OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Bachelor Thesis

Interactive Layout Visualization of Distributed Storage

Lennart Börchers
lennart.boerchers@ovgu.de

June 13, 2023

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

Second Reviewer:
Dr. Jakob Lüttgau

Supervisor:
Michael Kuhn, Jakob Lüttgau, Anna Fuchs

Abstract

As high-performance computing systems have evolved over the years to address increasingly complex questions, the speed of storage access has fallen behind and often poses a potential bottleneck, particularly in scientific computing. But existing command-line tools for analyzing the storage layout are largely textual, which can be unintuitive and makes it difficult to recognize suboptimal data distributions and find their underlying causes.
To counteract this issue, I developed interactive visualizations that provide an intuitive way of detecting and exploring these inefficiencies in four steps: First, the existing textual representations are extracted from the system and afterwards parsed into usable formats. Subsequently, a visualization library is used to transform the parsed data into graphs, multiple of which can finally be connected to create an interactive visualization.
Interactive visualizations serve as a good tool for illustrating storage imbalances and finding potential causes. However, effectiveness can vary depending on the distributed system, use case, and user.

Contents

Chapter 1.

Introduction

1.1. Motivation

In recent decades, the rapid advancement of computers has led to significant improvements in their speed and processing capabilities. Alongside these advancements, the problems they are used for have also increased in size and complexity. Be it simulating much of the earth for better climate predictions and weather forecasts, mimicking the human brain to create ever more intricate artificial intelligence, or modeling the molecular structure of proteins to find and improve cures for various diseases.

These supercomputers provide the computing power necessary by massively parallelizing their workload across numerous nodes. Nodes can be bundles of processors for computing small parts of a larger problem or storage devices to record the necessary data for and results of computations. For instance, distributed systems like the "HLRE-4 Levante" [DKRZ, nd] can comprise nearly 3000 computing nodes with over 370,000 cores, enabling approximately 14 quadrillion mathematical operations per second and offering over a hundred petabytes of storage. More advanced supercomputers can achieve even higher performance. This makes them expensive to build, keep cool, and supply with electricity, which is even more expensive for the environment. Thus, using all the available resources as efficiently as possible is not only an economic imperative, but also an ecological one.

For this, it is important to use all available storage devices roughly equally, to parallelize data access and balance the data load. 100 SSDs are not very helpful if the first 50 are filled to the brim, while the other 50 stand around empty. While such extreme cases are unlikely, similar situations are rather common. Both files and the storage devices they are recorded on are constantly in flux, with new ones being added and old ones removed or changed. Distributing data equally is a constant task that never ends.

But existing tools to analyze the storage layout of a distributed system are solely textual, consisting of thousands of lines of dense text. This makes it difficult for humans to spot outliers and potential problems, grasp the entire architecture, and find the cause of inefficiencies quickly. Visualizing the extensive text-based data can help users better understand their system. By utilizing charts, large amounts of information can be condensed and presented in an understandable fashion. Allowing users to interact with these charts empowers them to compare, correlate, and select different aspects to improve their understanding even further.

1.2. Goals

Hence, the primary focus of this thesis is to create novel tools that transform the textual outputs of existing command-line utilities into the aforementioned interactive graphs. These visual representations aim to provide users with a clearer and more comprehensive understanding of the storage layout, enabling them to quickly identify patterns, trends, and anomalies that may not be readily apparent in written format.
As this is a mostly underdeveloped aspect of supercomputing, it remains uncertain what is feasible or not, for which users and systems: Each distributed system varies in its approach to storage and might not record some critical information or restrict access to it. Therefor, I will also illustrate which visualizations could be used more generally, and which ones are specific to a unique system, noting who could achieve this.

As such, the research question I strive to answer in this thesis is: How can interactive visualizations enhance the understanding of the storage layout of distributed systems like Lustre or Ceph?

1.3. Outline

The remainder of the thesis is structured as follows: After this introduction, where I explained the importance of visualizing the textual outputs of distributed systems, I go over the necessary background information to understand this thesis: the inner workings of both distributed systems I worked with, Lustre and Ceph, as well as the visualization libraries I used and the basics about these visualizations.
Afterwards, I focus on the design of the tools I developed and how the textual storage information of distributed systems is gathered, then parsed to extract the interesting data and turn it into a format that can be read and turned into graphs by the visualization tools. Finally, the different graphs are connected together to provide an increased level of interactivity that allows for a better understanding and exploration of the storage layout.
Then I quickly go over related works of other authors that already build similar visualizations for supercomputers and show how they differ from my work.
Thereafter comes the evaluation of the tools I developed, describing the environments I performed my tests in and interpreting the results I gathered. Last but not least is the conclusion, where I discuss which parts of my work succeeded but also where it fell short, and thus where future works might pick up to improve and add to the tools I developed.

Chapter 2.

