurements with a file size of 100 MB differ significantly from those just described. At first glance, the figure 28 on page 69 shows a decreasing transfer rate in the first three runs for both read and write accesses.

As the data from the table 5 on page 110 show, the average read rate during the initial run is with 92 MiBps. The interquartile distances and ranges are tremendous for this result and the next read accesses.

In addition to lower transfer rates, the «fragmented» measurement showed a significantly greater dispersion (standard deviation of 15 MiBps). Accordingly, the average is just under 70 MiBps. After defragmentation, the dispersion and the reading rate decreased with an average rate of 52 MiBps.

The pattern is similar to the write rate. The initial run shows a substantial accumulation at the upper quartile and an average of 65 MiBps. The transfer rate also drops in the next two steps to 48 MiBps and then to 31 MiBps. The «defragmented» run also shows a low dispersion here (4 MiBps).

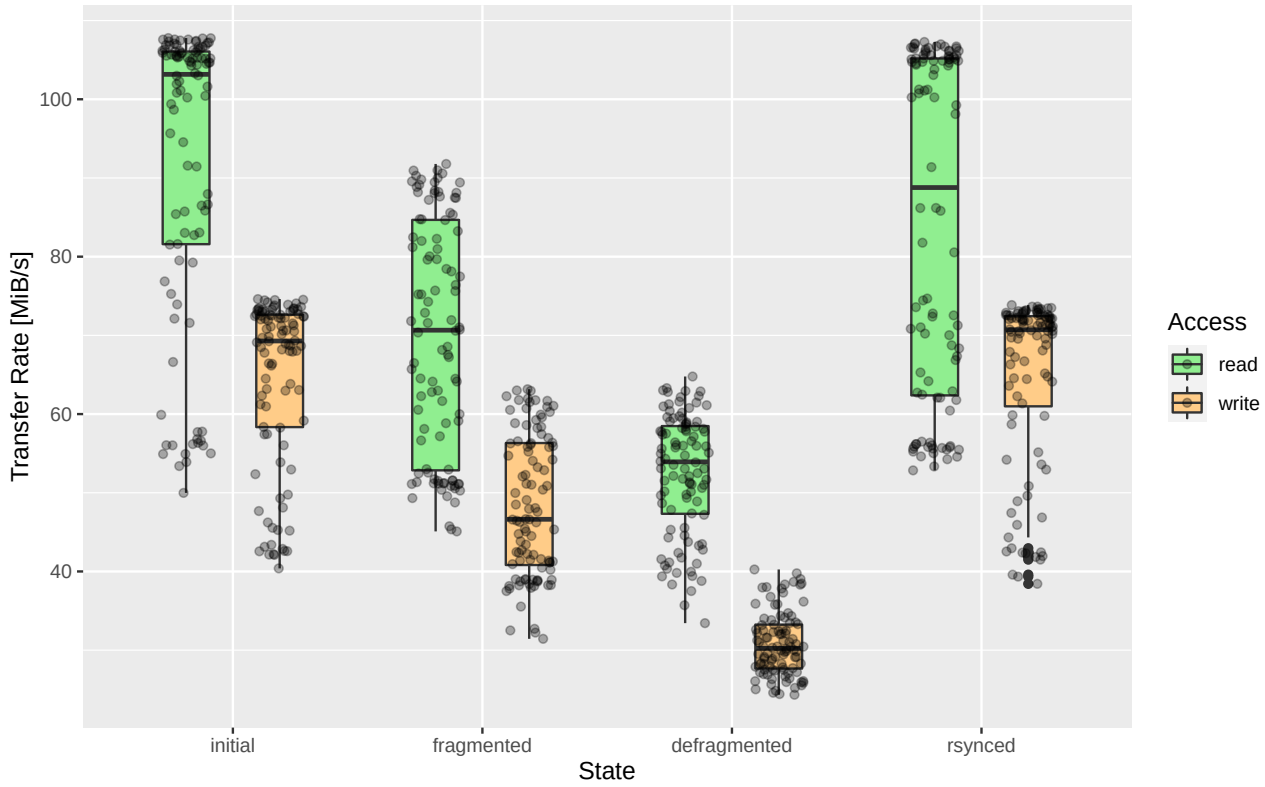


Figure 28: Seagate 500 GB (100 MB file size on *EXT4*)

The last run («rsynced») shows a notable increase in transfer rates. The write rate is on the same level as the initial run. Furthermore, the read rate here scatters most with a standard deviation of 21 MiBps and a range of 57 MiBps. There are indications for the formation of two clusters. The first one is located at the upper quartile at around 105 MiBps. By contrast, the second one can be found below the lower quartile at about 55 MiBps.

All results of the **Shapiro-Wilk Test** show a non-normally distributed data set. The **Wilcoxon Signed-Rank Test** continues to show highly significant differences between runs.

The results from the measurement with the *Seagate 1000 GB* are the same over time. As can be seen in the graphic figure 42 on page 99, there is a raised amount of cluster formation with several runs:

- «initial» read and write: Upper quartile and minimum
- «fragmented» read: Upper and lower quartile
- «defragmented» read: Upper quartile
- «rsynced» read and write: Upper quartile

The hard disk *WD Gold 1000 GB* (figure 45 on page 101), on the other hand, has significantly higher speeds for I/O operations. The average read speed in initial state is 145 MiBps. It is also noticeable that the progression and differences between the reading and writing operations differ from the other measurement results.

The average read and write rate for the «fragmented» run is almost the same (103 MiBps versus 104 MiBps). The next run is even more remarkable, because the write rate is higher than the read rate (92 MiBps) with 76 MiBps.

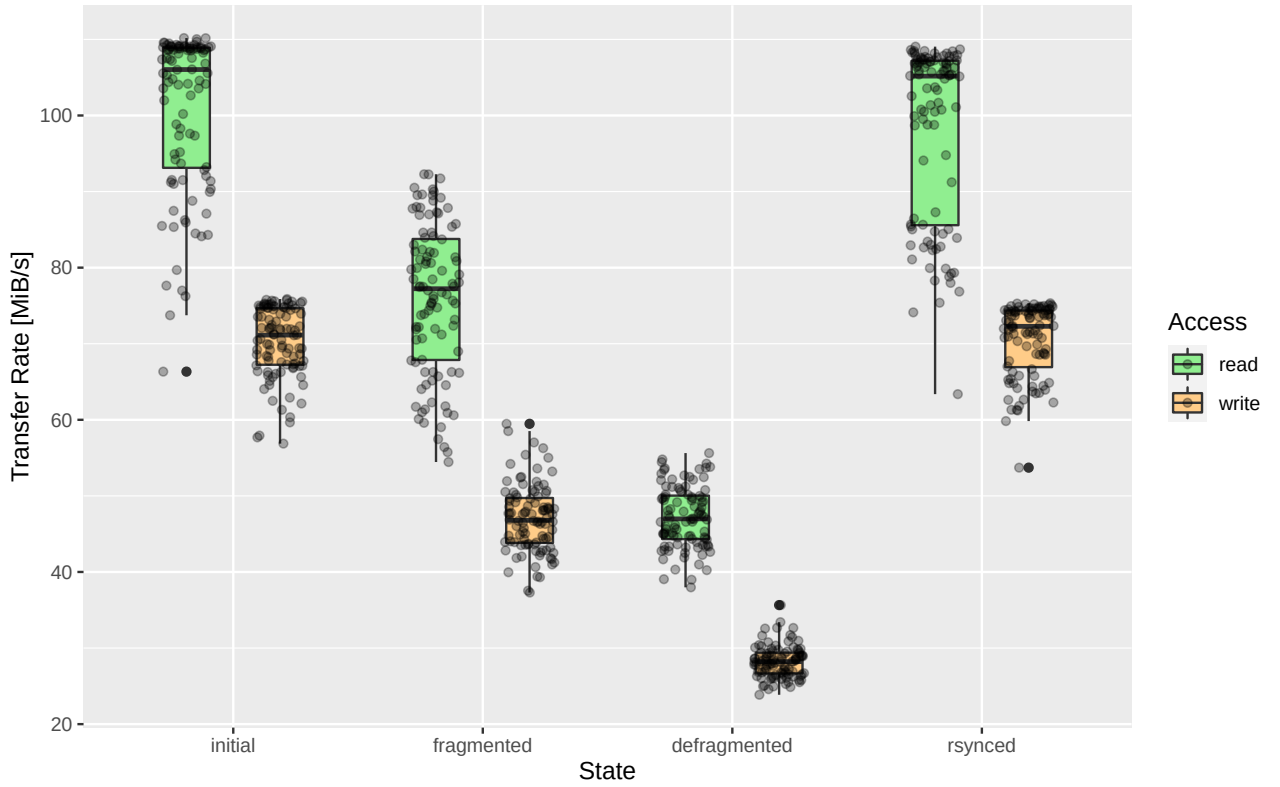


Figure 29: Seagate 500 GB (1 GB file size on *EXT4*)

However, the decreasing trend in the first three runs is also present here. Also the last run «rsynced» shows that the speeds are very similar to the initial measurement series. The read rate is slightly higher with 148 MiBps and the write rate is the same.

This disk also shows clustering:

- «initial» read and write: Upper quartile and minimum
- «fragmented» write: Maximum
- «defragmented» write: Maximum
- «rsynced» read and write: Maximum and minimum

Nevertheless, why do these clusters form?

The reason for this phenomenon is probably due to the design of the hard disks with higher capacity. These have several rotating disks. The best performance is achieved when the data has to be read or written from only one of these disks, preferably from the outer tracks.

The read and write rate is likely to be significantly worse when more than one disk is needed. Since the read and write heads do not operate simultaneously at different levels, the arm must be repositioned each time it is changed. Since this *seeking time* is relatively high, the performance can vary and form clusters in the evaluation. The verification of this assumption requires that a block trace determines the samples' corresponding addresses according to the respective benchmark.

5.2.4 EXT4: Performance (1 GB file size)

The staircase effect of the first three states from the last measurements is also evident in the next results. Also the speed differences compared to the previous measurement series with a file size of 100 MB are not very different.

The only difference to be emphasized in this measurement series is the smaller span of the measurement points. This is visible on the three hard disks both graphically (figure 29 on page 70, figure 43 on page 100 and figure 46 on page 101) and in the statistical evaluation (table 7 on page 112 and table 8 on page 113).

5.2.5 EXT4: Fragmentation and Performance of RAID 0

The fragmentation of the *EXT4* filesystem in a software *RAID 0* differs from the other measurement results.

The graph (figure 37 on page 97) clearly shows how few fragments exist in the sum in the RAID. Most (slightly below one million) files consist of one to three extents after being fragmented. Nearly 1000 files consist of two extents, which no longer exist after defragmentation.

High fragmentation starts at about 120 and stops at 345 extents. This irregular pattern (between 0 and 10 extents per file) is almost identical in both states. Probably there was no suitable free space for fragment reduction for these files. Only the tool *rsync* reduces the fragmentation of those files with high fragmentation. Nearly all files consist of one extent. Note that these results only show the extents of the filesystem. The actual fragmentation pattern caused by the software RAID is entirely transparent.

The performance of the RAID system is quite mixed and seems to depend on the sample size.

The graph with the 10 MB sample size (figure 47 on page 102) shows minimal changes between the runs. The average read accesses vary between 45 and 47 MiBps with a standard deviation of about 10 MiBps. Furthermore, there is no significance between runs, as shown in table 3 on page 108. The first pair has a significance level of 0.1, which is higher than the next two: 0.61 («fragmented» versus «defragmented») and 0.48 («defragmented» versus «rsynced»).

The write operations average out 29 to 33 MiBps with a deviation of 5 to 7 MiBps. The differences between the runs are marginal, again. Notwithstanding, the first two runs («initial» and «fragmented») tend with an increased significance towards normal distribution. Therefore, an interpretation of the *Wilcoxon Signed-Rank Test* is not appropriate.

Due to the small differences, the sample size does not appear suitable to measure possible fragmentation evidence. Thus, the evaluation of the 100 MB experimental series follows (figure 48 on page 102).

The measurement results show unusual behavior, which only occurs during a 100 MB run. Nearly every measurement path shows clustering in the lower quartile and the upper whiskers (except «write defragmented»). The evaluation of the average speed or the median mean with its broad range induced by this cluttering is meaningless. A table with a visually estimated average speed of each cluster gives a slightly better overview:

Access	State	Estimated Average Transfer Rates (MiBps)
read	initial	38 + 65
read	fragmented	55 + 105
read	defragmented	52 + 100
read	rsynced	55 + 105
write	initial	38 + 58
write	fragmented	40 + 71
write	defragmented	–
write	rsynced	40 + 71

It was apparent before the measurement that the speed with the selected measuring method would be significantly slower. The I/O syscalls with the flag `O_DIRECT` lead to a significant slowdown because every write operation must be written consistently on disk before returning. This configuration, combined with the block size of 4 KB per operation, leads to a massive overhead.

An interesting aspect, in this case, is the processing by the software RAID. Unfortunately, it is not known whether *mdadm* processes the direct write command correctly. Furthermore, the chunk size is 512 KB, which is 128 times larger than the write block. If caching is not allowed, then *mdadm* would have to read the complete chunk for each write operation of a 4 KB block, replace the addressed 4 KB part and save it again, which is relatively inefficient.

Why this clustering occurs, or on which level this phenomenon is caused, could not be clarified. By further tests, it might be possible to ascertain this behavior. Each of the following points describes a measure that could be changed in the setup. They should not be combined!

1. Benchmark *EXT4* on a disk with extended attributes, such as `stride`, to identify the filesystem as the source of the problem
2. Replace *mdadm* with a hardware RAID controller, as they perform much better and do not cause further syscalls.
3. Do not set the `O_DIRECT` flag.
4. Set the block size of 4 KB to something larger, for example, the chunk size of 512 KB.

With a larger sample size of 1 GB (figure 49 on page 103), the described phenomena no longer occur. Due to the comparatively high scattering of the measuring points, it can be assumed that the different speeds also happen. However, one measuring point represents the average speed; accordingly, fast and slow operations could compensate for the final result.

The measurement results show consistently poor performance, both when writing and reading. Subsequently, the differences between the states are no longer apparent since all measurements underperform. For this *EXT4* RAID configuration, the benchmark must be run under different configurations, or a completely different one must be chosen.

To get a comparison to the possible performance of the RAID, a simple benchmark with *GNOME Disk Utility* is helpful (figure 30 on page 73). As with the *Seagate 500 GB* measurement, 1000 measurement points with 100 MB sample size were also used. The course of the graph is expected.

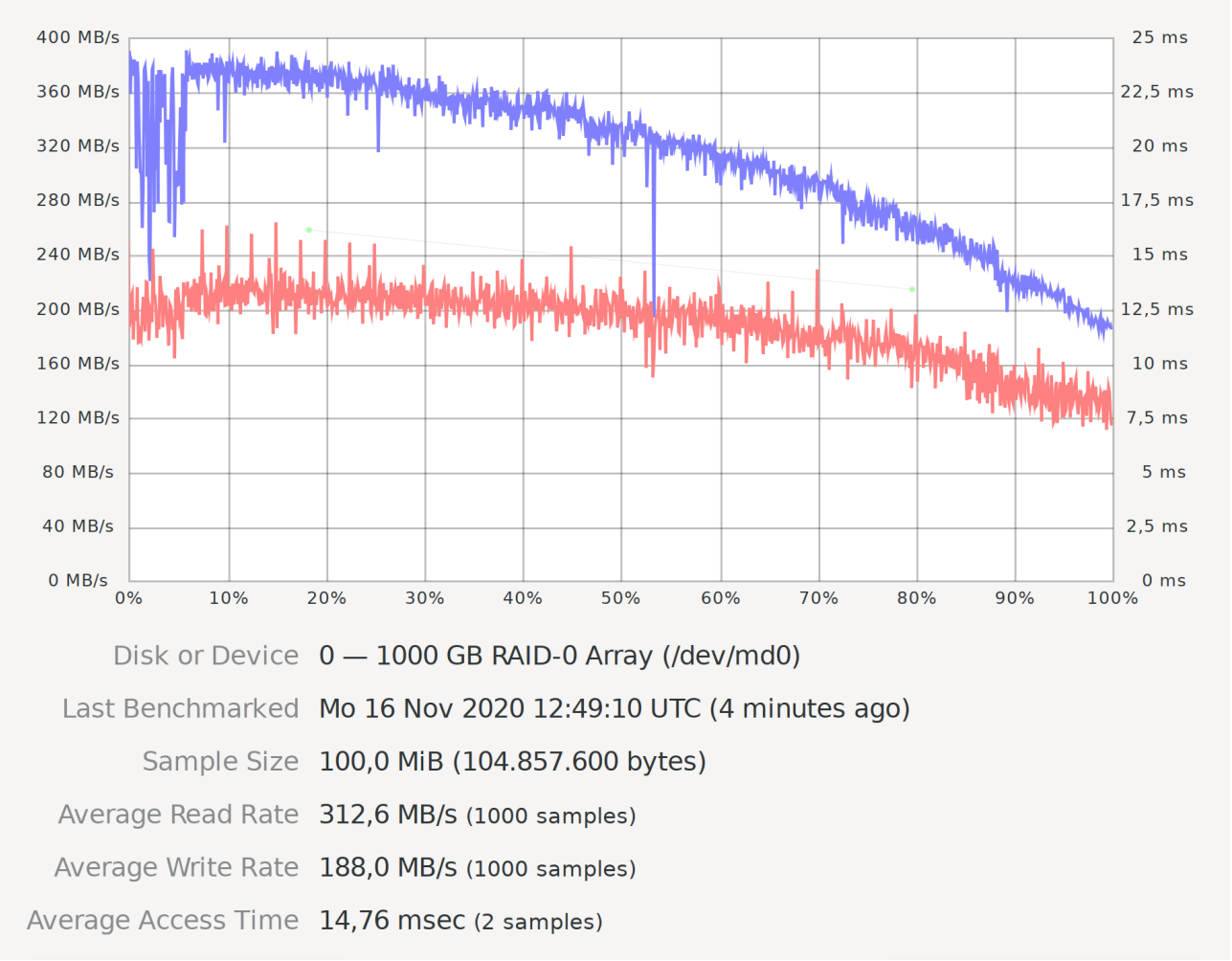


Figure 30: Benchmark of Seagate 500 GB RAID 0 with *GNOME Disk Utility*

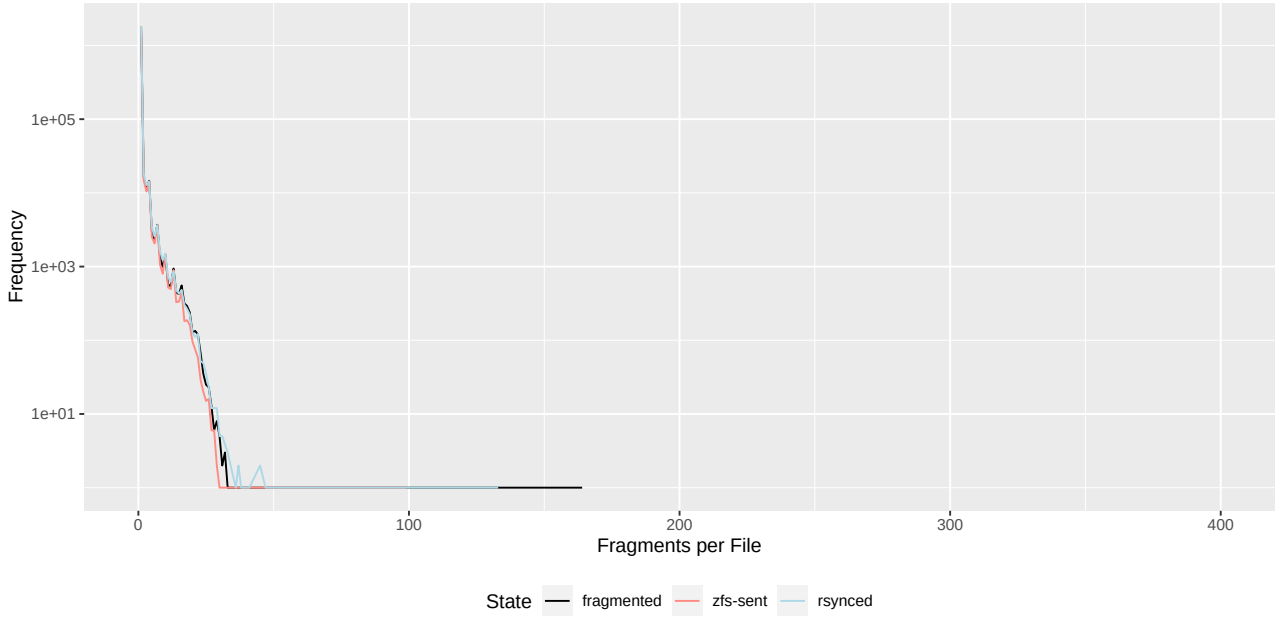


Figure 31: Seagate 500 GB (fragmentation of *ZFS*)

Drive	State	SUM == 1 Frags	SUM >5 Frags	SUM >10 Frags	MAX Frags
sg500	fragmented	1836785	14704	4901	1468
sg500	rsynced	1832854	15496	4969	1337
sg500	zfs-sent	1841568	12930	3878	1087
sg1000	fragmented	3661737	45501	23761	255
sg1000	rsynced	3684127	30550	10483	1309
sg1000	zfs-sent	3697451	27629	8922	1332
wdgold	fragmented	4109147	30866	9884	37
wdgold	rsynced	4102477	32507	11005	1349
wdgold	zfs-sent	4115308	28986	8671	36
sg500-raid	fragmented	3652605	47578	28650	239
sg500-raid	rsynced	4022348	61449	28651	2390
sg500-raid	zfs-sent	4069400	43309	15565	2658

Table 2: Statistics about *ZFS* fragmentation

5.2.6 ZFS: Fragmentation

The fragmentation of the *ZFS* filesystem seems to be almost unchanged at first sight. The runs with the *Seagate 500 GB* hard disk (shown in figure 31 on page 73) show hardly any differences. All three lines start with one fragment in over 185,000 files and drop exponentially (linearly in the illustration) to about 35 fragments per file.

The main difference between the runs is that in the «fragmented» run, files with up to 160 fragments exist. The «rsynced» run has an upper limit of 135, and «zfs-sent» closes at almost 50. Remarkable for the run between 0 and 30 are the small bumps, which appear quite regularly. With the exponential decrease, the frequency increases a little bit every 4-6 units on the x-axis. It is not known why this occurs.

The *Western Digital Gold* measurement curve gives a quite similar picture (figure 39 on page 98). Again, the difference between the three runs in the section 0 to 35 is barely present. The runs start at about 410,000 files with one fragment and decrease exponentially to a file with 35 fragments. The bumps just described also occur here at the same interval.

In contrast to the *Seagate 500 GB*, both the «fragmented» and the «zfs-sent» run do not produce more than 40 files. The «rsynced» line shows an erratic behavior between 40 and 90. The number of files varies between 1 and 8 files in the x-axis run. In the further course, files with higher fragmentation occur sporadically. The «rsynced» line closes at about 165 fragments.

The results of the *Seagate 1000 GB* hard disk (figure 38 on page 97) differ slightly. As with the measurement results mentioned, the «zfs-sent» and «rsynced» lines between 0 and 35 are very similar to the previous results. However, the frequency of «rsynced» falls more slowly to 35 and oscillates from 0 to 5 files at 35 to 120 fragments. The highest number of fragments is about 125 for «zfs-sent» and about 180 for «rsynced».

The «fragmented» line is significantly different from 16 fragments per file and jumps abruptly from 2900 files to a frequency of 230 files. Between 16 and 63, the frequency decreases linearly up to a peak of 64 fragments per file with rounded up 380 measuring points. In the further course, the frequency varies between 0 to 8 files up to a fragment number of 255. An anomaly in this range occurs between 127 and 129 fragments and has its maximum at 62 fragments.

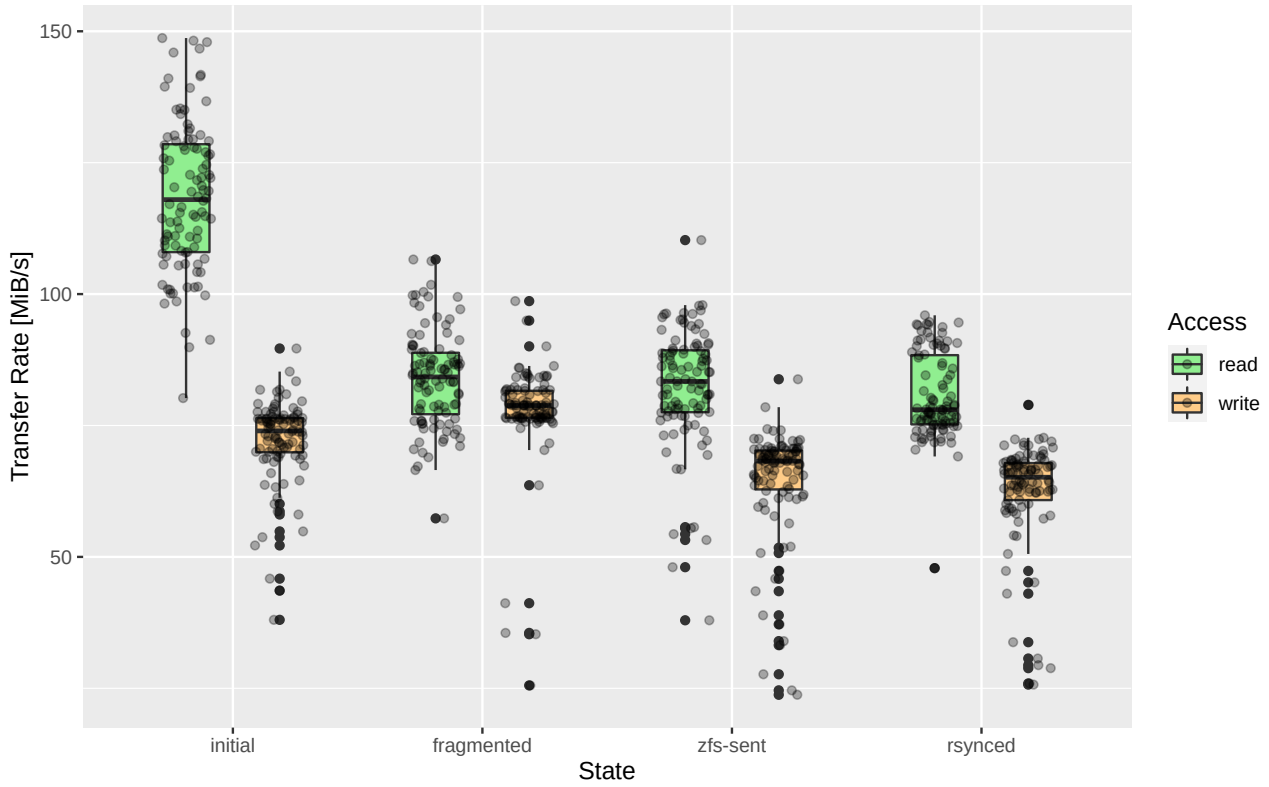


Figure 32: Seagate 500 GB (10 MB file size on *ZFS*)

In summary, however, it can be said for all hard disks that fragmentation in the *ZFS* filesystem occurs directly in advance and thus hardly changes in the further course of use.

5.2.7 ZFS: 10 MB File Size

The reading speed in the measurement series with a sample size of 10 MB is the highest with an empty, unused *ZFS* pool compared to the described condition.

The *Seagate 500 GB* hard disk (figure 32 on page 75) has an average reading speed of 118 MiBps, with the measurement points most likely normally distributed according to the *Shapiro-Wilk test* (table 9 on page 114). As the test progresses, the results are significantly worse, as was also assumed at the beginning. During reading for the three runs, the average data rate is in the range of 80 to 84 MiBps with a moderate standard deviation of 8 to 11. Since a small sample size was used in these test series, the overhead probably outweighs the results.

The difference between the write rates is slightly noticeable in all four runs. On average, it is 62 and 78 MiBps. In the graph, strong downward outliers are visible in the runs, which leads to a not normally distributed data situation.

The hard disk *Western Digital Gold* (figure 53 on page 105) shows similar readings. Again, the initial reading process outweighs the other measurement series with an average of 138 MiBps. Due to the few but strong outliers downwards, there is no normal distribution. The initial writing speed is about 90 MiBps and slightly outweighs the other runs.

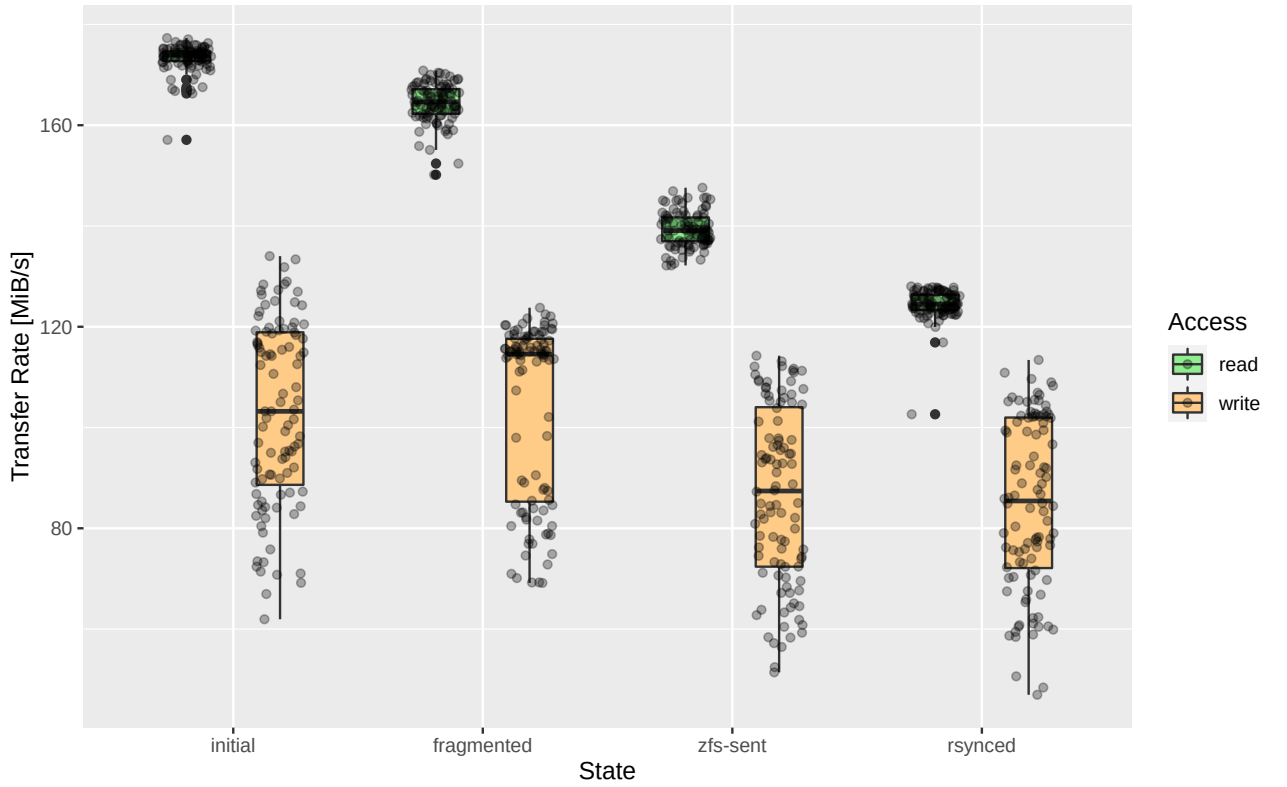


Figure 33: Seagate 500 GB (100 MB file size on *ZFS*)

In this case, the read and write speed is slightly better after defragmenting («zfs-sent»). The average data rate in the fragmented state is 70 MiBps (read) and 72 MiBps (write) versus 100 MiBps (read) and 82 MiBps (write) after the use of defragmentation measures. Only the program *rsync* does not seem to work well. The data rate is significantly lower than in the fragmented state (64 MiBps / 66 MiBps) with a relatively high spread of 18 MiBps when writing.

The performance curve of the *Seagate 1000 GB* hard disk figure 50 on page 103 runs like a sawtooth wave during both reading and writing. The initial performance with 123 MiBps reading and 72 MiBps writing forms the maximum of the graph. Once the hard disk has been artificially aged, the performance drops to a minimum of 53 MiBps read, and 43 MiBps write. The standard deviation is around 10 MiBps from this series of measurements; the initial state is much more scattered – especially the read access. This dispersion is most likely due to the small file size of 10 MB and the entire storage area availability.

It is also noticeable that the read rate is not very different from the write rate. This similarity shows that the sample is too small. The first defragmentation measure unfolds its effect and thus increases the performance. Also, the *rsync* measure does not do too badly. The reading performance is even minimally better than with «zfs-sent».

5.2.8 ZFS: 100 MB File Size

For the next measurement group with a sample size of 100 MB, the overhead should no longer be in the foreground, and the scatter should decrease.

The first hard disk *Seagate 500 GB* (figure 33 on page 76) shows an unusual behavior with its read rate. The data rate decreases by 15 to 25 MiBps with every subsequent series of measurements, but with a very small scatter of almost three MiBps. It is not possible to explain why this behavior occurs concerning the other measurement results.

The write rate also does not behave as expected. The average speed, both initially and after fragmentation, is almost 103 MiBps. According to the *Wilcoxon Signed-Rank Test*, there is a p-value of 0.94. Therefore, the difference between the two series of measurements is not resent. After each defragmentation measure, the rate drops to 85 MiBps. According to the statistical test, these two measurement series are not different ($p \approx 0.51$).

Furthermore, the results of the *Western Digital Gold* (figure 54 on page 105) are more in line with expectations. The read and write rates in the empty state are the highest of all other runs. On average, the benchmark reaches a read rate of 191 MiBps and a write rate of 141 MiBps. Remarkable is the small distance between the two quartiles.

The fragmented filesystem shows a significant reduction of I/O operations. Their average equals 192 MiBps (read) and 110 MiBps (write) and increases again in the next step, after the defragmentation measure. As with the hard disk just described, the tool `rsync` delivers worse results, again. These are even below the performance in the fragmented state.

The hard disk *Seagate 1000 GB* figure 51 on page 104 also has a sawtooth wave, and its behavior is in line with expectations. Due to the technical limitation (reading the same file), the read rate shows a minimal deviation. The speeds can be summarized in two different groups: the measurement series «initial» and «zfs-sent» with a high average value around 185 MiBps and the other two with a rate around 150 MiBps. It should be emphasized that the performance has not improved after using *rsync*.

The write rate is similar, but any measure does not exceed the initial write rate (124 MiBps). Also, no one writes slower outside the fragmented state (65 MiBps). The respective defragmentation measures deliver the familiar image: «zfs-sent» is more performant than «rsynced» (108 MiBps versus 83 MiBps).

5.2.9 ZFS: 1 GB File Size

Even with a higher sample size of one gigabyte, the pattern remains quite similar to the *Seagate 500 GB* (figure 34 on page 78) hard disk. The read rate between the initial and fragmented state decreases minimally (176 MiBps versus 174 MiBps) and decreases rapidly with defragmentation. As shown in the box plot, the reading performance is decimated after using *rsync*.

On the other hand, the write rates are somewhat lower after defragmentation, but there is no real trend here. It moves in a corridor from 71 MiBps to 91 MiBps. Assuming that the free space fragmentation, in this case, behaves similarly to the determined file fragmentation, the write rate can hardly change.

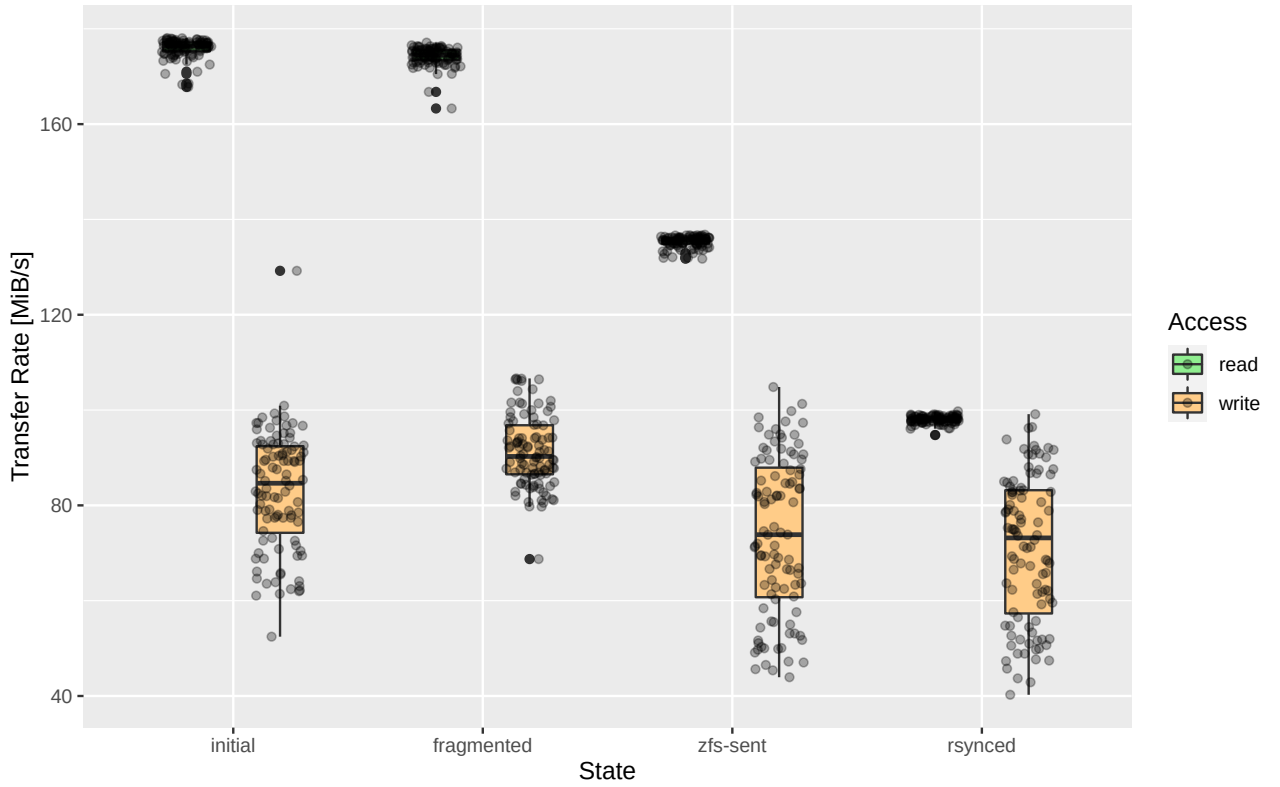


Figure 34: Seagate 500 GB (1 GB file size on *ZFS*)

The defragmentation measures seem to have improved performance for the *Western Digital Gold* (figure 55 on page 106) with large files. The read rate of the last two runs is 143 MiBps and 148 MiBps, respectively. The write rate also increased from 72 MiBps to 90 MiBps. Furthermore, the write difference between the two defragmenters is tiny with a p-value of about 0.88. Nevertheless, the defragmented filesystems do not reach the initial values (187 MiBps read, 121 MiBps write).

Additionally, there is an anomaly in the measurement result when reading from the fragmented filesystem. This transfer rate equals only 31 MiBps. In this case, the available free space seems to have been very cluttered. The innermost track's reading speed is much higher than this measured value so that this reason can be excluded as the primary cause.