Background

In the following, I will provide an overview of the fundamentals of the two distributed storage systems I experimented with, Lustre and Ceph, as well as briefly mention other similar systems. Afterwards, I will introduce the visualization libraries used, Plotly/Dash, and go over the basics of graphs, the different kinds there are, and how and when to use them.

2.1. Distributed Storage Systems

2.1.1. Lustre

Lustre is a distributed file system that is used by many supercomputers for large-scale storage and access to files by potentially hundreds of thousands of users. To achieve this, it is made up of five components [Lustre, nd] [Lustre, 2010]:

1. The Object Storage Targets (OST), the physical devices that store the files, be they HDD or SSD. But users can not simply access these OSTs directly, as this might pose a security risk and allow malicious actors to corrupt the data. Instead, they can only communicate with an Object Storage Server (OSS) that manages one or several OSTs and handles the input and output (I/O) requests.

2. To store the namespace metadata about the files, be they filenames, file layouts, or access permissions, Lustre uses storage devices called Meta Data Targets (MDT), though these can also only be accessed through an intermediate server, the Meta Data Server (MDS). They help users find the correct OSS for their I/O request and give them the location of the desired data. That is, if they have the necessary permission to do so, which the MDS also checks. But they do not handle the I/O operations because the OSSs do so themselves as needed, avoiding a bottleneck at the MDTs.

3. To store information about its configuration, Lustre has a Management Server (MGS) with a corresponding Management Target (MGT) as its storage device.

4. Clients are the users reading and writing files, tasking MDS with the creation or deletion of files, and much more.

5. To connect all the different parts Lustre uses Lustre Networking (LNet), a high-speed data network protocol that works on various network technologies like Ethernet [Metcalfe and Boggs, 1976] or InfiniBand.

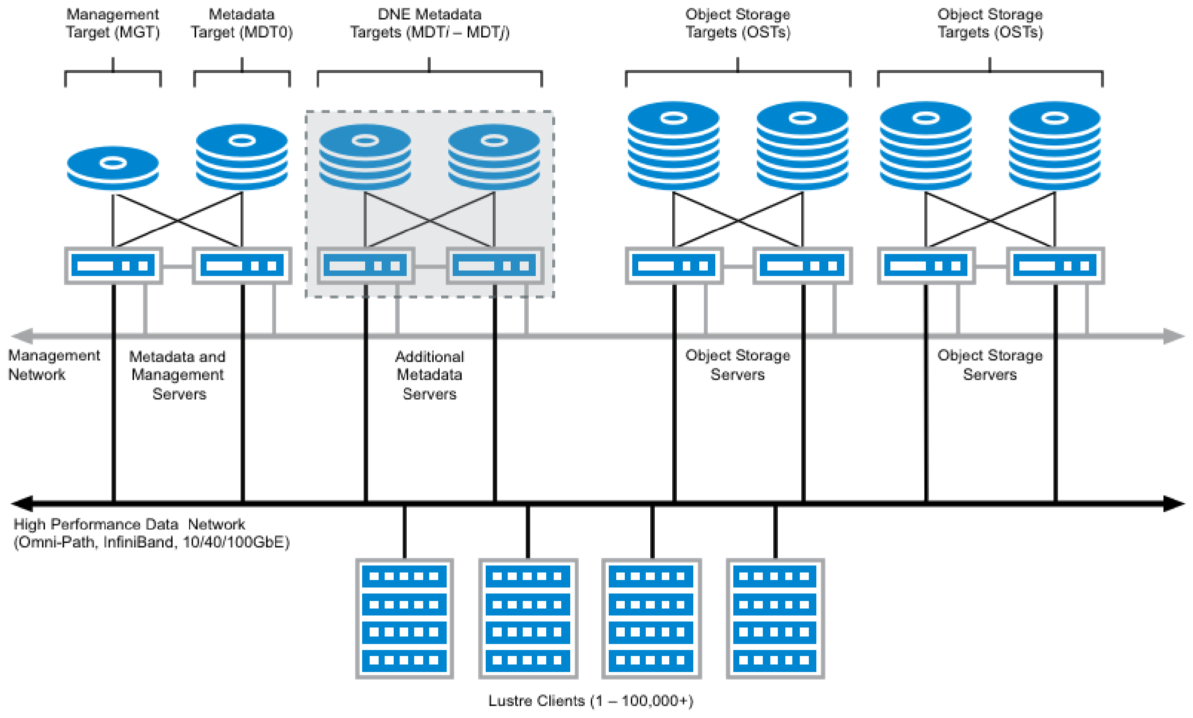

Management Target (MGT)
Metadata Target (MDT0)
DNE Metadata Targets (MDT*i* – MDT*j*)
Object Storage Targets (OSTs)
Object Storage Targets (OSTs)
Management Network
Metadata and Management Servers
Additional Metadata Servers
Object Storage Servers
Object Storage Servers
High Performance Data Network (Omni-Path, InfiniBand, 10/40/100GbE)
Lustre Clients (1 – 100,000+)

Figure 2.1.: Schematic architecture showing the five components of Lustre from [Lustre, nd]

Figure 2.1 illustrates the relationship between these components: At the bottom are the clients which communicate with the Management, Metadata, and Object Storage Servers that provide the interface to their respective Management, Metadata, and Object Storage Targets, all connected through LNet.