The measurement series's progression with the *Seagate 1000 GB* hard disk figure 52 on page 104 looks very similar to the 100 MB sample size: once again, a zigzag line is present. The peak reading speed is again about 185 MiBps and includes both the initial and the defragmented measurement («zfs-sent»). In contrast, there is a significant drop in the fragmented reading speed – it is only 55 MiBps. In this case, the sample is probably distributed very unfavorably on the disk. It could be heavily fragmented, lying on the inner tracks, or even on several data disks. The «rsynced» state benefits from the «fragmented» state's poor performance and therefore performs better at 93 MiBps.

The write performance is significantly lower in all measurement series due to the enormous sample size. The files may have to be fragmented more often, which leads to a high cumulative *seeking time* of the device. The hard disk causes this disadvantage with its several data layers. The consequences can be seen in the first measurement series. The measurement points can be divided into clusters: the upper whisker and the section between the upper quartile and the lower whisker. Accordingly, the upper cluster is unlikely to have been distributed over several levels, which leads to a higher overall speed.

No more surprises appear in the further course of the game. The fragmented write rate of 50 MiBps is below grade and rises higher with the first defragmentation method «zfs-sent» (81 MiBps) than with *rsync* (69 MiBps).

5.2.10 ZFS: Fragmentation and Performance of RAID 0

The main difference between a *ZFS* RAID and the *EXT4* software RAID is an additional abstraction layer. For the *ZFS* filesystem, the topology of the existing devices is visible, and it decides itself how the distribution of the data is to proceed. This fact makes it possible to distribute the data well in terms of performance.

The fragmentation of the software RAID (figure 40 on page 98) is similar to the *Seagate 1000 GB* (figure 38 on page 97) in its pattern. The two runs, «zfs-sent» and «rsynced», are both quite similar. They start at over 400,000 files consisting of one fragment and decrease to about 50,000 with two or three fragments. The two processes continue to decrease rapidly and end at about five files with 50 fragments each. Thus they show a slightly higher fragmentation than the measurement series without the use of a RAID. Between frequencies 50 and 130, the «rsynced» line oscillates between 0 and 8 files.

The course of the «fragmented» measurement data is a bit strange, as with the *Seagate 1000*. The frequency between 1 and 16 fragments per file is similar to the other two measurement lines. However, the frequency decreases rapidly from 17 to 100 occurrences, increases up to 375 (40 fragments), and decreases again to 19 (63 fragments).

Anomalies follow with 64 fragments (95 files), 108 to 114 (up to 9 files), and 126 to 129 fragments (up to 36 files). As with the other hard disk, these occur preferably in the range of the powers of two.

The fact that the two hard disks behave similarly can be explained by the fact that the number of data slices should be identical. The *Seagate 1000 GB* has two disks, and the software *RAID 0* was realized with two hard disks with one disk each.

The performance of the storage system shows no new conclusions compared to the previous *ZFS* measurements. All write rates during the 10 MB run (figure 56 on page 106) are on average 100 to 110 MiBps. Due to the lack of variation, they show that the overhead is too high in this mode.

At a higher sample size with 100 MB and 1 GB, it is visible that the initial and by «zfs-sent» defragmented measurement series reach a similar average performance. It equals 180 MiBps for the 100 MB run and 190 MiBps for the 1 GB run. The other two states («fragmented» and «rsynced») were significantly slower, as can be seen from table 11 on page 116 and table 13 on page 118.

This operating mode of *ZFS* also shows that defragmentation using *rsync* is not recommended. The write patterns are less suitable in all appearances than with the `zfs send | zfs recv` method.

The measurement results of the read accesses are not meaningful enough to derive a pattern from them. The initial speeds of the 100 MB and 1 GB run are within the common corridor with 266 and 286 MiBps, since these are two disks in a *RAID 0*. The fact that the measurements between the two sample sizes differ so much in the fragmented filesystem indicates a worst-case fragmentation state. Furthermore, the last two states show a contrary speed. These data cannot be used for the evaluation of a general ruleset.

5.3 Discussion

The file fragmentation analysis of the *EXT4* file system has shown that the majority of files (over 99%) have hardly been fragmented at all. The number of files with more than two fragments is more or less only a fraction of all files.

The problem is that the remaining files consist of many fragments. Furthermore, the tool *Geriatrix* also splits the free space. This section is essential as soon as new data has to be stored on the disk.

The performance measurements on *EXT4* were performed with `O_DIRECT`, so the measurement results are significantly lower concerning the actual read and write speed using typical applications. For each I/O access, all software caches are bypassed and waited for the disk result. Hardware caches' are probably not bypassed, but due to the high number of 4 KB requests, the real performance suffers. The advantage is that the differences between the states are better visible.

A sample size of 10 MB is demonstrably unsuitable for detecting differences. Either the administration overhead is too large, or the runtime variations are too small. Sample sizes of 100 MB and above show differences very clearly.

As suspected, the reading and writing speed of new files on a fragmented hard disk is significantly reduced. A parallel analysis of the free space would be interesting (future work) to find possible correlations.

Somewhat surprising is that the in-house defragmentation tool of *EXT4* makes the situation significantly worse. Yes, it has reduced the number of fragments, but unfortunately, not the number of highly fragmented files. Moreover, the performance of new entries, which occur in a production system, has become all the worse.

From this, it can be deduced that the tool has reduced file fragmentation at the expense of free space fragmentation. This decision might be useful for a read-only or archiving system, but not for a productive storage system.

The best countermeasure with *EXT4* is to copy the data to a new disk. Whether this has to be done through *rsync* remains to be seen. Probably other file-based copying processes work similarly well. In-place defragmentation is difficult due to the lack of unused space.

The measurements of the *EXT4* RAID 0 are unfortunately not meaningful enough to make a statement. Further series of measurements with other configurations are necessary because obviously, a block size of 4 KB with `O_DIRECT` does not work very well for this RAID configuration.

The filesystem *ZFS* uses an entirely different strategy. As can be seen from the fragmentation graphs, files are quite often fragmented. Concerning the *copy-on-write* procedure, this can be a reasonable approach: As soon as changes are made, the files have to split up.

The differences in file fragmentation between the different states are not significant, so an analysis of free space is appropriate. It might clarify some anomalies in performance measurement. At some points, it has become visible that countermeasures can be partially promising.

However, simply copying files, as in *EXT4*, causes a more significant performance problem than the fragmented state. While using file-based programs like *rsync*, the files are written individually to the target system. Each file transfer is seen as a single transaction, which leads to an administration overhead. This performance outcome indicates a strong fragmentation of free space and shows the unsuitability of this method.

Interestingly, the in-house copy process using `zfs send | zfs recv` works much better. In this process, the files are sent as a continuous data stream. The complete transfer process of the snapshot is probably considered one transaction, and the extents can be written directly to the disk. If in doubt, this will not reduce the number of extents, but it will defragment the free space.

In summary, it can be said that an aging process in the form of fragmentation exists in both the files and the free space and can drive to performance losses. It always depends on the actual applications, configurations, usage time, and intensity. Unfortunately, the in-house defragmenting tools are unsuitable; only copying will deliver an improvement.

6 Conclusion

6.1 Summary

This master thesis deals with the phenomenon of aging in file systems and takes a closer look at the file systems *EXT4* and *ZFS*. To investigate this problem with scientific methods, knowledge about the functioning of a file system is essential. Accordingly, the first step is to define what a file system is and how the address schemes *Cylinder Head Sector (CHS)* and *Logical Block Address (LBA)* work. To understand how these addresses are stored in blocks in the metadata area, *inodes* and *extents* are explained using the filesystem *EXT4*. Since *ZFS* is a *copy-on-write* system, the basic functionality and special features are described.

In the next section, storage systems, such as *Redundant Array of Independent Disks (RAID)* with its different operating modes are explained, and their characteristics are shown. The *Distributed Filesystem* contributes to the completeness.

How modern storage media, more precisely hard disks and SSDs, operate is also part of this thesis. It is essential which different technologies are used to increase the data density on the magnetic media. These can not only have positive effects on performance (see *Shingled Magnetic Recording (SMR)*).

Furthermore, to demonstrate fragmentation, a simulation of how files and their extents change over time is shown. This fragmentation can lead to massive performance losses on hard disks due to multiple repositioning processes to read the right tracks. Accordingly, the further simulation shows how defragmentation theoretically works in-place of a filesystem.

The proof of different fragmentation and the possibly associated performance loss is significant for the experiment. For this purpose, the two named filesystems on different hard disks are heavily loaded with the *Geriatrix* tool, and thus an aging process is performed. As countermeasures, existing defragmentation tools, as well as various copy mechanisms, are tested. The fragmentation of the files is determined in each step, and the performance is measured.

6.2 Contributions

The results of the measurements show that there are signs of aging. The performance of the respective filesystems suffers greatly under long-term stress, especially at high fill levels. A significant reason for this is the fragmentation of the complete storage system, which severely restricts further use in the long run.

The determination of the file fragments using the *fiemap* interface is efficient. With the newly developed tool *fraggy*, it is now possible to evaluate this data with all filesystems that offer this I/O interface. For this purpose, a patch for the filesystem *ZFS* exists to provide this functionality.

Artificial aging using *Geriatrix* had to be changed from allocating file storage to writing data. Otherwise, the file size under *ZFS* always equals to block size, even if compression has been disabled.

The use of the native defragmentation tool `e4defrag` will reduce file fragmentation. However, the program fissured the free storage area so that a further use with the aspect of storing new data is inefficient. It is better to avoid this tool unless it is used to defragment a read-only archive system.

An excellent way to improve filesystem's general state is to copy the data to a fresh, new file system. From an economic point of view, this practice is a sensible method, as well. Theoretically, the copying process only requires one additional storage medium or one additional node in a distributed system. After the defragmentation, the «older» storage system is available for the next copy process. After some time, the general condition improves by the increasing average performance. Depending on the system, the defragmentation could be designed as a rotational policy to ensure good performance in the long run.

The selection of the tool depends on the installed filesystem. For example, using a file-based program like *rsync* in the case of *EXT4* demonstrates some improvements. The performance after the copy process corresponds approximately to an empty state.

For instance, using file-based transactions in combination with *ZFS* is counterproductive. The performance is poor. By contrast, the command `zfs send | zfs recv` works much better. It transfers the data as a data stream as a single transaction to a *ZFS* pool. Besides, this process takes considerably less time.

6.3 Future Work

Furthermore, the frequency analysis of the fragments with *ZFS* showed that the states do not differ significantly among themselves. Therefore, a parallel analysis of the free space would be necessary, also for other filesystems. Reading existing files is improved by `e4defrag`, for example, but an analysis of the remaining space will probably prove its bad behavior. These two data sets can be correlated with the respective performance measurement afterward.

Moreover, measuring the read and write operation of existing data would be an outstanding addition to evaluating new files' speed. For this, either the used programs (*Geriatrix*) would have to aggregate this data, or additional I/O measurement tools would have to collect this data in parallel.

It is also interesting to note the corresponding file size and extent sizes for the respective measurement points when collecting the file fragmentation. With this additional information, an evaluation of the decision-maker of the used filesystem is possible. Depending on the intended use, this information facilitates the filesystem's choice.

The read and write speed of hard disks is strongly dependent on the *seeking time* and track position. Block tracking the samples or the entire hard disk can explain strange measurement phenomena and evaluate the quality of filesystem algorithms. Parallel to this, tracking the I/O load concerning the ratio of operation and waiting time across the different states would be interesting.

In this thesis, several parameters of filesystems and benchmark tools were not changed. In a large-scale measurement study the following, further variations would be conceivable:

- Fill levels of the filesystems
- Block/chunk size of the file and RAID systems
- RAID operating modes (RAID 0, 1, 5, 6, 10)
- Number of storage volumes used and transfer size per I/O operation of the benchmark tool.

At least, the host system's configuration may not be forgotten. Does the determined performance change by CPU pinning or by enforcing the CPU frequency? How considerable is the influence of the single-core performance on the benchmark? The evaluation of storage systems is difficult due to the variety and complexity of measurement methods, configurations, and application behavior and can only shed light on partial aspects.

Eidesstattliche Versicherung

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

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

Hamburg, den 27.11.2020

List of Figures

References

[1] Michael Kuhn; Julius Plehn; Yevhen Alforov. "Improving Energy Efficiency of Scientific Data Compression with Decision Trees". In: *ENERGY 2020: The Tenth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. Lisbon, Portugal: IARIA XPS Press, Sept. 2020. ISBN: 978-1-61208-788-7 (cit. on p. 9).

[2] Andrew S. Tanenbaum; Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015. ISBN: 978-0-133-59162-0 (cit. on p. 13).

[3] T13 Technical Committee. "Information Technology - AT Attachmentwith Packet Interface - 6(ATA/ATAPI-6)". In: (2002) (cit. on p. 15).

[4] The kernel development community. *EXT4 – Dynamic Structures – Index Nodes*. Sept. 15, 2019. URL: `https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html#index-nodes` (cit. on p. 19).

[5] The kernel development community. *Fiemap Ioctl*. Nov. 27, 2020. URL: `https://www.kernel.org/doc/html/latest/filesystems/fiemap.html` (cit. on p. 46).

[6] Intel Corporation. *CVE-2018-12126: Microarchitectural Store Buffer Data Sampling*. June 11, 2018. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12126` (cit. on p. 48).

[7] Intel Corporation. *CVE-2018-12127: Microarchitectural Load Port Data Sampling*. June 11, 2018. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12127` (cit. on p. 48).

[8] Intel Corporation. *CVE-2018-12130: Microarchitectural Fill Buffer Data Sampling*. June 11, 2018. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12130` (cit. on p. 48).

[9] Intel Corporation. *CVE-2018-12207: iTLB multihit*. June 11, 2018. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12207` (cit. on p. 48).

[10] Intel Corporation. *CVE-2018-3639: Speculative Store Bypass, Variant 4*. Dec. 28, 2017. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639` (cit. on p. 48).

[11] Intel Corporation. *CVE-2019-11091: Microarchitectural Data Sampling Uncacheable Memory*. Apr. 11, 2019. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11091` (cit. on p. 48).

[12] Intel Corporation. *CVE-2019-11091: TSX Asynchronous Abort*. Apr. 11, 2019. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11091` (cit. on p. 48).

[13] Microsoft Corporation. *CVE-2019-1125: SWAPGS*. Nov. 26, 2018. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1125` (cit. on p. 48).

[14] Oracle; Delphix; Saso Kiselkov; Intel Corporation. *openzfs/zfs.git*. `https://github.com/openzfs/zfs.git`. Version e64cc4954c7862db6a6b4dc978a091ebc3f870da. July 15, 2020 (cit. on p. 45).

[15] Western Digital. *WD Internal Client SMR HDDs*. Apr. 22, 2020. URL: `https://blog.westerndigital.com/wp-content/uploads/2020/04/2020_04_22_WD_SMR_SKUs_1Slide.pdf` (cit. on p. 48).

[16] Alex Conway; Ainesh Bakshi; Yizheng Jiao; William Jannen; Yang Zhan; Jun Yuan; Michael A. Bender; Rob Johnson; Bradley C. Kuszmaul; Donald E. Porter; Martin Farach-Colton. "File Systems Fated for Senescence? Nonsense, Says Science!" In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 45–58. ISBN: 978-1-931971-36-2. URL: `https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway` (cit. on pp. 59, 60).

[17] Alex Conway; Eric Knorr; Yizheng Jiao; Michael A. Bender; William Jannen; Rob Johnson; Donald Porter; Martin Farach-Colton. "Filesystem Aging: It's more Usage than Fullness". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/conway (cit. on pp. 59, 60).

[18] Saurabh Kadekodi; Vaishnavh Nagarajan; Gregory R. Ganger. "Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 691–704. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/kadekodi (cit. on pp. 41, 51, 60).

[19] Moritz Lipp; Michael Schwarz; Daniel Gruss; Thomas Prescher; Werner Haas; Anders Fogh; Jann Horn; Stefan Mangard; Paul Kocher; Daniel Genkin; Yuval Yarom; Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018 (cit. on p. 48).

[20] Troy D. Hanson. *troydhanson/uthash.git*. https://github.com/troydhanson/uthash.git. Version 8e67ced1d1c5bd8141c542a22630e6de78aa6b90. July 22, 2020 (cit. on p. 46).

[21] Tony Sammes; Brian Jenkinson. *Forensic Computing*. 2nd ed. Springer-Verlag London, 2007. ISBN: 978-1-84628-397-0 (cit. on p. 13).

[22] David A. Patterson; Garth Gibson; Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. SIGMOD '88. ACM, 1988, pp. 109–116. ISBN: 0-89791-268-3. DOI: 10.1145/50202.50214. URL: http://doi.acm.org/10.1145/50202.50214 (cit. on p. 25).

[23] Drew Eckhardt; Michael Haardt; Ian Jackson; Greg Banks; Michael Kerrisk. *open(2) — Linux manual page*. July 28, 2020. URL: https://man7.org/linux/man-pages/man2/open.2.html (cit. on p. 55).

[24] Glenn K. Lockwood. *Basics of I/O Benchmarking*. Oct. 24, 2020. URL: https://glennklockwood.blogspot.com/2016/07/basics-of-io-benchmarking.html (cit. on p. 53).

[25] Nitin Agrawal; William J. Bolosky; John R. Douceur; Jacob R. Lorch. "A Five-Year Study of File-System Metadata". In: *ACM Trans. Storage* 3.3 (2007), 9–es. ISSN: 1553-3077. DOI: 10.1145/1288783.1288788. URL: https://doi.org/10.1145/1288783.1288788 (cit. on p. 51).

[26] Peter M. Chen; Edward K. Lee; Garth A. Gibson; Randy H. Katz; David A. Patterson. "RAID: High-Performance, Reliable Secondary Storage". In: *ACM Comput. Surv.* 26.2 (1994), pp. 145–185. ISSN: 0360-0300. DOI: 10.1145/176979.176981. URL: https://doi.org/10.1145/176979.176981 (cit. on p. 26).

[27] Alex Reece. *OpenZFS Code Walk: Metaslabs and Space Maps*. Jan. 25, 2015. URL: https://www.delphix.com/blog/delphix-engineering/openzfs-code-walk-metaslabs-and-space-maps (cit. on p. 44).

[28] Ohad Rodeh. "IBM Research Report Defragmentation Mechanisms for Copy-on-Write Filesystems". In: 2010 (cit. on p. 61).

[29] Akira Fujita; Takashi Sato. *ext2/e2fsprogs.git*. https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git. Version 8fd92e9af006da392d667775b8bbe0db9e8639d2. July 15, 2020 (cit. on p. 41).

[30] Keith A. Smith; Margo I. Seltzer. "File System Aging—Increasing the Relevance of File System Benchmarks". In: *SIGMETRICS Perform. Eval. Rev.* 25.1 (June 1997), pp. 203–213. ISSN: 0163-5999. DOI: 10.1145/258623.258689. URL: https://doi.org/10.1145/258623.258689 (cit. on p. 59).

[31] Jo Van Bulck; Marina Minkin; Ofir Weisse; Daniel Genkin; Baris Kasikci; Frank Piessens; Mark Silberstein; Thomas F. Wenisch; Yuval Yarom; Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, Aug. 2018 (cit. on p. 48).

[32] Lars Thoms. “Linux-Dateisysteme – Seminar »Speicher- und Dateisysteme«”. Seminar. Universität Hamburg, 2016 (cit. on p. 19).

[33] Lars Thoms. *SpaceCafe/fraggy.git*. `https://github.com/SpaceCafe/fraggy.git`. Version 92d2b8ffd30cb21a087ae81dd6ae946aaadd96a0. Nov. 27, 2020 (cit. on p. 47).

[34] Lars Thoms. *SpaceCafe/geriatrix.git*. `https://github.com/SpaceCafe/geriatrix.git`. Version f1b30718da9c9255e56ec00fd3d772c5413844cd. Nov. 27, 2020 (cit. on p. 52).

[35] Lars Thoms. *SpaceCafe/zfs.git*. `https://github.com/SpaceCafe/zfs.git`. Version 71c9a17045-237b6f3e8cc0b6014960202f3290b7. Nov. 27, 2020 (cit. on p. 50).

[36] Ofir Weisse; Jo Van Bulck; Marina Minkin; Daniel Genkin; Baris Kasikci; Frank Piessens; Mark Silberstein; Raoul Strackx; Thomas F. Wenisch; Yuval Yarom. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical report* (2018) (cit. on p. 48).

[37] Paul Kocher; Jann Horn; Anders Fogh; Daniel Genkin; Daniel Gruss; Werner Haas; Mike Hamburg; Moritz Lipp; Stefan Mangard; Thomas Prescher; Michael Schwarz; Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019 (cit. on p. 48).

Attachment

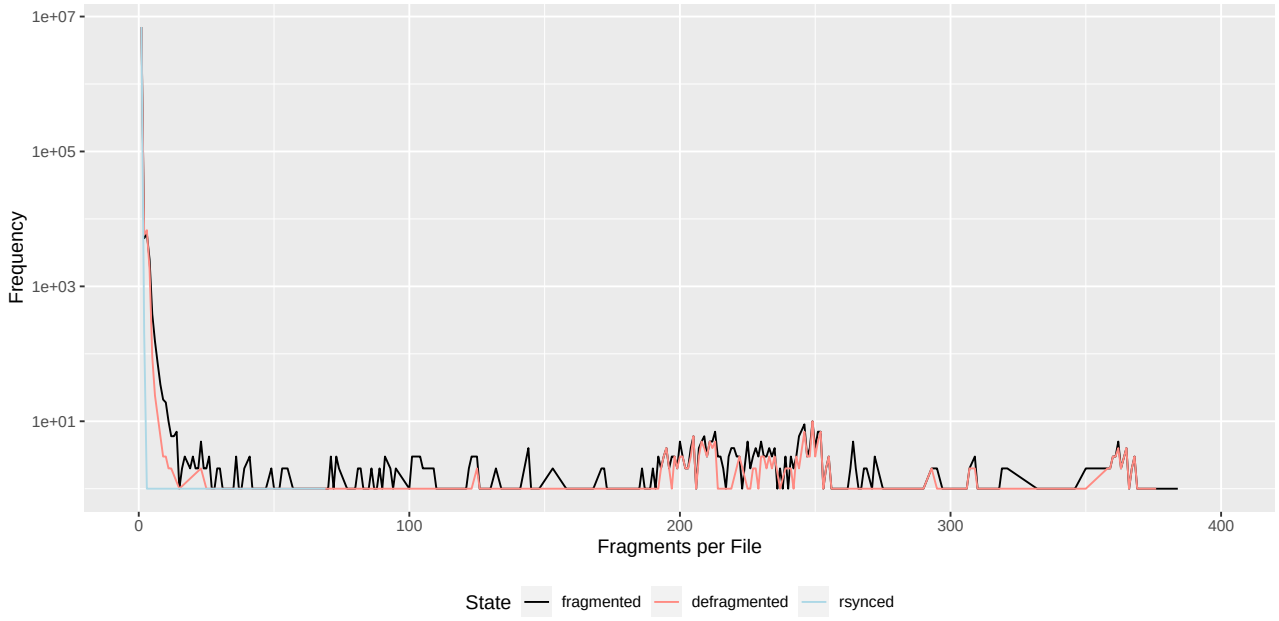


Figure 35: Seagate 1000 GB (fragmentation of *EXT4*)

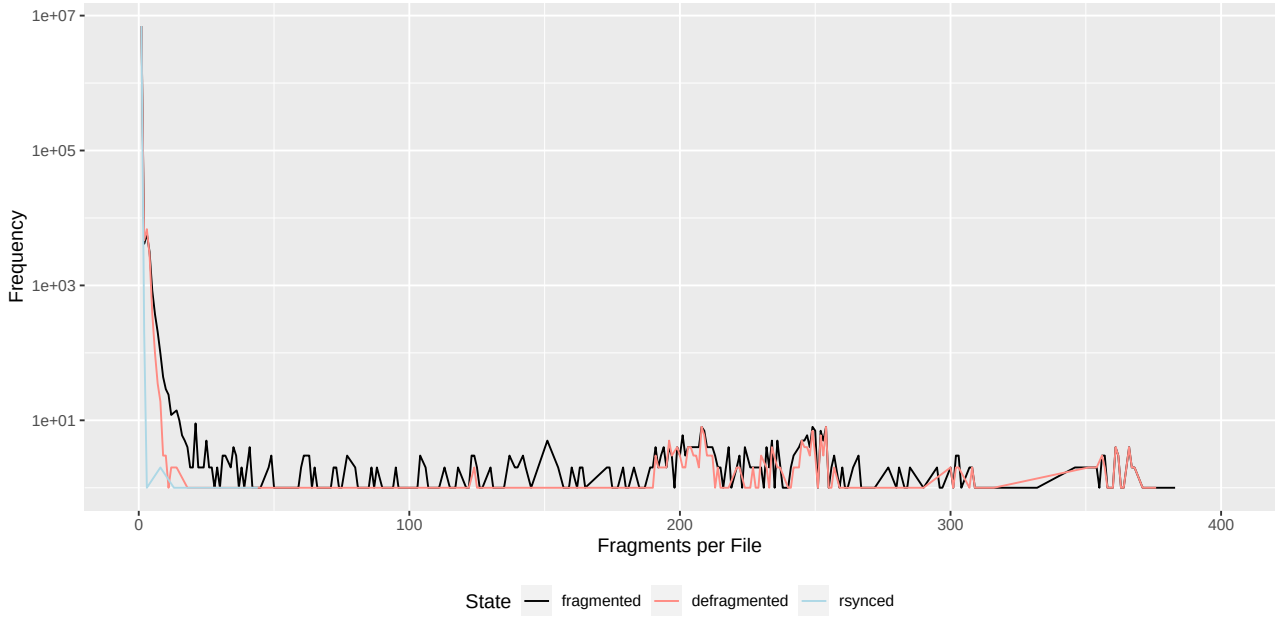


Figure 36: WD Gold 1000 GB (fragmentation of *EXT4*)

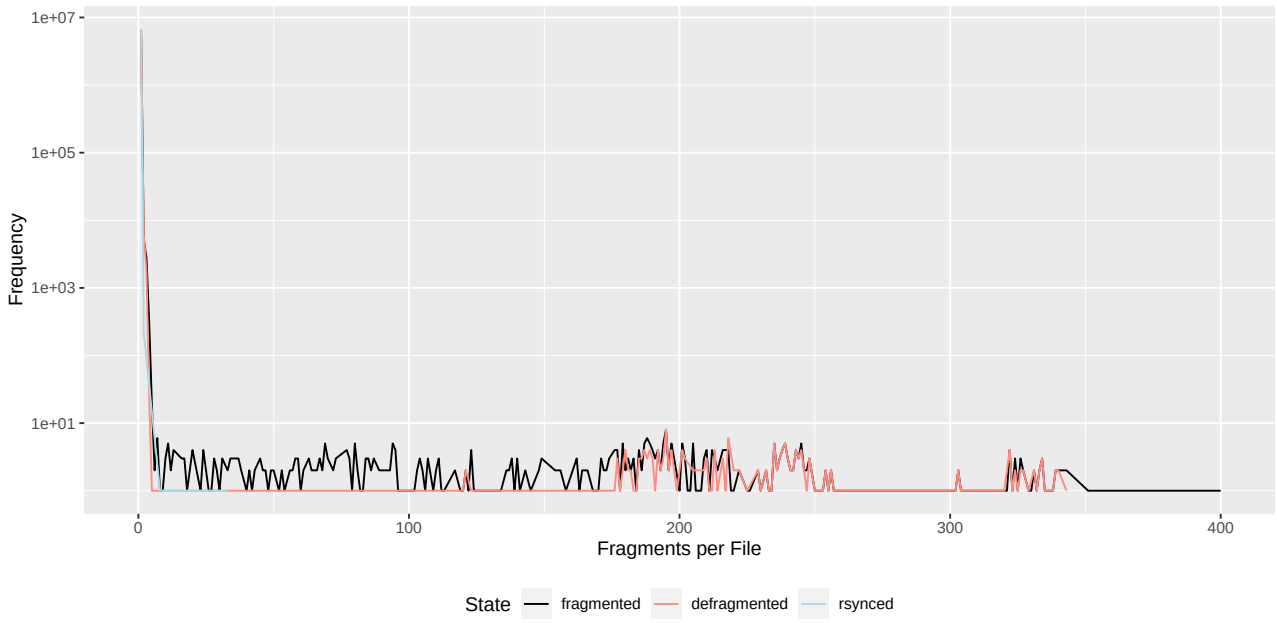


Figure 37: Seagate 500 GB RAID-0 (fragmentation of *EXT4*)

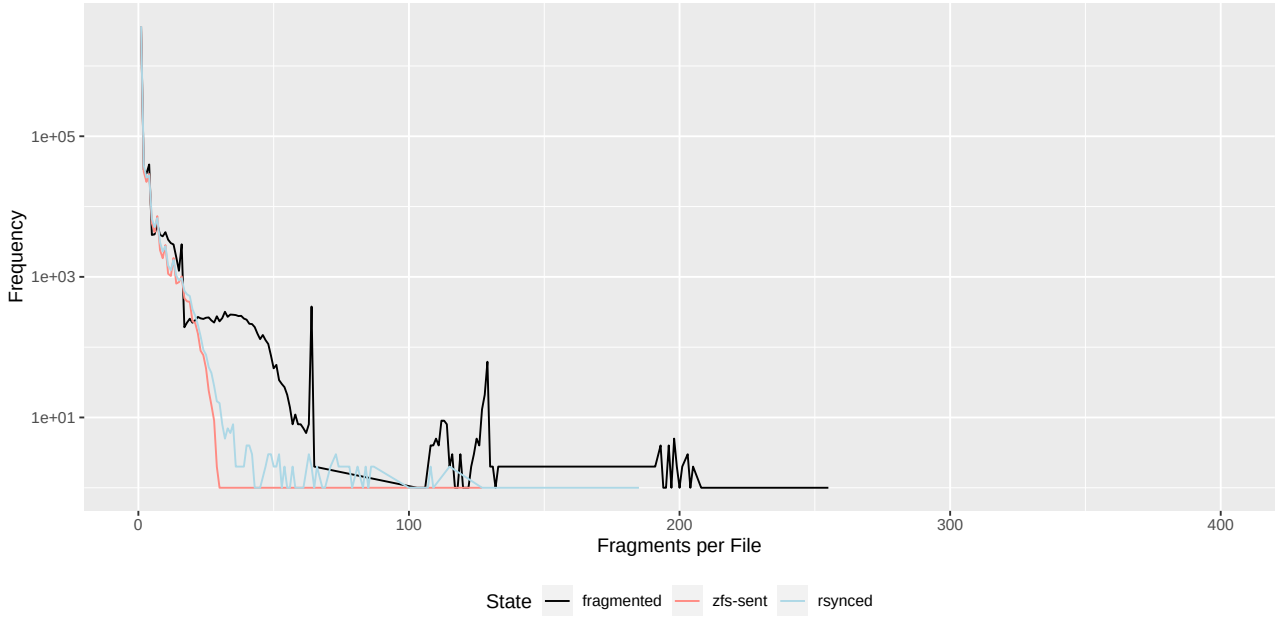


Figure 38: Seagate 1000 GB (fragmentation of *ZFS*)

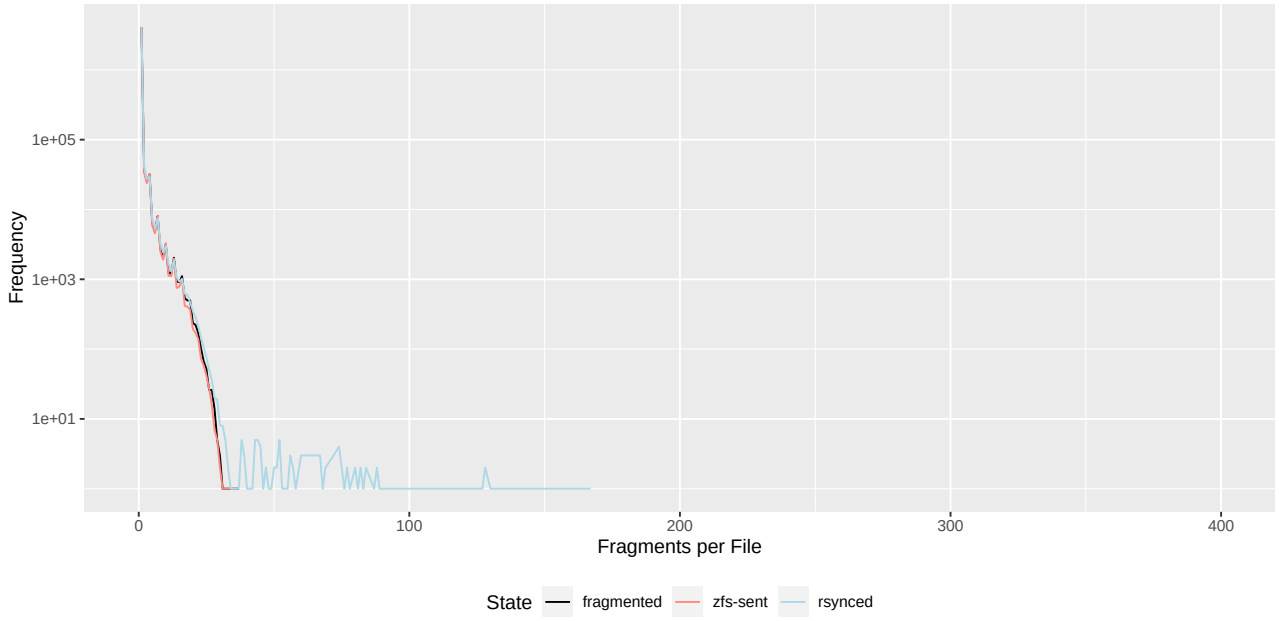


Figure 39: WD Gold 1000 GB (fragmentation of *ZFS*)

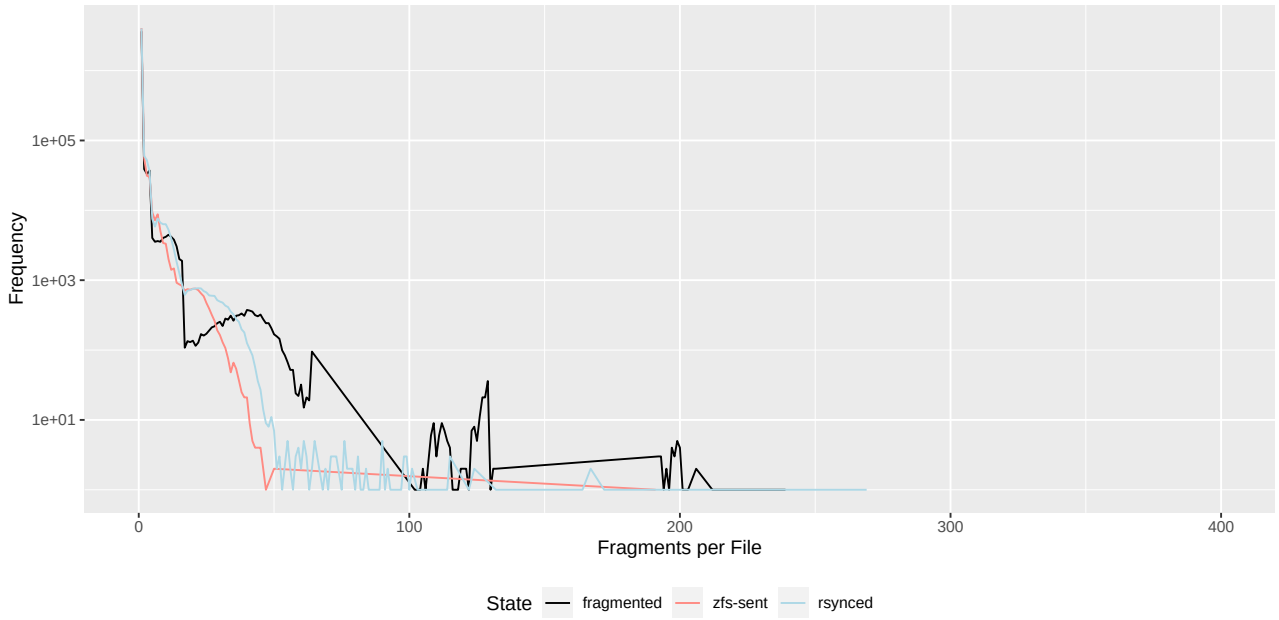


Figure 40: Seagate 500 GB RAID-0 (fragmentation of *ZFS*)

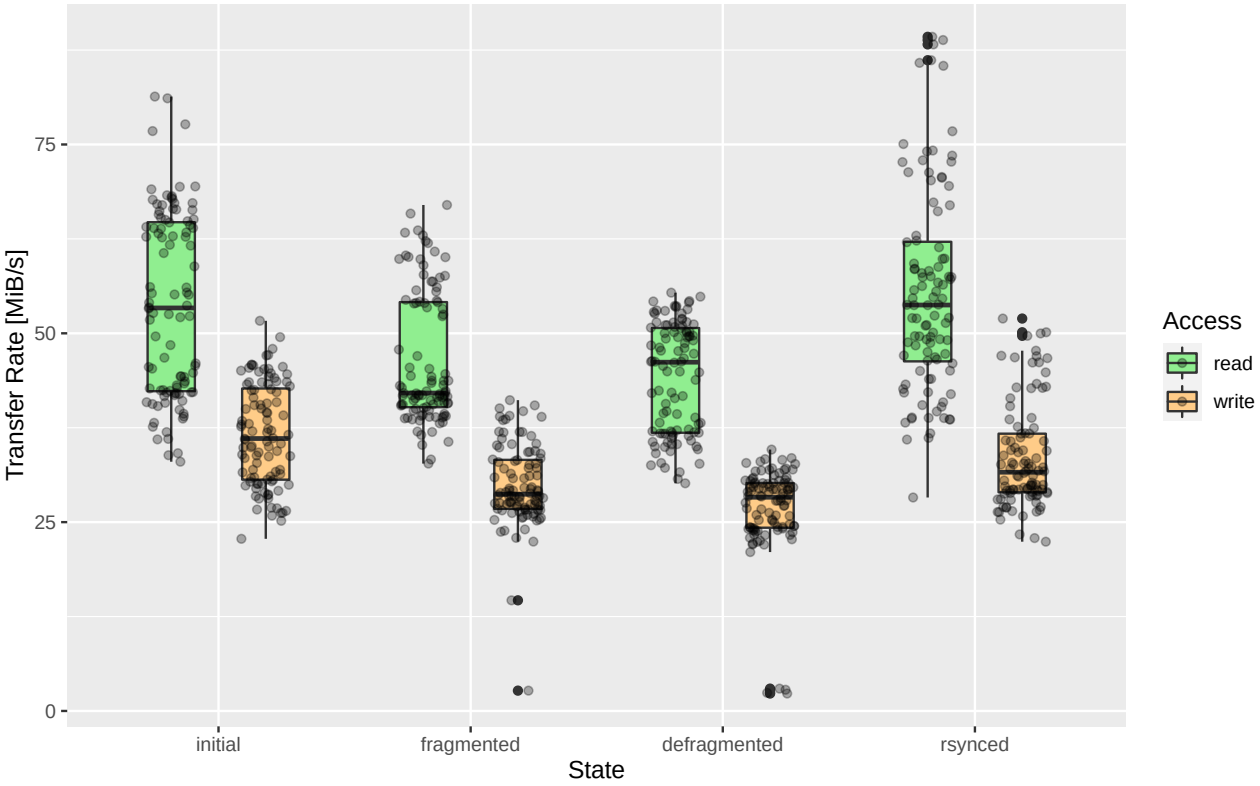


Figure 41: Seagate 1000 GB (10 MB file size on *EXT4*)