To utilize multiple storage devices for faster parallel access, files can be striped across several OSTs. How large these stripes are, over how many and which devices they are stored can be both manually configured or set by the default configuration of a directory. The many stripes are then distributed over the OSTs in a round-robin manner. While striping a file across more devices increases the bandwidth at which it can be read and written, it also incurs an overhead cost for managing all the stripes and increases the risk of contention. Finding a balance between the two is crucial. For optimal performance, it is also good practice to use all storage devices roughly equally so that there is no unnecessary congestion or conflict over resources.

Sometimes, especially for very large files, it is helpful to use a Progressive File Layout (PFL) where certain parts of the file are striped differently depending on their size, because as a file grows or shrinks in size the best way to stripe it also changes. Users can specify byte ranges and how the parts of a file in those ranges should be striped. The example in Figure 2.2 of a storage layout might be created with the command `"setstripe -E 4M -c 1 -E 64M -c 4 -E -1 -c -1 -i 4"` where the first 4 MB of a file would be split over just a single OST into 1 MB parts. Till 64 MB it would be striped over four OSTs and the rest of the file would be distributed over all available OSTs.

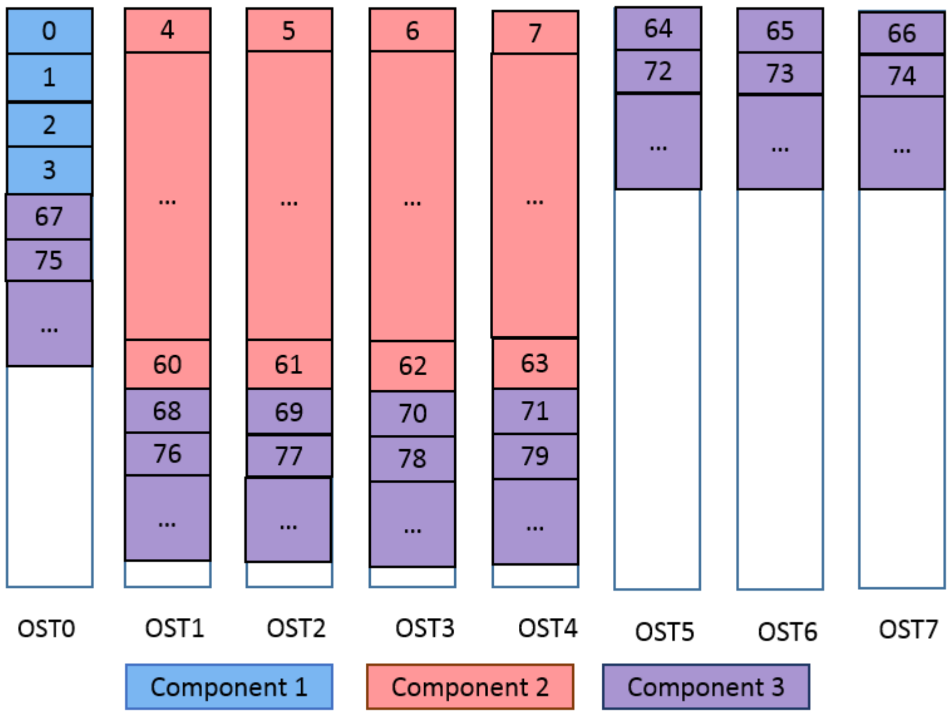


Figure 2.2.: Exemplary Progressive File Layout from the Lustre Manual [Lustre, 2010]

2.1.2. Ceph

Ceph [Ceph-Foundation, nd] is, similarly to Lustre, a distributed storage system that can store petabytes of data on potentially hundreds or thousands of Object Storage Daemons (OSD). In contrast to Lustre Ceph uses an algorithm called "CRUSH" [Weil et al., 2006] to store and replicate all files on the cluster, while allowing clients to use the same algorithm to access their desired files. Through CRUSH it has self-healing capacities and can recover most device failures. It can also quickly reshuffle data when old OSDs are removed or new ones added, with minimal work load.

Ceph is made up of these four daemons:

1. as mentioned **OSDs** actually store the objects and peer with each other to verify their integrity, keep up to date, distribute the data load and communicate the state of the cluster. This means that a lot of work can be done by them among themselves, with no central bottleneck.