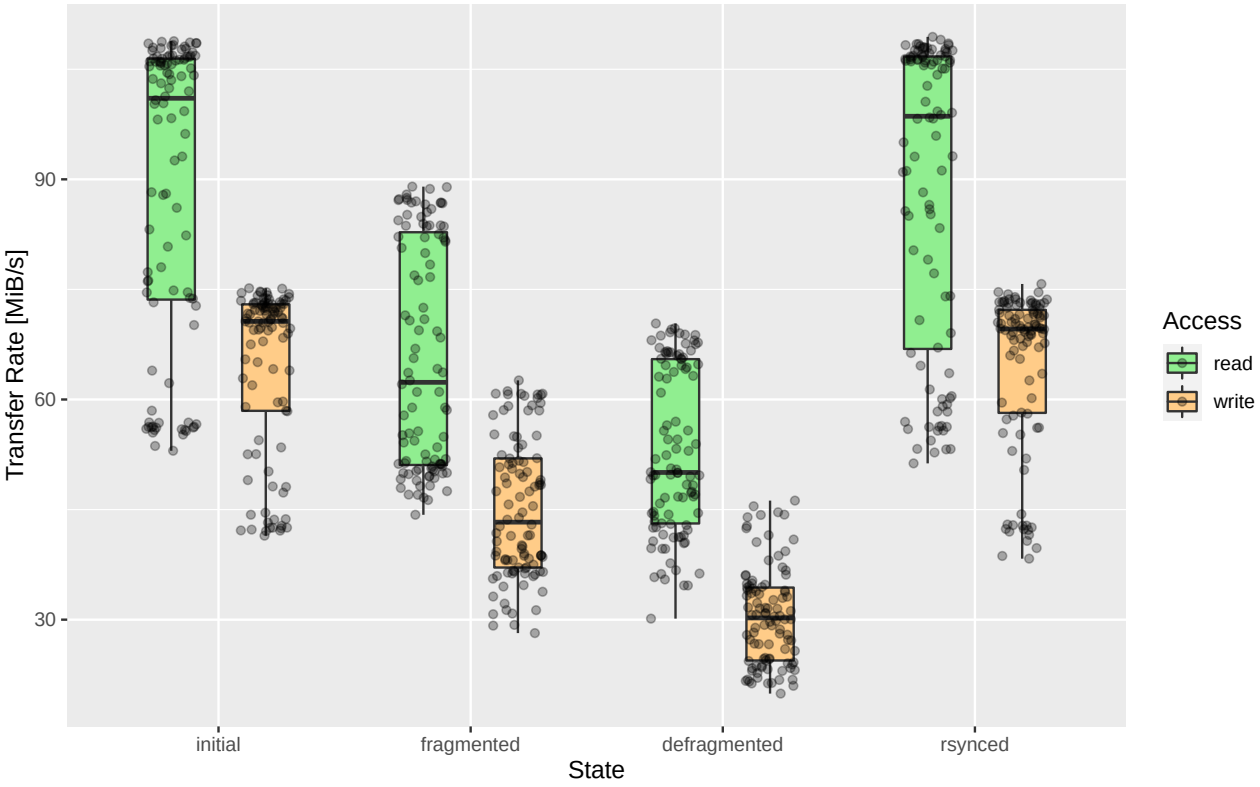


Figure 42: Seagate 1000 GB (100 MB file size on *EXT4*)

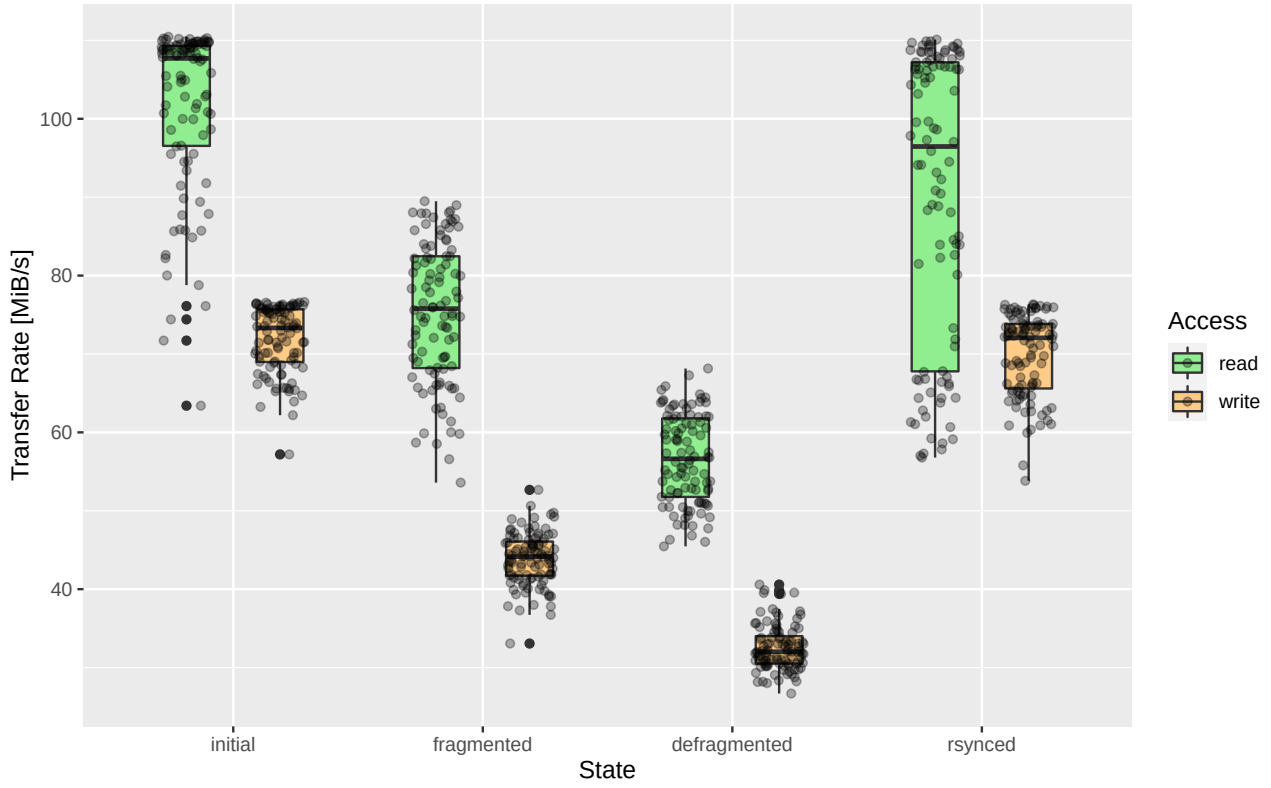


Figure 43: Seagate 1000 GB (1 GB file size on *EXT4*)

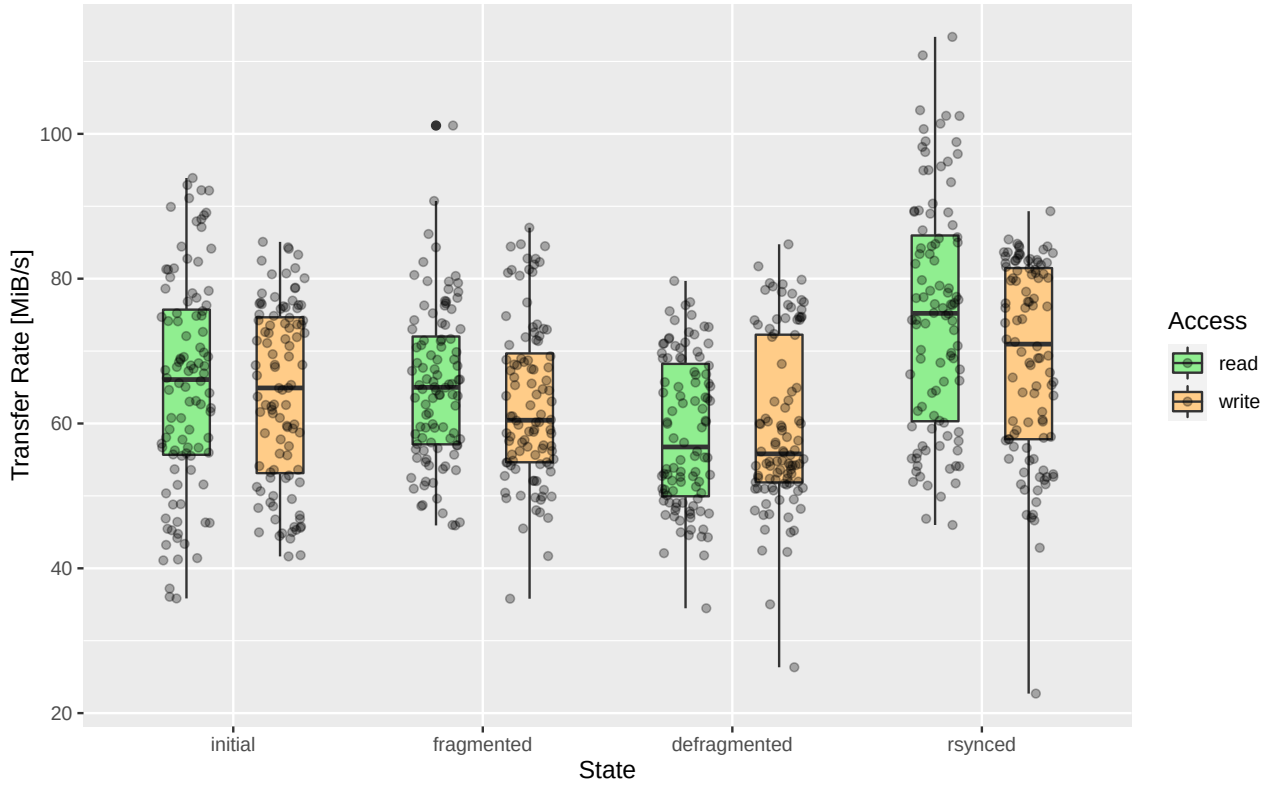


Figure 44: WD Gold 1000 GB (10 MB file size on *EXT4*)

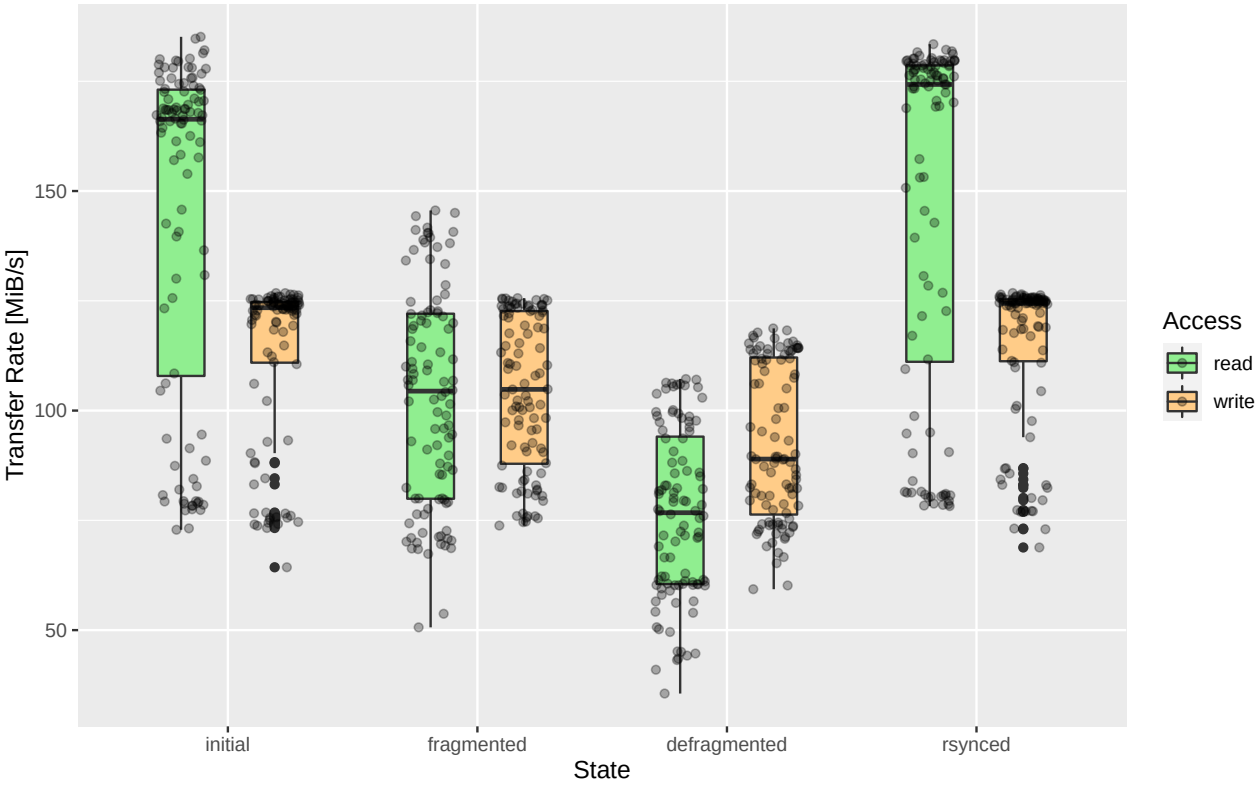


Figure 45: WD Gold 1000 GB (100 MB file size on *EXT4*)

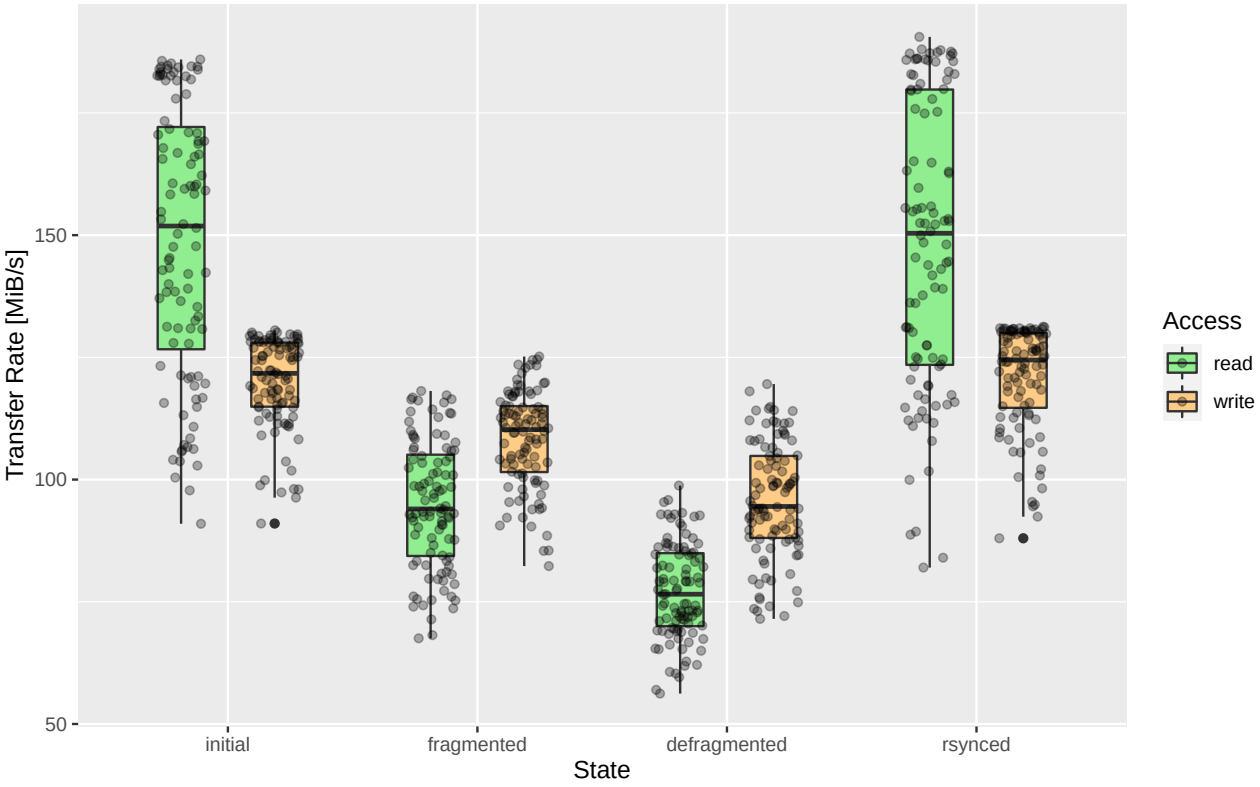


Figure 46: WD Gold 1000 GB (1 GB file size on *EXT4*)

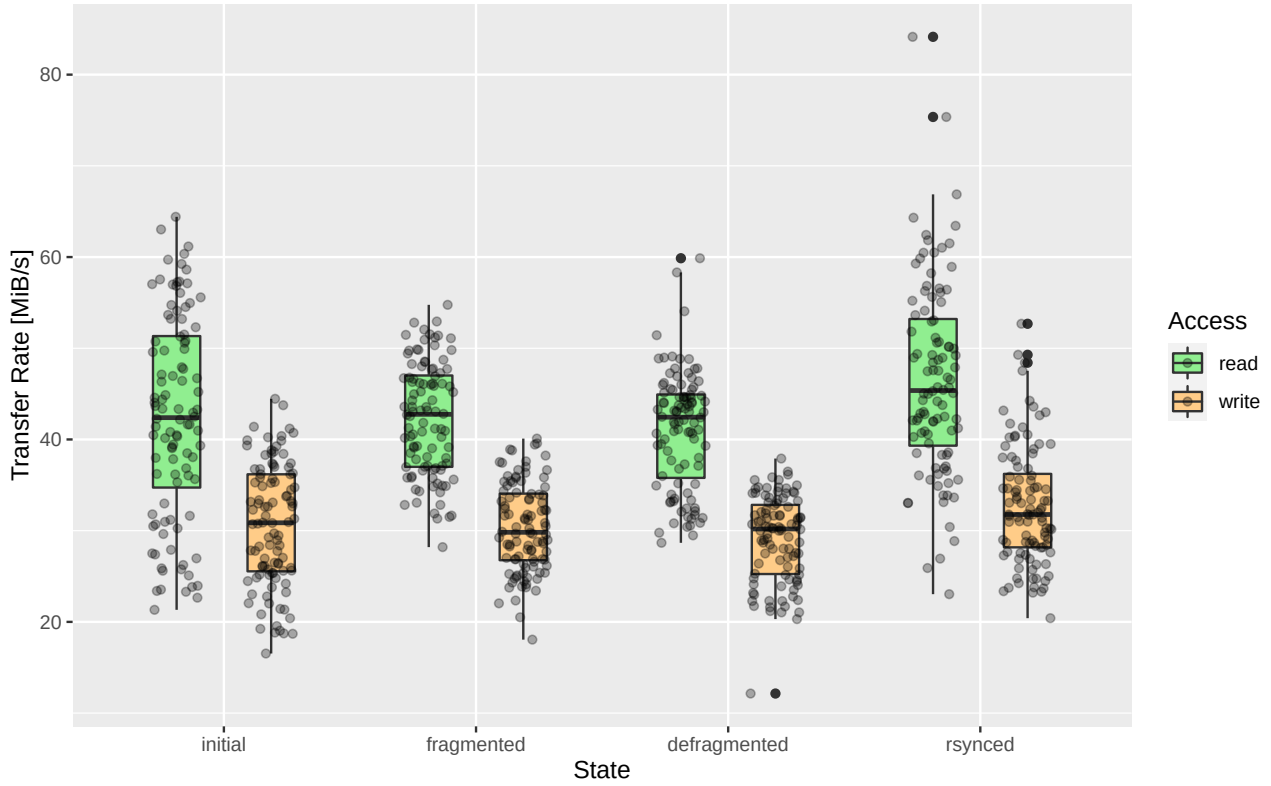


Figure 47: Seagate 500 GB RAID-0 (10 MB file size on *EXT4*)