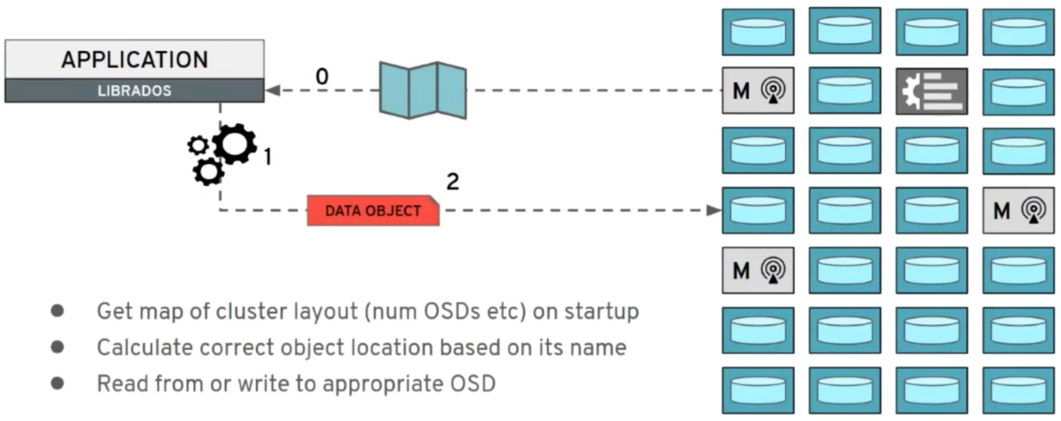
2. **Ceph Monitors** keep track of the cluster membership, so which OSDs are currently in and working, and which are out of the cluster and currently at least not usable. It also authenticates clients, coordinates the other components and provides external services with an interface to the cluster.

3. **Ceph Managers** perform bookkeeping and maintenance tasks, recording the performance of the cluster and providing the information for people to oversee Ceph.

4. **Ceph Metadata Servers** store, as the name suggests, all the metadata and, for example, tell users where to find a desired file, though the specific OSD still serves this request itself.

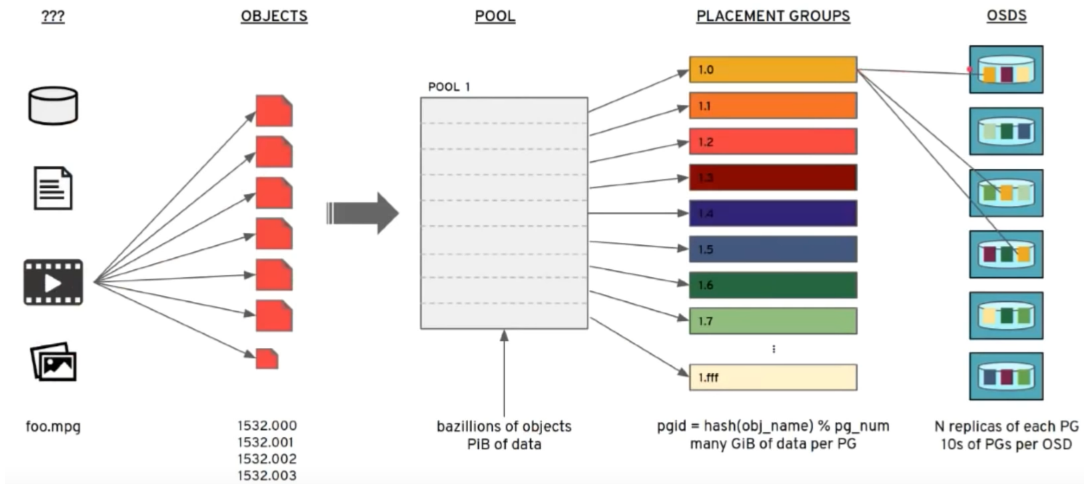
As can be seen in Figure 2.3, to determine the placement of an object in Ceph, a user or application first needs to retrieve the cluster map from a Ceph monitor. With the cluster map and the name of the object, it is then possible to calculate the location of the object and read from or write to the correct OSD.

In conclusion, when storing a file of any kind, Ceph typically breaks it down into many small objects that belong to a pool that groups similar files and their objects into a single unit. This pool is then broken down into a certain number of placement groups, and each object is pseudo-randomly mapped to one of them. Only these PGs are then stored on an OSD and potentially replicated onto certain others, depending on its pool parameters.



- Get map of cluster layout (num OSDs etc) on startup
- Calculate correct object location based on its name
- Read from or write to appropriate OSD

(a) The Ceph placement procedure



(b) A single file is split into a number of objects, that all belong to the same pool. All object in the pool are pseudo-randomly distributed across placement groups, which are stored and replicated on OSDs

Figure 2.3.: Overview of Ceph and how it stores a file from [Weil, 2019]

2.1.3. CRUSH Algorithm

While Lustre uses explicit metadata to keep track of its data, Ceph instead relies on a mapping function to empower clients to efficiently both find old and place new data, while adapting to changes in the topology of the system. This mapping function is the Controlled, Scalable, Decentralized Placement of Replicated Data (CRUSH) algorithm [Weil et al., 2006]. It distributes data by taking the **state of the system**, the **pool parameters** of the pool the object is in, and the **placement group** to determine the primary and secondary storage devices that an object should be stored in.

The **state of the system** is encoded as a hierarchical cluster map that reflects the topology of the physical storage resources. In this tree all storage devices are represented with leaves and can be grouped together into inner nodes/buckets to reflect the physical topology. Several devices may be grouped into a host, which is part of a larger rack, which in turn is one of many racks in a room. A room might be part of a data center, and all data centers together make up a whole system. This allows CRUSH to take the physical location and dependencies into account, as these can be the source of concurrent hardware failures and can easily lead to data loss. Ceph gives all storage devices a weight metric that corresponds with their capacity. The more a device can store, the higher its weight, and the higher the likelihood that it is chosen to store an object. A bucket has the combined weight of all its children.

The **pool parameters** are the placement rules that determine how objects in a certain pool (a logical grouping of files with similar attributes or function) are stored and replicated. When working with hundreds of devices, some will inevitably malfunction. These rather sporadic and random failures can be mostly absorbed by replicating files across two or more devices. Larger incidents like a water leak might damage all hosts and their disks in a specific rack, or a cable fire could take out all devices in an entire room. To guard against these concurrent failures, it is paramount to save copies of the data in different places to reduce the chance that natural and man-made disasters lead to the complete loss of data. CRUSH decides how and where a file should be stored with placement rules. These data distribution policies determine how many copies need to be stored where, and for which type of data, on which type of device. A rule that distributes three replicas of a file across three cabinets in the same row can be seen in figure 2.4.

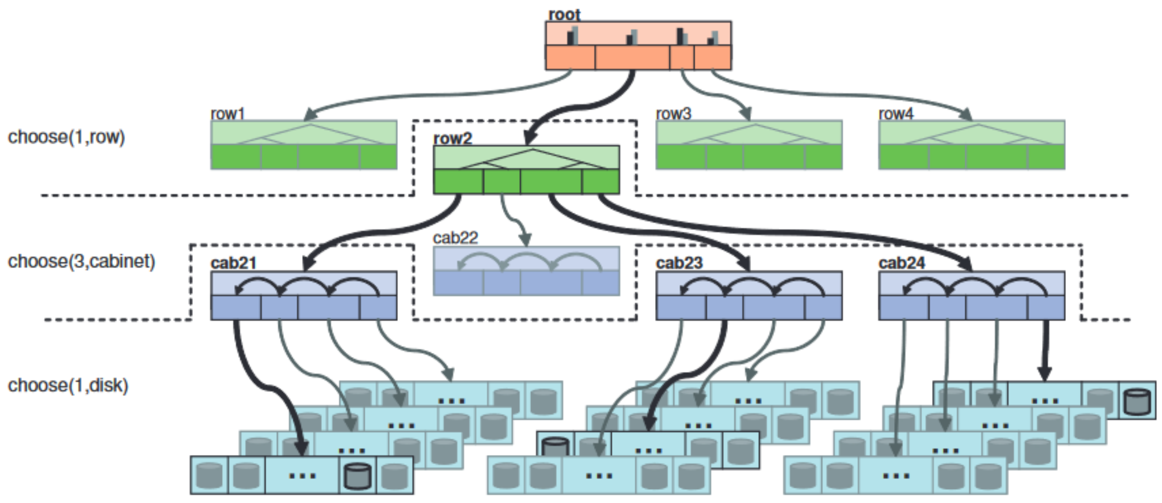


Figure 2.4.: CRUSH placement example from [Weil et al., 2006]

Finally a **placement group** (PG) is a collection of objects in a pool that offers a level of indirection between pools and storage devices to help with data placement, recovery, and balancing. It is much easier to map 100.000 objects to one PG and this PG to several storage devices, than to individually map 100.000 objects to devices. Moreover, the amount and status of physical devices constantly change as new devices are added and defective ones are removed.

It is much easier to just update the perhaps dozens of placement groups affected by a change, instead of potentially millions or billions of objects. For this to work well, it is generally recommended to have between 50 and 100 placement groups per storage device. An object is placed in a placement group by taking a hash of its name and, with a modulo operation, reduce the result to a valid placement group identification number (pg_id = hash(obj_name) % number_of_PGs).

2.1.4. Other Distributed Systems

In this thesis, I have only looked at the aforementioned two systems, as they are quite popular examples that I had access to. But there ar a wide array of similar distributed systems that I will briefly mention now:

One popular system of these is **Hadoop** [Hadoop, nd], which is based on the Hadoop Distributed File System (HDFS) and the MapReduce programming model. It consists of several nodes that store data and can work efficiently in parallel on partial solutions with their local data. A master node manages jobs, tracks their progress, and finally combines the sub-solutions of several nodes into a final one. To do this, it uses Map-Reductions: In the mapping phase it filters and sorts the data to create key-value-pairs of the desired information. These tuples are then reduced by combining/summarising the many small, simple solutions into fewer and fewer big solutions. It is very fault-tolerant as it defends against disk failures by replicating file data across several storage devices, depending on the specification of the user.