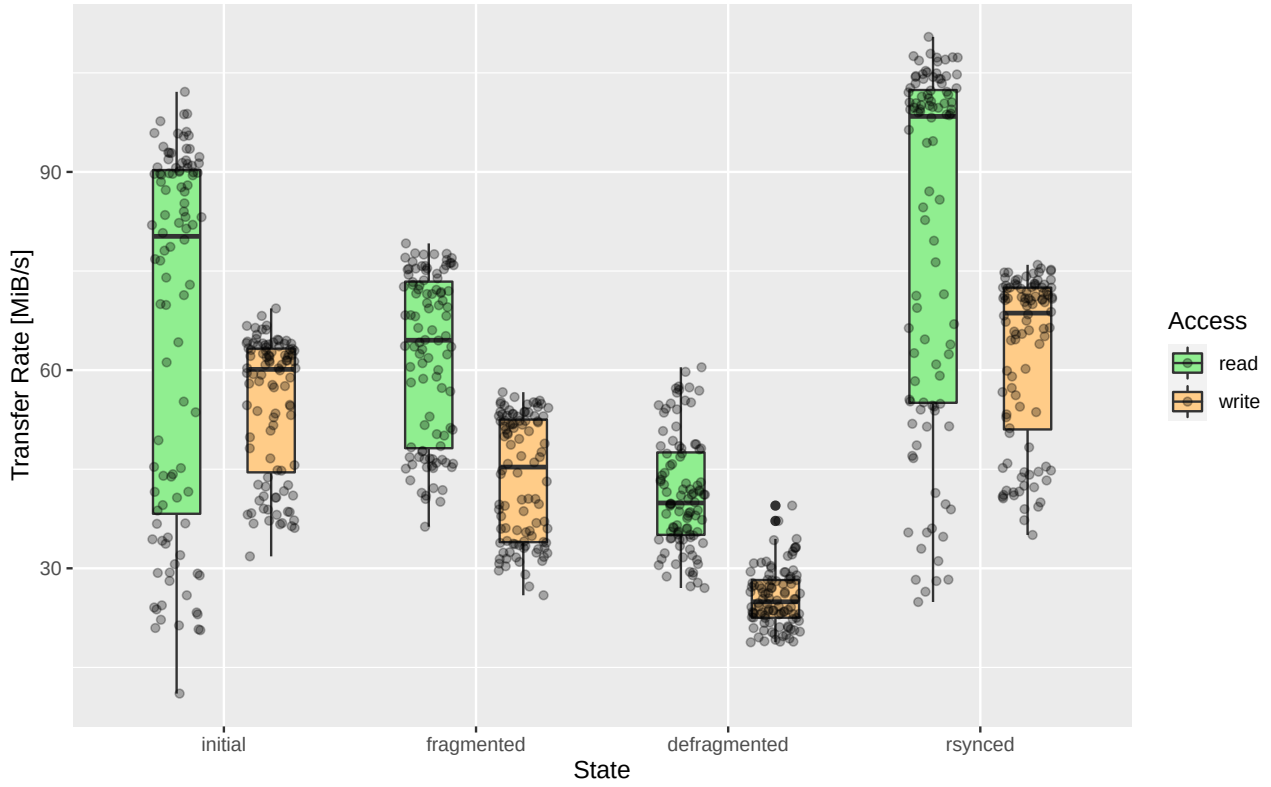


Figure 48: Seagate 500 GB RAID-0 (100 MB file size on *EXT4*)

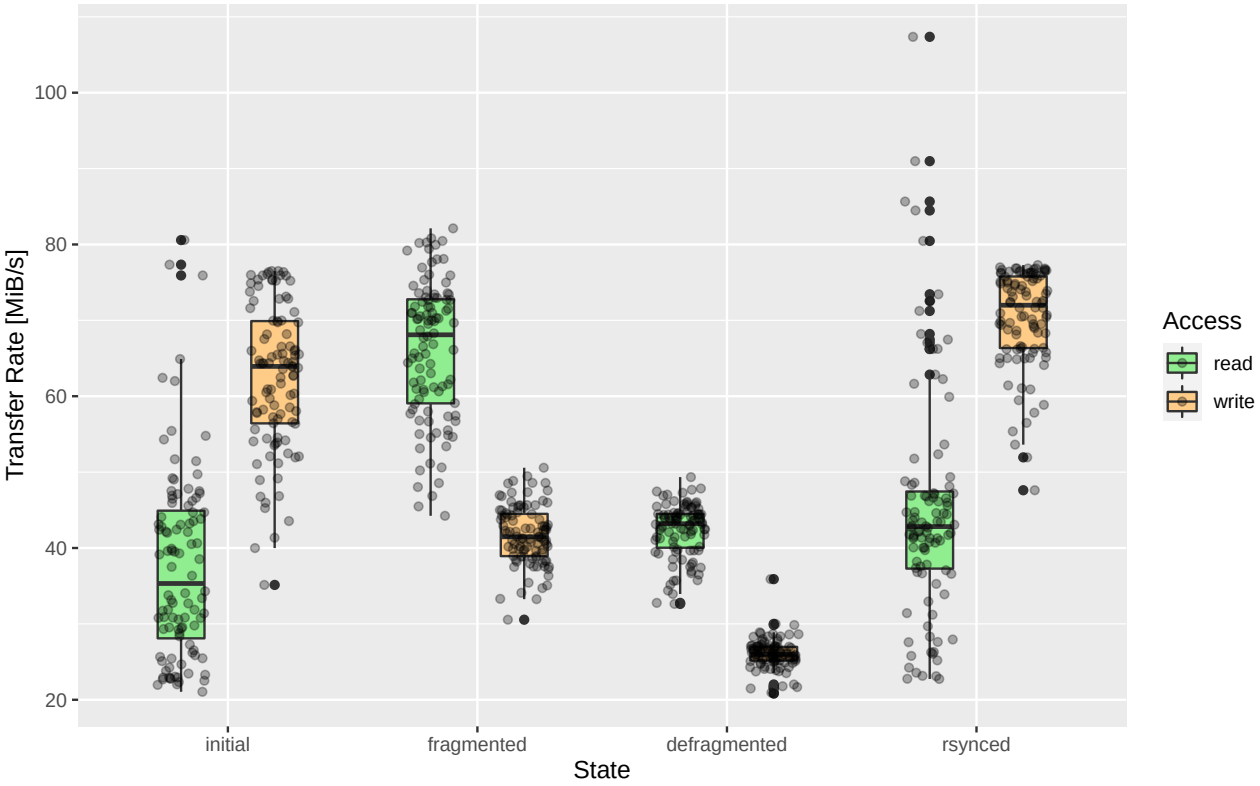


Figure 49: Seagate 500 GB RAID-0 (1 GB file size on *EXT4*)

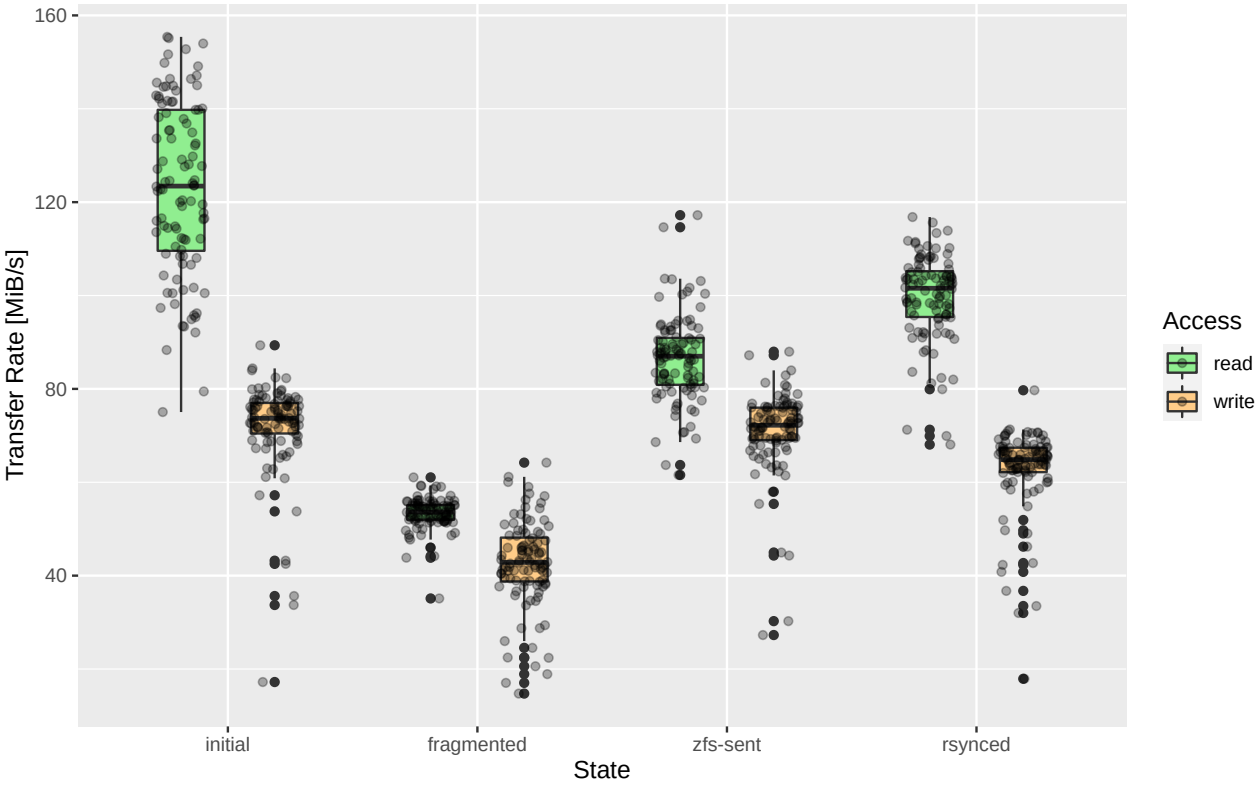


Figure 50: Seagate 1000 GB (10 MB file size on *ZFS*)

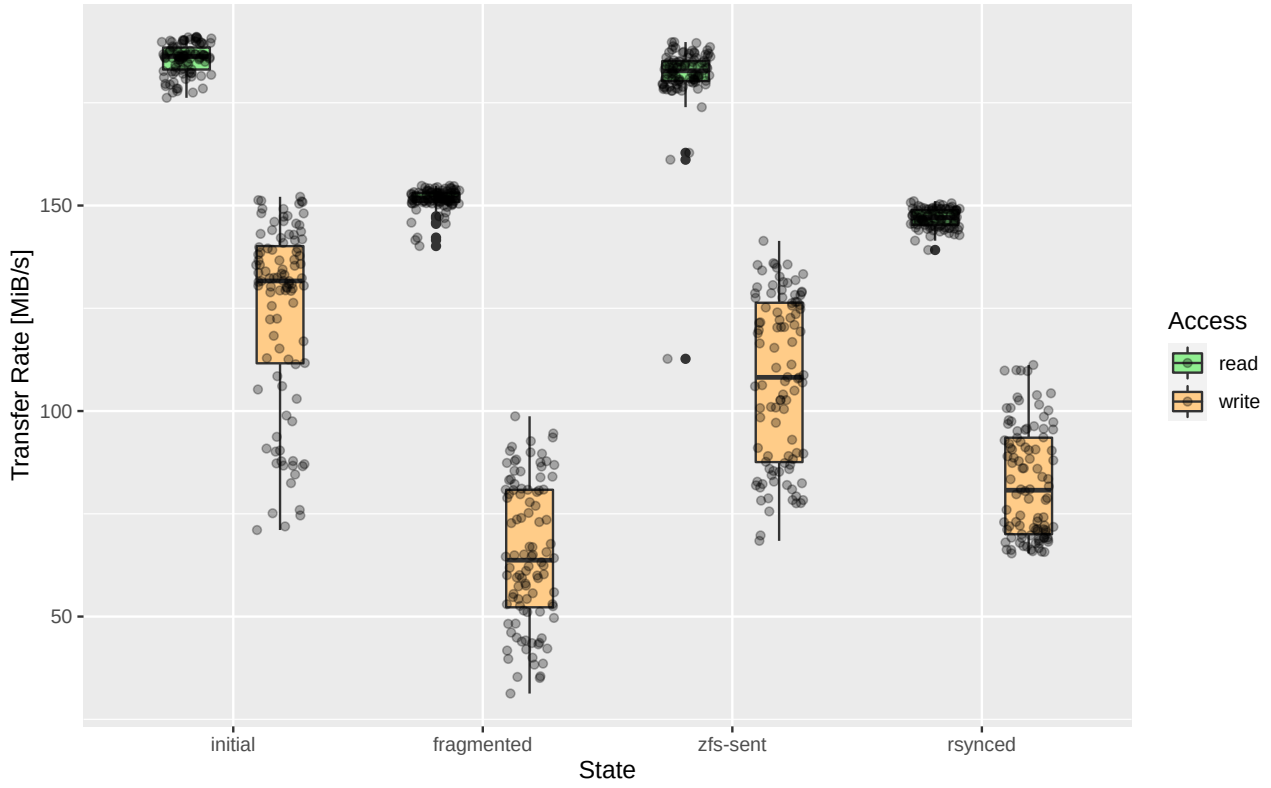


Figure 51: Seagate 1000 GB (100 MB file size on *ZFS*)

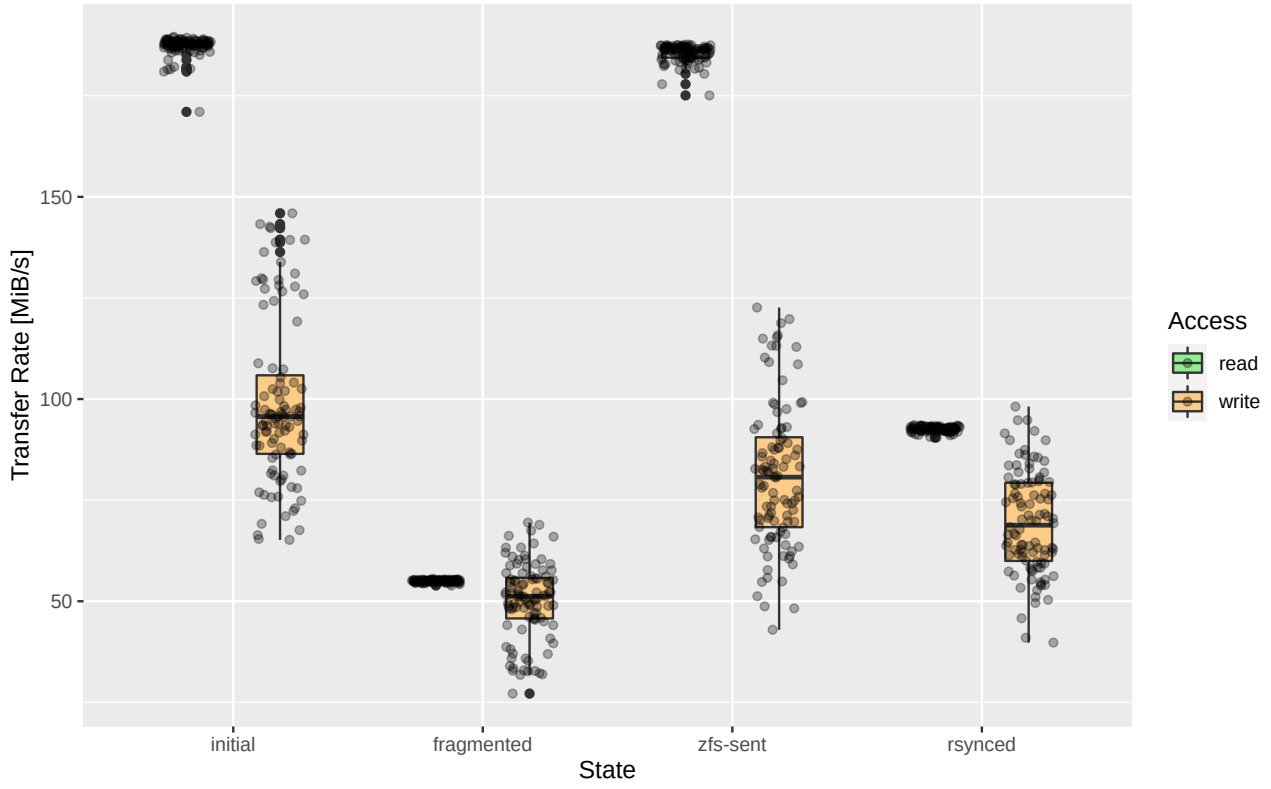


Figure 52: Seagate 1000 GB (1 GB file size on *ZFS*)

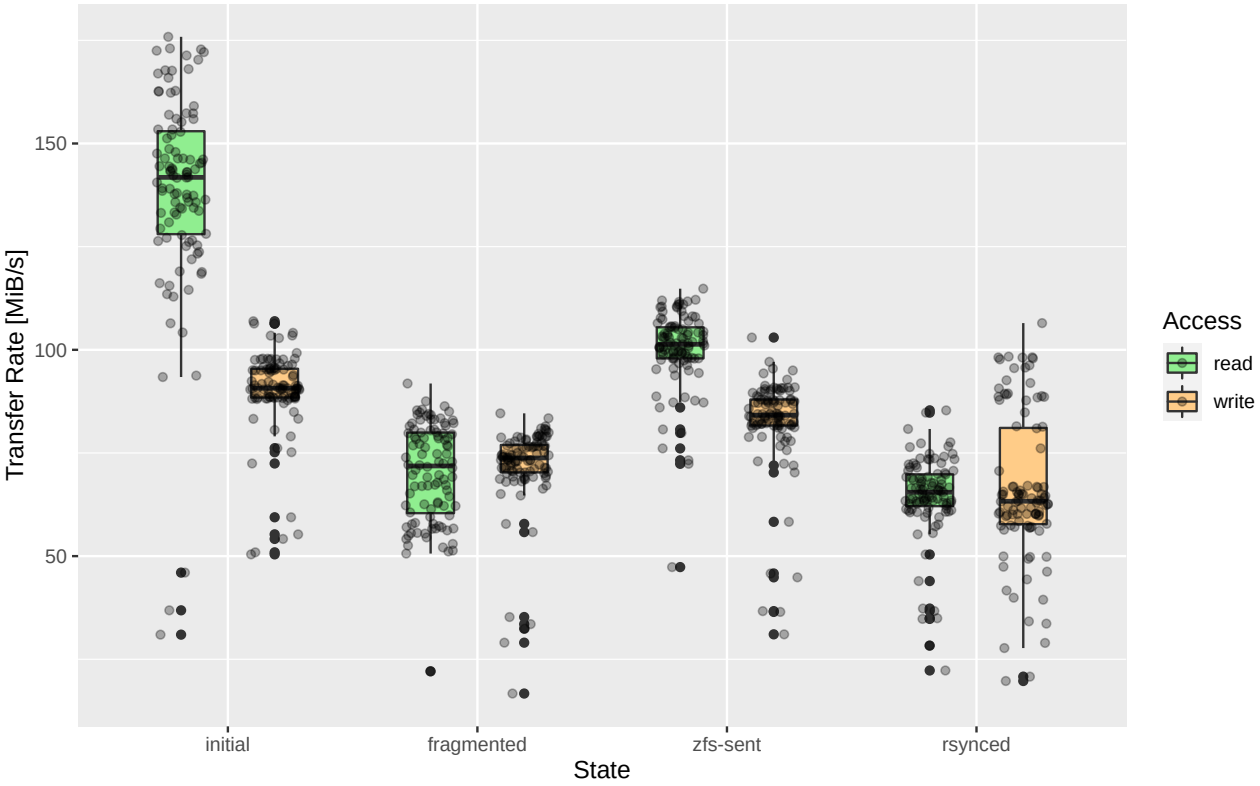


Figure 53: WD Gold 1000 GB (10 MB file size on *ZFS*)