An older technology is **GlusterFS** [Gluster, nd] which is quite similar to Lustre: It consists of storage servers called bricks that are grouped together into volumes, where all the servers share a single name space. Files can then be distributed and replicated across several bricks in a volume, ensuring high performance through parallel access by the clients as well as fault tolerance against device failures. To adjust the capacity as needed, it is possible to add more bricks or remove unused devices. It is a rather simple system, but more complex operations can be implemented through translators that sit between the clients and server, to perform operations like access control or data migration.

A third example is **BeeGFS** (Bee Giant File System) [Herold and Breuner, 2018]. It is a rather simple and flexible system consisting of a number of storage servers that save smaller chunks of files for faster parallel access. Metadata servers then manage the metadata of the files such as the location of the many chunks of all the files and who is authorized to access which. Clients get the necessary information for reading or writing a file from a metadata server, but they are not involved in any actual I/O operations to avoid bottlenecks at the metadata servers.

2.2. Visualization

2.2.1. Plotly and Dash

While it is possible to create visuals with most programming languages, doing so is extremely complicated and time consuming, which is why over the years programmers have developed visualization libraries for most programming languages. These can range from a rather low-level libraries that offer rather limited visualization capacities and leave a lot of room for the individual computer scientist to adapt it to their use case. To very high-level ones like Bokeh and Plotly that can take over a lot of busy work, but through their added complexity can be harder to adapt for very specific problems and might work in ways not necessarily intended.

In this thesis, I used Plotly and Dash [Plotly, nd], as these are powerful tools that offer a wide variety of graphical representations and a relatively easy access to helpful features, be it color bars, zoomable graphs, or interactivity between several graphs and menus. The libraries offer an application programming interface (API) where programmers can access a function from the library and give these calls the necessary information. This ranges from what data should go to the x-axis, which to the y-axis, if these elements should be colored and have a certain size, and if so, which color or size for which element. An application build with Dash is also called a Dashboard,

Plotly and Dash are built as JavaScript libraries that can be accessed with Python for easier preparation, as the interactive visualizations are realized through web pages that necessarily utilize HTML, CSS, and JavaScript.

2.2.2. Graphs

With visualization tools like Plotly, it is possible to turn large amounts of data that are difficult for humans to understand into more readable graphs. It might be difficult to see trends or find outliers when looking through thousands of lines of text, while it is much easier to see groups and concentrations in a plot or spot elements that have a distinct color from the rest. For this to work well, it is important to use an appropriate graph for both the type of data that is visualized and the goal one wishes to achieve. [Heer et al., 2010] [Tufte, 1983]

Data can be roughly divided into three types: **categorical**, **numerical**, and **ordinal**:
Categorical data are qualitative instead of quantitative and represent certain categories/groups, be they countries, users, expenditures or cultures. While it is possible to represent them with numbers, they are only another symbol and do not have any mathematical meaning.
Numerical data, on the other hand, are numbers that one can calculate with and order. They are quantitative and often comprised of measurements, like the population of a country, the number of bytes used on a disk, or the price of a stock.
Ordinal data are a mixture of the two: categorical data that has an innate order/hierarchy. This could be the satisfaction level in a survey, where "very happy", "happy", and "miserable" have a clear order from most to least satisfied.

To visualize them, there are many types of graphs that have been developed over the centuries, of which I will present a selection of common ones:

- Bar charts are a good way to visualize mostly categorical and ordinal data, where each category is represented with a bar and the height corresponds to a certain numerical value of that category.
- Pie charts are another way of showing a combination of categorical and numerical data, where the area/degree in a circle shows the portion of a category. Very similar to them are tree maps that show the parentage of a category with the area of a rectangle. They are often preferred over pie charts, as these can become difficult to read when showing to

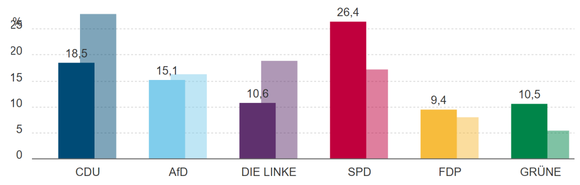
many categories. Degrees in a circle are harder for most humans to estimate and compare than the length of rectangles. Tree map s can also be used to denote a hierarchical relation between elements by creating ensted rectangles.

- Scatter plots are useful for showing the relationship between two numerical values for even large numbers of data points.

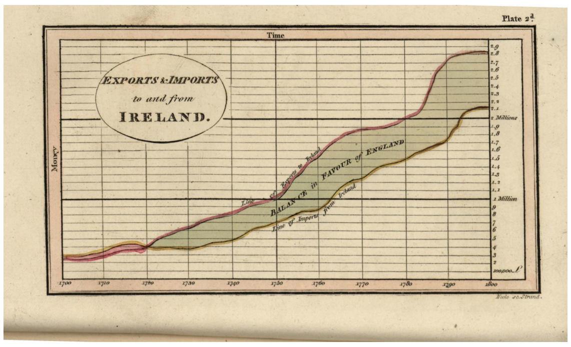
- Histograms are somewhat similar to bar charts but show the frequency of certain numerical ranges, where choosing a good size for these ranges can make or break the visualization.

- Line charts track a value over time and are thus most useful for numerical data that also has a time component, like stock prices.

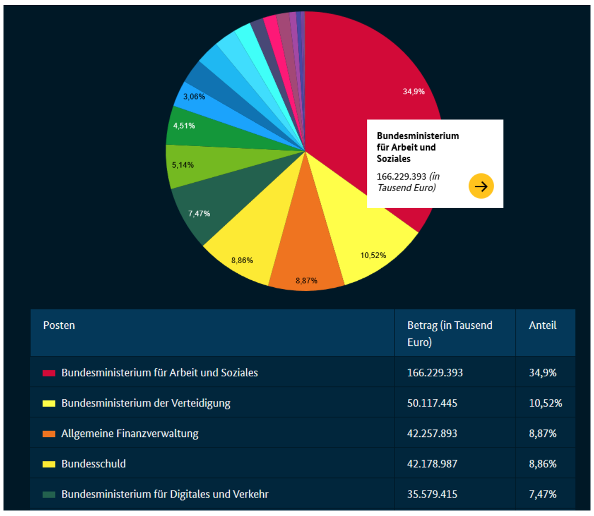
- Finally, there is also a wide variety of map graphs where the geographic location is used to either place other graphs on a map or to change elements of the map itself. It is most useful for categorical data like states or streets, but also for numerical data like longitudes and latitudes.



(a) A bar chart of the results of the 2021 and 2018 Bundestagswahl in Magdeburg from [Der Bundeswahlleiter, 2023]

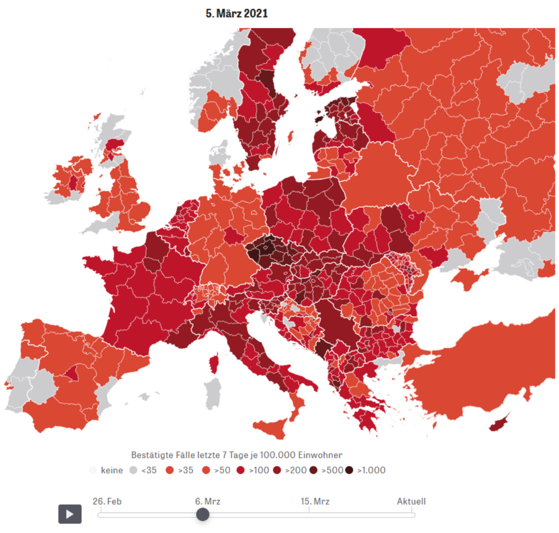


(b) This historical line chart from [Tufte, 1983] uses the space between two lines to show the trade balance between Ireland and England


34,9%
10,52%
8,87%
8,86%
7,47%
5,14%
4,51%
3,06%
Bundesministerium für Arbeit und Soziales
166.229.393 *(in Tausend Euro)*

Posten	Betrag (in Tausend Euro)	Anteil
🟥 Bundesministerium für Arbeit und Soziales	166.229.393	34,9%
🟨 Bundesministerium der Verteidigung	50.117.445	10,52%
🟧 Allgemeine Finanzverwaltung	42.257.893	8,87%
🟨 Bundesschuld	42.178.987	8,86%
🟩 Bundesministerium für Digitales und Verkehr	35.579.415	7,47%

(a) A pie chart that displays the expenditures of the German government from [BMF, 2023] per resort



(b) A colored map of european Corona cases in March 2020 from [Schach et al., 2020]

These and many other charts can be used by themselves, but it is sometimes preferred to add additional dimensions to them to compare more aspects of the data.
One way is to adjust the size of an element. In a scatter plot, this means enlarging the data points with either the area or radius/diameter corresponding to a third numerical value. When working with several pie charts, the size of them could represent the total value of all the slices. Multiple pie charts could show the makeup of a country's exports while the size of the chart shows the total tonnage.

Another popular dimension is color: this might mean coloring categorical data with distinct colors to better differentiate between them, like giving the slices in a pie chart or the points in scatter plot distinct colors corresponding to their category.
In some contexts, color can be associated with a certain category and thus should be used accordingly: most political ideologies and their corresponding parties have colors associated with them, be it red for socialists, black for christian democrats, yellow for libertarians, green for environmentalists, and brown for fascists. Whereas the individual departments of a government rarely have any deep seated connections to specific colors.
But color can also be continuous and represent a numerical value for which many graphs have a color scale that shows the range of a value and the corresponding color. The bar for a storage device might be colored to show its fill level, from blue for empty to red for full, or a state could be colored from white to orange to red and finally to black, depending on the amount of COVID cases. In both cases, the color scale has an easy-to-understand range that is established in our culture. Color scales like the rainbow scale do offer more distinct colors, but they make it harder to judge a color's value.