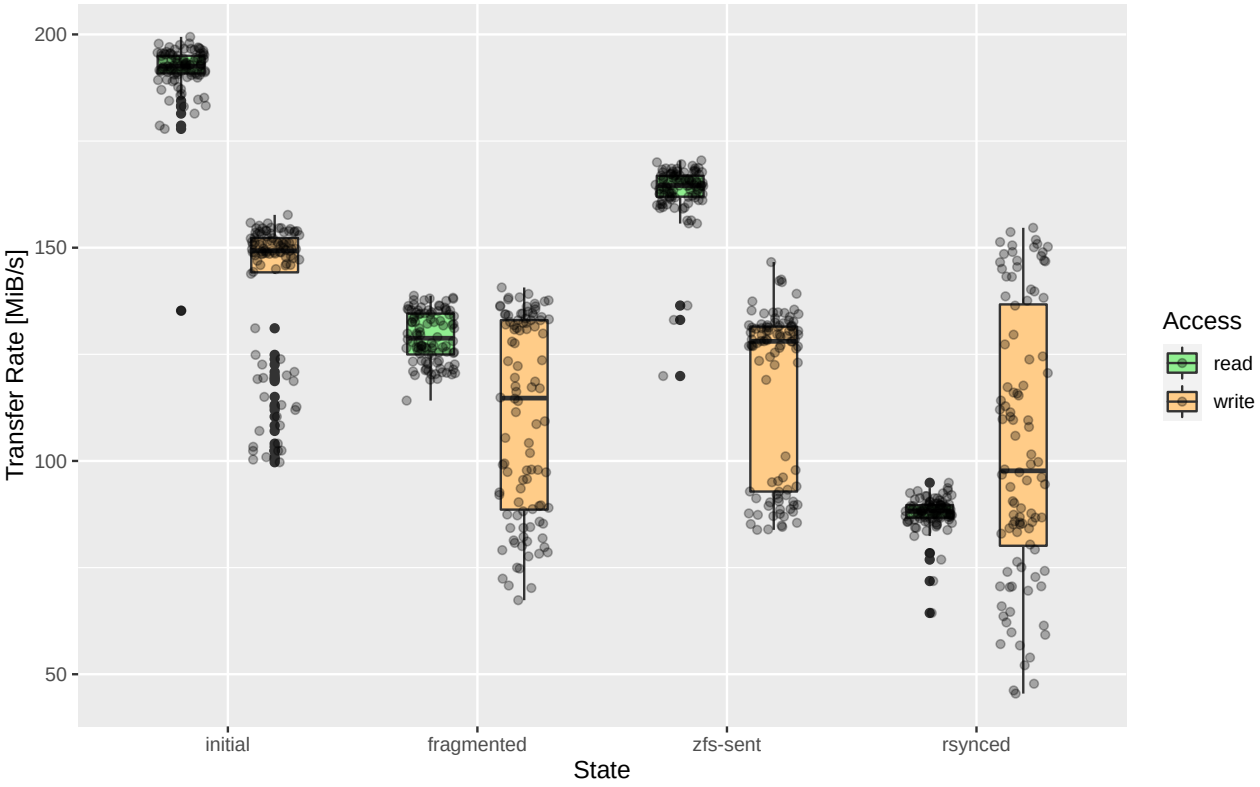


Figure 54: WD Gold 1000 GB (100 MB file size on *ZFS*)

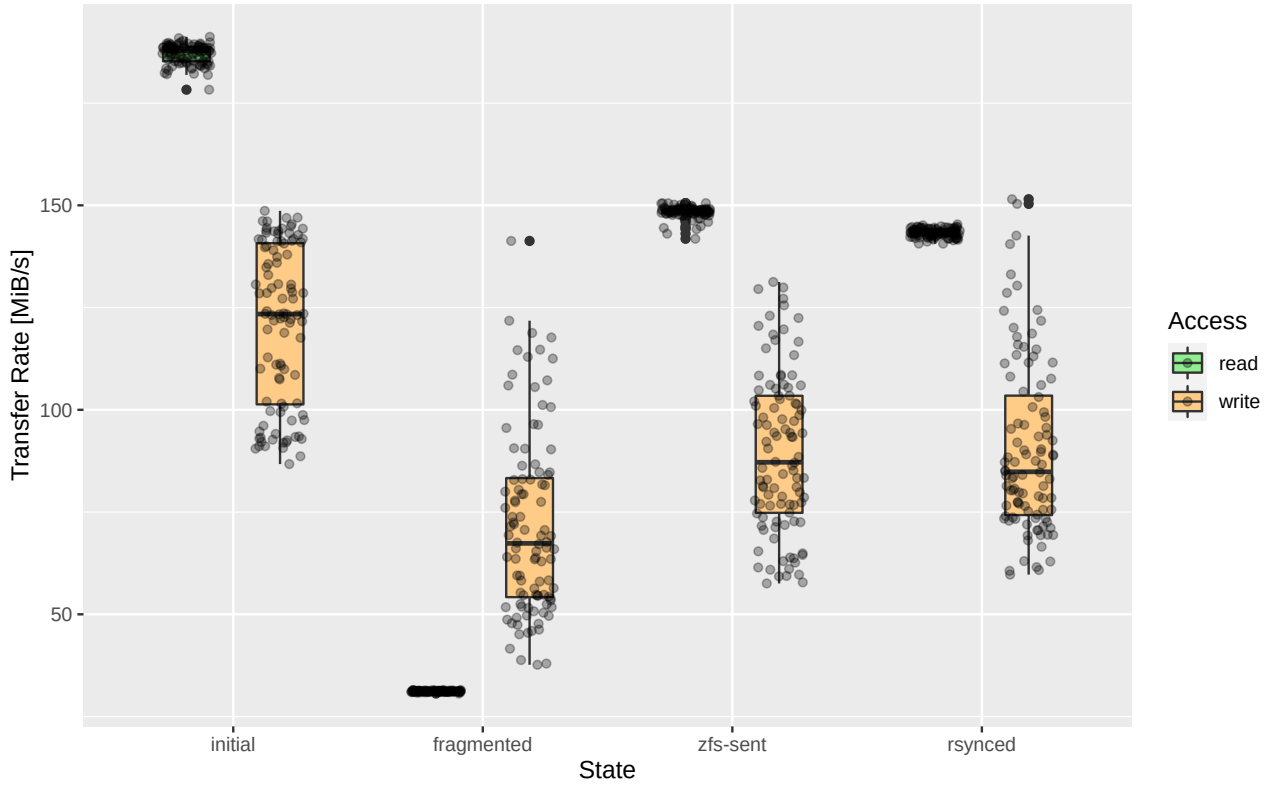


Figure 55: WD Gold 1000 GB (1 GB file size on *ZFS*)

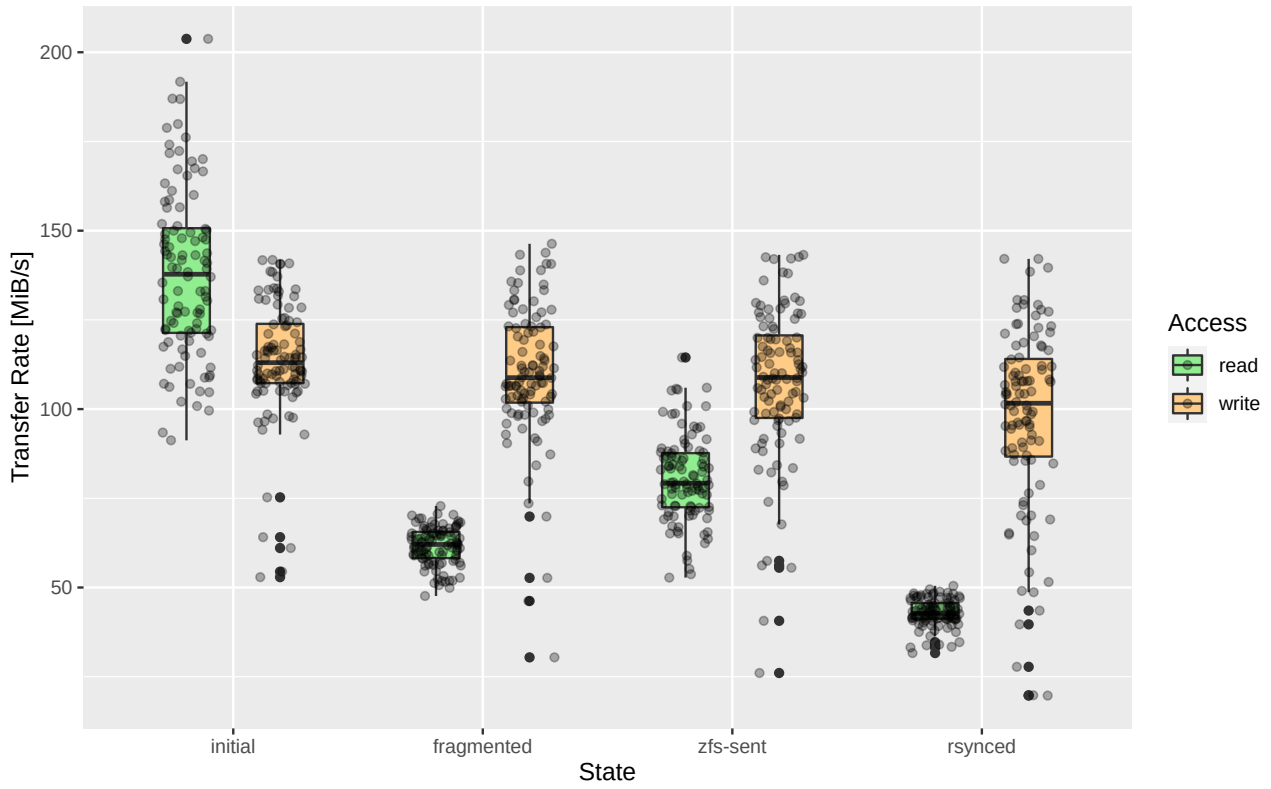


Figure 56: Seagate 500 GB RAID-0 (10 MB file size on *ZFS*)

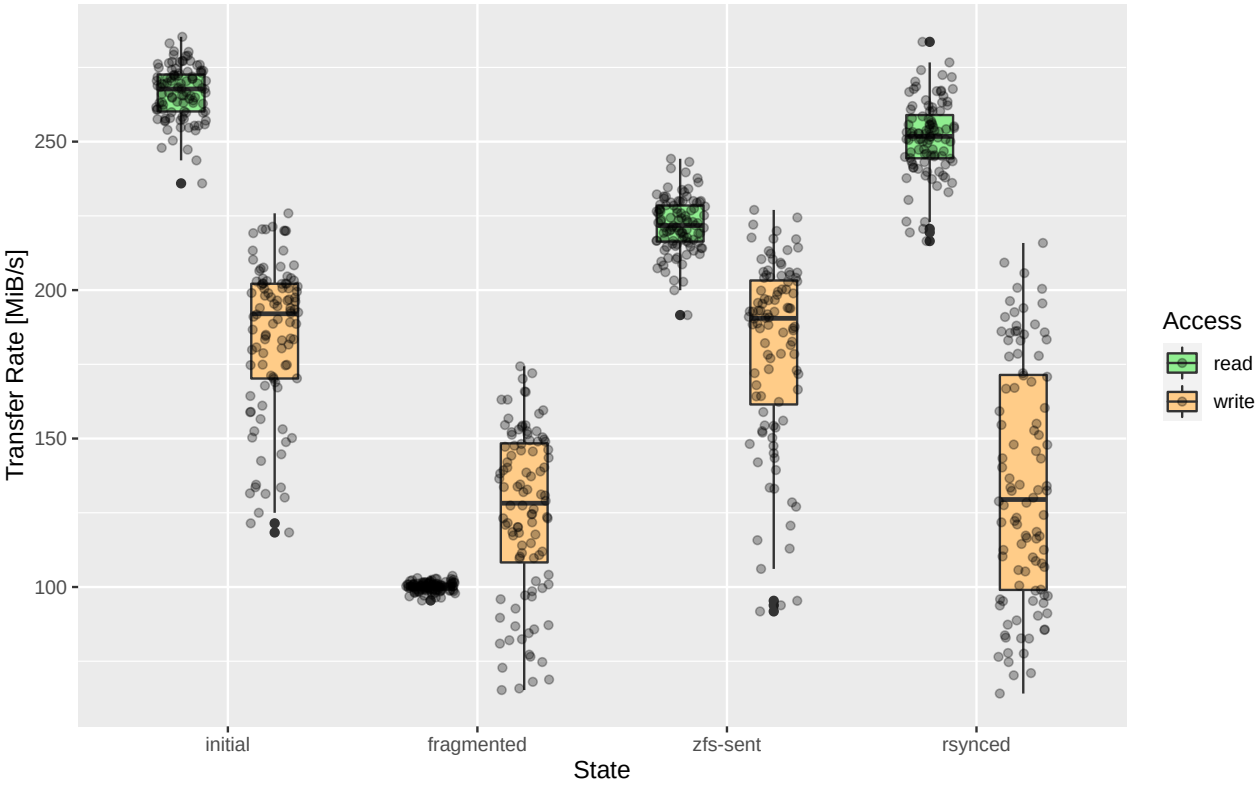


Figure 57: Seagate 500 GB RAID-0 (100 MB file size on *ZFS*)

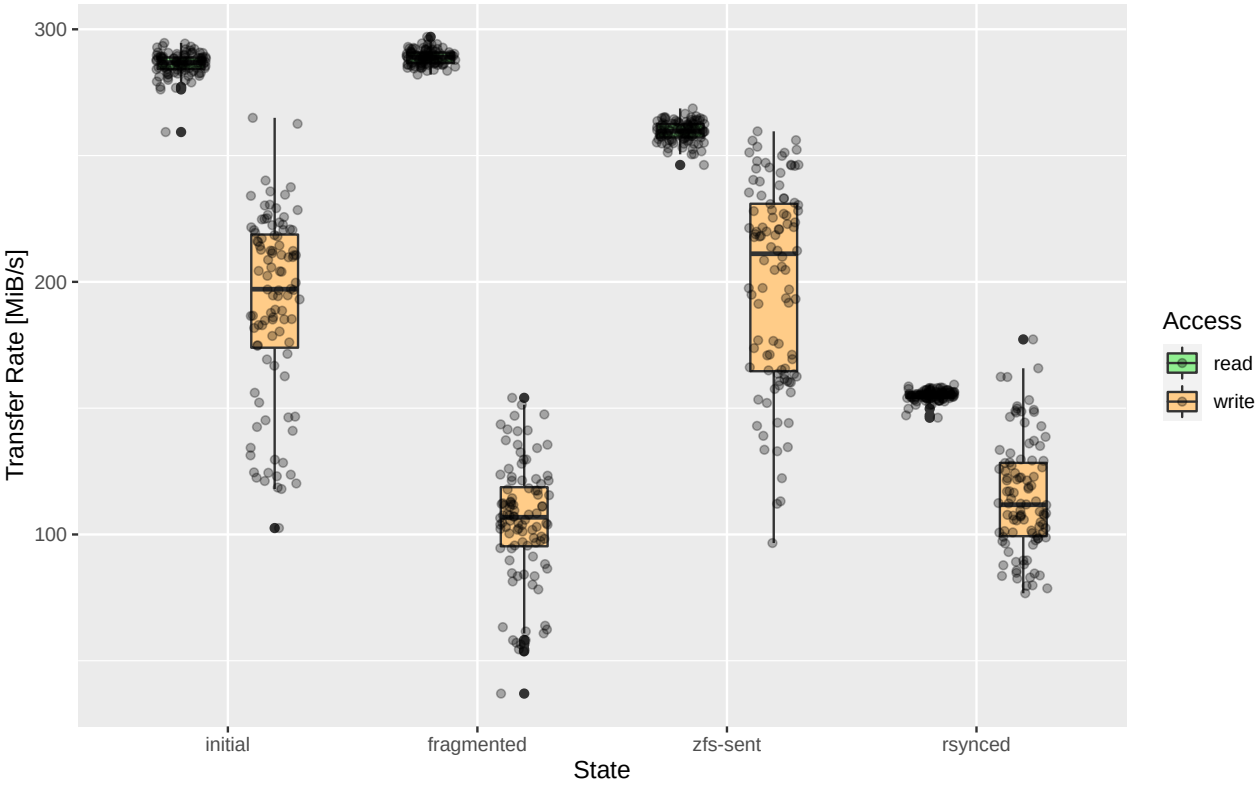


Figure 58: Seagate 500 GB RAID-0 (1 GB file size on *ZFS*)

Drive	IO	State	Mean	Standard Deviation	Range	Shapiro-Wilk Test
sg500	read	defragmented	44.21	6.45	24.69	0.00107301
sg500	read	fragmented	49.3	8.46	36.64	2.415e-05
sg500	read	initial	49.43	9.64	33.79	0.00019593
sg500	read	rsynced	58.93	13.91	56.63	0.00762721
sg500	write	defragmented	28.04	3.23	14.94	0.49151363
sg500	write	fragmented	31.97	4.94	28.36	0.00174966
sg500	write	initial	32.63	5.51	22.98	0.27068466
sg500	write	rsynced	39	8.76	35.79	0.00638585
sg1000	read	defragmented	44.09	7.32	25.24	9.51e-06
sg1000	read	fragmented	46.43	8.72	34.24	1.3e-07
sg1000	read	initial	53.69	12.23	48.34	7.916e-05
sg1000	read	rsynced	55.47	13.49	60.99	0.00131833
sg1000	write	defragmented	26.78	5.96	32.3	0
sg1000	write	fragmented	29.87	5.48	38.46	7.6e-07
sg1000	write	initial	36.56	6.75	28.86	0.0138269
sg1000	write	rsynced	33.54	6.94	29.54	8.3e-07
wdgold	read	defragmented	58.59	10.28	45.21	0.00169605
wdgold	read	fragmented	65.25	10.62	55.23	0.1584675
wdgold	read	initial	65.35	14.87	58.07	0.11557992
wdgold	read	rsynced	74.98	16.14	67.41	0.03366207
wdgold	write	defragmented	59.61	11.71	58.42	4.019e-05
wdgold	write	fragmented	62.84	11.03	51.24	0.0216231
wdgold	write	initial	64.14	12.3	43.44	0.00039563
wdgold	write	rsynced	69	13.41	66.62	7.24e-06
sg500-raid	read	defragmented	41.11	6.42	31.2	0.00249835
sg500-raid	read	fragmented	42.27	6.18	26.57	0.04946512
sg500-raid	read	initial	42.43	11.24	43.09	0.01715511
sg500-raid	read	rsynced	46.51	10.58	61.11	0.11844916
sg500-raid	write	defragmented	29.12	4.74	25.76	0.00237438
sg500-raid	write	fragmented	30.33	4.86	22.04	0.19982208
sg500-raid	write	initial	30.58	6.69	27.92	0.06513319
sg500-raid	write	rsynced	32.69	6.49	32.28	0.00893702

Table 3: Basic statistics (10 MB file size and *EXT4*) 

Drive	IO	State 1	State 2	Wilcoxon Signed-Rank Test	Difference (pct)
sg500	read	initial	fragmented	0.96297758	0
sg500	read	fragmented	defragmented	1.966e-05	-10
sg500	read	defragmented	rsynced	0	33
sg500	write	initial	fragmented	0.48411524	-2
sg500	write	fragmented	defragmented	2e-08	-12
sg500	write	defragmented	rsynced	0	39
sg1000	read	initial	fragmented	1.21e-06	-14
sg1000	read	fragmented	defragmented	0.20637924	-5
sg1000	read	defragmented	rsynced	0	26
sg1000	write	initial	fragmented	0	-18
sg1000	write	fragmented	defragmented	0.00141054	-10
sg1000	write	defragmented	rsynced	0	25
wdgold	read	initial	fragmented	0.86215201	0
wdgold	read	fragmented	defragmented	6.051e-05	-10
wdgold	read	defragmented	rsynced	0	28
wdgold	write	initial	fragmented	0.50146115	-2
wdgold	write	fragmented	defragmented	0.0235657	-5
wdgold	write	defragmented	rsynced	3.1e-07	16
sg500-raid	read	initial	fragmented	0.82716918	0
sg500-raid	read	fragmented	defragmented	0.11334203	-3
sg500-raid	read	defragmented	rsynced	7.851e-05	13
sg500-raid	write	initial	fragmented	0.97668454	-1
sg500-raid	write	fragmented	defragmented	0.07746637	-4
sg500-raid	write	defragmented	rsynced	0.00010148	12

Table 4: Wilcox and difference (10 MB file size and *EXT4*) 

Drive	IO	State	Mean	Standard Deviation	Range	Shapiro-Wilk Test
sg500	read	defragmented	52.47	7.7	31.34	0.00027352
sg500	read	fragmented	69.7	14.94	46.69	4.6e-06
sg500	read	initial	92.07	18.9	57.8	0
sg500	read	rsynced	84.36	21.41	54.44	0
sg500	write	defragmented	30.8	4.02	15.92	0.00299815
sg500	write	fragmented	48.34	8.98	31.71	0.00013056
sg500	write	initial	64.55	10.79	34.22	0
sg500	write	rsynced	64.91	11.01	35.42	0
sg1000	read	defragmented	52.95	11.35	40.22	1.221e-05
sg1000	read	fragmented	65.85	15.45	44.72	2e-08
sg1000	read	initial	89.28	20.5	55.81	0
sg1000	read	rsynced	88.91	20.77	58.11	0
sg1000	write	defragmented	30.42	6.66	26.3	0.00113056
sg1000	write	fragmented	45.06	9.51	34.44	0.00026899
sg1000	write	initial	64.63	11.45	33.72	0
sg1000	write	rsynced	64.51	11.07	37.43	0
wdgold	read	defragmented	76.11	19.29	71.63	0.00230325
wdgold	read	fragmented	102.86	24.77	94.95	0.00257047
wdgold	read	initial	145.17	38.41	112.26	0
wdgold	read	rsynced	148.17	40.32	105.21	0
wdgold	write	defragmented	92.47	17.68	59.4	1.73e-06
wdgold	write	fragmented	104.19	17.5	51.8	8.9e-07
wdgold	write	initial	113.04	18.95	62.45	0
wdgold	write	rsynced	114.62	17.29	57.95	0
sg500-raid	read	defragmented	41.35	8.51	33.41	0.00226788
sg500-raid	read	fragmented	62.08	12.63	42.89	7.5e-07
sg500-raid	read	initial	66.48	27.74	91.09	1e-08
sg500-raid	read	rsynced	79.96	27.1	85.54	0
sg500-raid	write	defragmented	25.53	4.35	20.69	0.00718971
sg500-raid	write	fragmented	43.62	9.36	30.74	1.9e-07
sg500-raid	write	initial	54.92	10.5	37.53	2e-08
sg500-raid	write	rsynced	62.23	12.7	40.89	0

Table 5: Basic statistics (100 MB file size and *EXT4*)

Drive	IO	State 1	State 2	Wilcoxon Signed-Rank Test	Difference (pct)
sg500	read	initial	fragmented	0	-24
sg500	read	fragmented	defragmented	0	-25
sg500	read	defragmented	rsynced	0	61
sg500	write	initial	fragmented	0	-25
sg500	write	fragmented	defragmented	0	-36
sg500	write	defragmented	rsynced	0	111
sg1000	read	initial	fragmented	0	-26
sg1000	read	fragmented	defragmented	1e-08	-20
sg1000	read	defragmented	rsynced	0	68
sg1000	write	initial	fragmented	0	-30
sg1000	write	fragmented	defragmented	0	-32
sg1000	write	defragmented	rsynced	0	112
wdgold	read	initial	fragmented	0	-29
wdgold	read	fragmented	defragmented	0	-26
wdgold	read	defragmented	rsynced	0	95
wdgold	write	initial	fragmented	0.00026246	-8
wdgold	write	fragmented	defragmented	4.19e-05	-11
wdgold	write	defragmented	rsynced	0	24
sg500-raid	read	initial	fragmented	0.14440859	-7
sg500-raid	read	fragmented	defragmented	0	-33
sg500-raid	read	defragmented	rsynced	0	93
sg500-raid	write	initial	fragmented	0	-21
sg500-raid	write	fragmented	defragmented	0	-41
sg500-raid	write	defragmented	rsynced	0	144

Table 6: Wilcox and difference (100 MB file size and *EXT4*) 

Drive	IO	State	Mean	Standard Deviation	Range	Shapiro-Wilk Test
sg500	read	defragmented	47.2	4.02	17.64	0.27154235
sg500	read	fragmented	76.01	9.88	37.82	0.00930818
sg500	read	initial	100.91	10.43	43.84	0
sg500	read	rsynced	98.18	11.59	45.68	0
sg500	write	defragmented	28.2	2.09	11.79	0.04401622
sg500	write	fragmented	47.03	4.41	22.18	0.44609191
sg500	write	initial	70.19	4.65	19	1.047e-05
sg500	write	rsynced	70.53	4.74	21.61	1e-08
sg1000	read	defragmented	56.6	5.71	22.68	0.00835585
sg1000	read	fragmented	75.3	8.93	35.9	0.00865805
sg1000	read	initial	101.74	10.59	47.05	0
sg1000	read	rsynced	90.13	18.72	53.31	1e-08
sg1000	write	defragmented	32.54	2.89	13.9	0.00359657
sg1000	write	fragmented	43.84	3.37	19.6	0.93515915
sg1000	write	initial	71.94	4.24	19.44	6.9e-07
sg1000	write	rsynced	69.87	5.26	22.51	6.57e-06
wdgold	read	defragmented	77.16	9.74	42.56	0.20469724
wdgold	read	fragmented	94.95	13.01	50.59	0.0544366
wdgold	read	initial	148.99	27.8	94.98	4.316e-05
wdgold	read	rsynced	148.15	29.68	108.56	0.00012406
wdgold	write	defragmented	95.73	11.84	48.03	0.11933707
wdgold	write	fragmented	108.31	9.9	42.86	0.0241924
wdgold	write	initial	119.96	9.25	39.54	2.6e-07
wdgold	write	rsynced	120.89	10.71	43.28	1e-08
sg500-raid	read	defragmented	42.32	3.58	16.71	0.00020465
sg500-raid	read	fragmented	66.1	9.16	37.88	0.01337835
sg500-raid	read	initial	37.65	12.65	59.52	5.73e-06
sg500-raid	read	rsynced	45.19	15.62	84.62	2.4e-07
sg500-raid	write	defragmented	26.04	2.01	15.07	4.78e-06
sg500-raid	write	fragmented	41.71	3.89	20.02	0.86167009
sg500-raid	write	initial	62.52	9.58	41.36	0.00719621
sg500-raid	write	rsynced	70.41	6.35	29.67	1.1e-07

Table 7: Basic statistics (1 GB file size and *EXT4*)

Drive	IO	State 1	State 2	Wilcoxon Signed-Rank Test	Difference (pct)
sg500	read	initial	fragmented	0	-25
sg500	read	fragmented	defragmented	0	-38
sg500	read	defragmented	rsynced	0	108
sg500	write	initial	fragmented	0	-33
sg500	write	fragmented	defragmented	0	-40
sg500	write	defragmented	rsynced	0	150
sg1000	read	initial	fragmented	0	-26
sg1000	read	fragmented	defragmented	0	-25
sg1000	read	defragmented	rsynced	0	59
sg1000	write	initial	fragmented	0	-39
sg1000	write	fragmented	defragmented	0	-26
sg1000	write	defragmented	rsynced	0	115
wdgold	read	initial	fragmented	0	-36
wdgold	read	fragmented	defragmented	0	-19
wdgold	read	defragmented	rsynced	0	92
wdgold	write	initial	fragmented	0	-10
wdgold	write	fragmented	defragmented	0	-12
wdgold	write	defragmented	rsynced	0	26
sg500-raid	read	initial	fragmented	0	76
sg500-raid	read	fragmented	defragmented	0	-36
sg500-raid	read	defragmented	rsynced	0.41415583	7
sg500-raid	write	initial	fragmented	0	-33
sg500-raid	write	fragmented	defragmented	0	-38
sg500-raid	write	defragmented	rsynced	0	170

Table 8: Wilcox and difference (1 GB file size and *EXT4*) 

Drive	IO	State	Mean	Standard Deviation	Range	Shapiro-Wilk Test
sg500	read	fragmented	83.9	9.19	49.25	0.52963569
sg500	read	initial	118.45	14.32	68.49	0.59833096
sg500	read	rsynced	80.58	8.11	48.1	1e-06
sg500	read	zfs-sent	82.31	11.3	72.33	1.284e-05
sg500	write	fragmented	77.57	9.96	73.12	0
sg500	write	initial	72.09	8.23	51.62	1e-08
sg500	write	rsynced	62.38	10.19	53.22	0
sg500	write	zfs-sent	64.2	11.08	60.03	0
sg1000	read	fragmented	53.3	3.47	25.93	2e-07
sg1000	read	initial	122.96	18.64	80.36	0.0589943
sg1000	read	rsynced	99.64	9.32	48.73	6.319e-05
sg1000	read	zfs-sent	86.43	9.32	55.65	0.06493208
sg1000	write	fragmented	42.61	9.41	49.47	0.00292187
sg1000	write	initial	71.65	10.41	72.13	0
sg1000	write	rsynced	62.58	9.2	61.81	0
sg1000	write	zfs-sent	70.81	9.07	60.69	0
wdgold	read	fragmented	70.14	11.97	69.75	9.886e-05
wdgold	read	initial	138.31	25	144.88	1e-08
wdgold	read	rsynced	64.46	10.37	63.03	0
wdgold	read	zfs-sent	100.14	9.72	67.47	0
wdgold	write	fragmented	71.63	10.95	67.9	0
wdgold	write	initial	90.09	10.25	56.5	0
wdgold	write	rsynced	66.25	18.28	86.73	0.00011372
wdgold	write	zfs-sent	82.57	11.74	71.92	0
sg500-raid	read	fragmented	61.58	5.38	25.2	0.10520516
sg500-raid	read	initial	138.09	23.68	112.5	0.27388515
sg500-raid	read	rsynced	42.95	3.85	18.84	0.00378858
sg500-raid	read	zfs-sent	80.45	12.15	61.68	0.64925035
sg500-raid	write	fragmented	109.8	19.49	115.9	2.416e-05
sg500-raid	write	initial	113.48	16.96	88.9	7.4e-07
sg500-raid	write	rsynced	97.48	26.14	122.38	0.00029674
sg500-raid	write	zfs-sent	107.41	21.59	117.16	0.00017791

Table 9: Basic statistics (10 MB file size and *ZFS*) 

Drive	IO	State 1	State 2	Wilcoxon Signed-Rank Test	Difference (pct)
sg500	read	initial	fragmented	0	-29
sg500	read	fragmented	zfs-sent	0.4587183	-2
sg500	read	zfs-sent	rsynced	0.0724111	-2
sg500	write	initial	fragmented	0	8
sg500	write	fragmented	zfs-sent	0	-17
sg500	write	zfs-sent	rsynced	0.02001712	-3
sg1000	read	initial	fragmented	0	-57
sg1000	read	fragmented	zfs-sent	0	62
sg1000	read	zfs-sent	rsynced	0	15
sg1000	write	initial	fragmented	0	-41
sg1000	write	fragmented	zfs-sent	0	66
sg1000	write	zfs-sent	rsynced	0	-12
wdgold	read	initial	fragmented	0	-49
wdgold	read	fragmented	zfs-sent	0	43
wdgold	read	zfs-sent	rsynced	0	-36
wdgold	write	initial	fragmented	0	-20
wdgold	write	fragmented	zfs-sent	0	15
wdgold	write	zfs-sent	rsynced	0	-20
sg500-raid	read	initial	fragmented	0	-55
sg500-raid	read	fragmented	zfs-sent	0	31
sg500-raid	read	zfs-sent	rsynced	0	-47
sg500-raid	write	initial	fragmented	0.10351496	-3
sg500-raid	write	fragmented	zfs-sent	0.41219303	-2
sg500-raid	write	zfs-sent	rsynced	0.00094655	-9

Table 10: Wilcox and difference (10 MB file size and *ZFS*) 

Drive	IO	State	Mean	Standard Deviation	Range	Shapiro-Wilk Test
sg500	read	fragmented	164.45	3.81	20.68	0.00061389
sg500	read	initial	173.32	2.7	20.2	0
sg500	read	rsynced	124.51	3.02	25.38	0
sg500	read	zfs-sent	139.28	3.48	15.43	0.26322236
sg500	write	fragmented	104.21	17.62	54.62	0
sg500	write	initial	102.9	18.33	72.07	0.00242688
sg500	write	rsynced	84.86	17	66.45	0.0008218
sg500	write	zfs-sent	86.61	17.62	62.84	0.00083088
sg1000	read	fragmented	151.57	2.52	14.66	0
sg1000	read	initial	185.65	3.67	14.84	0.00024208
sg1000	read	rsynced	146.91	2.25	11.87	0.01898785
sg1000	read	zfs-sent	181.87	8.26	77.06	0
sg1000	write	fragmented	65.15	17.35	67.45	0.00438867
sg1000	write	initial	124.36	22.55	81.07	8e-08
sg1000	write	rsynced	82.55	13.48	45.85	5.19e-06
sg1000	write	zfs-sent	107.7	20.18	72.97	2.542e-05
wdgold	read	fragmented	129.07	5.88	24.58	0.00122392
wdgold	read	initial	191.58	6.95	64.19	0
wdgold	read	rsynced	87.75	3.97	30.52	0
wdgold	read	zfs-sent	163.32	6.86	50.54	0
wdgold	write	fragmented	110.25	23	73.31	3.3e-07
wdgold	write	initial	141.54	17.31	58	0
wdgold	write	rsynced	103.2	31.74	109.2	0.00028715
wdgold	write	zfs-sent	117.42	19.76	62.77	0
sg500-raid	read	fragmented	100.1	1.54	8.31	0.55018359
sg500-raid	read	initial	266.41	8.91	49.35	0.10250677
sg500-raid	read	rsynced	251.26	12.45	67.09	0.20254174
sg500-raid	read	zfs-sent	221.91	9.31	52.65	0.76377801
sg500-raid	write	fragmented	125.09	27.94	109.03	0.00331588
sg500-raid	write	initial	184.21	25.73	107.5	4.819e-05
sg500-raid	write	rsynced	134.79	40.43	151.74	0.00111765
sg500-raid	write	zfs-sent	180.04	31.39	135.22	5.83e-06

Table 11: Basic statistics (100 MB file size and *ZFS*) 

Drive	IO	State 1	State 2	Wilcoxon Signed-Rank Test	Difference (pct)
sg500	read	initial	fragmented	0	-5
sg500	read	fragmented	zfs-sent	0	-15
sg500	read	zfs-sent	rsynced	0	-11
sg500	write	initial	fragmented	0.93833532	1
sg500	write	fragmented	zfs-sent	2e-08	-17
sg500	write	zfs-sent	rsynced	0.51468341	-2
sg1000	read	initial	fragmented	0	-18
sg1000	read	fragmented	zfs-sent	0	20
sg1000	read	zfs-sent	rsynced	0	-19
sg1000	write	initial	fragmented	0	-48
sg1000	write	fragmented	zfs-sent	0	65
sg1000	write	zfs-sent	rsynced	0	-23
wdgold	read	initial	fragmented	0	-33
wdgold	read	fragmented	zfs-sent	0	27
wdgold	read	zfs-sent	rsynced	0	-46
wdgold	write	initial	fragmented	0	-22
wdgold	write	fragmented	zfs-sent	0.01349483	6
wdgold	write	zfs-sent	rsynced	0.00051195	-12
sg500-raid	read	initial	fragmented	0	-62
sg500-raid	read	fragmented	zfs-sent	0	122
sg500-raid	read	zfs-sent	rsynced	0	13
sg500-raid	write	initial	fragmented	0	-32
sg500-raid	write	fragmented	zfs-sent	0	44
sg500-raid	write	zfs-sent	rsynced	0	-25

Table 12: Wilcox and difference (100 MB file size and *ZFS*) 

Drive	IO	State	Mean	Standard Deviation	Range	Shapiro-Wilk Test
sg500	read	fragmented	174.24	1.94	13.82	0
sg500	read	initial	175.92	1.99	10.27	0
sg500	read	rsynced	98.01	0.77	4.97	0.00019326
sg500	read	zfs-sent	135.31	1.14	5.03	1e-07
sg500	write	fragmented	91.32	7.46	37.91	0.04555898
sg500	write	initial	83.01	12.38	76.77	0.00142916
sg500	write	rsynced	70.9	15.27	58.88	0.00359131
sg500	write	zfs-sent	73.65	16.56	60.9	0.00111735
sg1000	read	fragmented	54.98	0.28	1.63	0.00027441
sg1000	read	initial	187.29	2.34	18.55	0
sg1000	read	rsynced	92.57	0.61	3.11	9.861e-05
sg1000	read	zfs-sent	185.47	2.11	12.6	0
sg1000	write	fragmented	50.12	9.41	42.27	0.01372056
sg1000	write	initial	99.14	20.44	80.76	2.124e-05
sg1000	write	rsynced	69.39	12.64	58.37	0.33846
sg1000	write	zfs-sent	80.95	17.76	79.69	0.06479438
wdgold	read	fragmented	31.17	0.15	0.85	0.11369123
wdgold	read	initial	187.04	2.34	12.93	6.613e-05
wdgold	read	rsynced	143.26	1.03	4.72	0.16917608
wdgold	read	zfs-sent	148.38	1.37	8.72	0
wdgold	write	fragmented	71.58	22.18	103.63	0.00017182
wdgold	write	initial	121.03	19.5	61.92	2.2e-06
wdgold	write	rsynced	90.49	21.18	91.78	1.733e-05
wdgold	write	zfs-sent	89.56	19.4	73.71	0.01877421
sg500-raid	read	fragmented	288.9	2.9	14.95	0.78042096
sg500-raid	read	initial	286.26	4.66	35.26	6e-08
sg500-raid	read	rsynced	155.11	2.11	13.11	9.09e-06
sg500-raid	read	zfs-sent	259.52	4.02	22.38	0.14914147
sg500-raid	write	fragmented	104.74	24.1	117.09	0.00935157
sg500-raid	write	initial	190.86	36.78	162.42	7.772e-05
sg500-raid	write	rsynced	114.74	22.19	100.52	0.0233155
sg500-raid	write	zfs-sent	199.63	39.88	163	0.000266

Table 13: Basic statistics (1 GB file size and *ZFS*)

Drive	IO	State 1	State 2	Wilcoxon Signed-Rank Test	Difference (pct)
sg500	read	initial	fragmented	1e-08	-1
sg500	read	fragmented	zfs-sent	0	-22
sg500	read	zfs-sent	rsynced	0	-28
sg500	write	initial	fragmented	2.4e-07	10
sg500	write	fragmented	zfs-sent	0	-19
sg500	write	zfs-sent	rsynced	0.30961749	-4
sg1000	read	initial	fragmented	0	-71
sg1000	read	fragmented	zfs-sent	0	237
sg1000	read	zfs-sent	rsynced	0	-50
sg1000	write	initial	fragmented	0	-49
sg1000	write	fragmented	zfs-sent	0	62
sg1000	write	zfs-sent	rsynced	2.9e-06	-14
wdgold	read	initial	fragmented	0	-83
wdgold	read	fragmented	zfs-sent	0	376
wdgold	read	zfs-sent	rsynced	0	-3
wdgold	write	initial	fragmented	0	-41
wdgold	write	fragmented	zfs-sent	1e-08	25
wdgold	write	zfs-sent	rsynced	0.88381971	1
sg500-raid	read	initial	fragmented	1.052e-05	1
sg500-raid	read	fragmented	zfs-sent	0	-10
sg500-raid	read	zfs-sent	rsynced	0	-40
sg500-raid	write	initial	fragmented	0	-45
sg500-raid	write	fragmented	zfs-sent	0	91
sg500-raid	write	zfs-sent	rsynced	0	-43

Table 14: Wilcox and difference (1 GB file size and *ZFS*)