Other dimensions, like the texture or shape of an element are also possible and can work well, especially for categorical data. But they are more difficult to use and might quickly overwhelm the viewer with visual clutter, which can already be a problem when using just size and color. It is just as important not to show certain information as it is to do, so that the graph can be as simple and easily understandable as possible.

Finally, it is often useful to layer several graphs or graph elements next to each other, allowing viewers to easily compare them. In a bar chart, this can mean grouping several thematically coherent bars together, like election results from different years, or stacking bars on top of each other if they are all subdivisions of a larger whole. In a line chart, layering means showing how several values change over time, for example the imports from and exports to a specific country. To differentiate the other elements, color is often helpful. Though depending on the number of layered elements, the graph can very quickly become illegible.

Summary

In this chapter we looked at the basics of Lustre as well as Ceph and its underlying algorithm CRUSH, learning how they allow for the storage and access of hundreds of Tera- or even petabytes. Then we briefly learned about Plotly and Dash, two of many visualization libraries that create interactive graphs with JavaScript. Last but not least, we looked at some fundamentals of these graphs, what types there are and how they can be used.

Chapter 3.

Design and Implementation

With this background information on the systems and tools we are working with, we can now return to the initial research question and delve into the specifics:
***What** exactly do we want to visualize? Are there shared aspects among the storage layouts of all distributed systems, and what are interesting components unique to Lustre and Ceph?*
*But also **how** can we visualize them? What are the necessary steps to achieve the desired interactive graphs and how did I implement them for the specific systems in question?*

3.1. Design Goals

3.1.1. General

1. A commonality between all distributed systems is their use of storage devices, as regardless of whatever unique data distribution method they utilize, data still need to be stored somewhere. To ensure an even and thus efficient distribution, it is crucial to always have an **overview** of the cluster that shows the **workload of every storage device**.
   Supplementary information about the devices can be invaluable when encountering irregularities, as this can provide valuable insight into potential causes. These might be the type of device (HDD, SSD, etc.) or the date it was added to the cluster, though this in-depth information is not always readily available in all distributed systems.

2. In contrast to the previous high-level view, often a more **focused** analysis of just **a handful or even a single file** is needed, as applications might read from and write to a small collection of files. In those cases, it is important to detect potential inefficiencies or congestions as well as discrepancies between the theoretical data distribution a user intended and the actual result in practice. Moreover, as mentioned in the CRUSH section (2.1.3), concurrent failures can be devastating and lead to permanent data loss. Hence, the physical location of a file and its copies is of great significance. For larger files in particular, an even distribution across multiple storage devices is very beneficial, as it allows the system to leverage the combined bandwidth of several storage devices simultaneously, resulting in faster file retrieval.
   To visualize this, a general goal would be to show the distribution of one or several files across different devices and display the utilized storage. This can be rather difficult or unhelpful when working with distributed systems that are not file-based but instead object-based, like Ceph. These systems do not directly store files but instead split them

into potentially thousands of objects first and only then assign storage devices to them, adding a level of indirection.

3. Finally, in some cases, an **entire folder** is of interest, whether it is a user's personal folder, a project-specific folder, or a shared folder for common resources. Since these directories can contain large amounts of files, it becomes necessary to consolidate the information and illustrate how the files are distributed across the whole storage system to ensure a balanced distribution. In cases where an uneven distribution happens, it can be helpful to have other metrics readily available to quickly identify the source of the issue or provide hints regarding the problem. These might be metrics such as the age, ownership, or size of the files, as these might have a bearing on the maldistribution.

While these are quite generic wishes for the visualization of pretty much all distributed systems, there are also very system-specific ones that rely on their unique characteristics:

3.1.2. Lustre

For Lustre, a notable feature is its (progressive) file striping, which empowers users to explicitly define how and where a file should be stored. A user can specify how many slices are on which Object Storage Target, and even define how various sections of a file should be handled to accommodate file growth considerations.
This hands-on approach is quite powerful and gives the individual user a lot of agency to optimally store files, but also the responsibility to do so correctly. Visualizing where the individual stripes are actually stored could aid in asserting that the defined behaviour aligns with the intended one, as mistakes can easily creep into these layouts.

3.1.3. Ceph

Ceph, on the other hand, does not offer the tools to directly control the placement and replication of individual files. Instead, this is done indirectly by adjusting various variables, such as the number of placement groups for a given pool and placement and replication rules.

Therefore, visualizing the quite interwoven relationship between files, objects, pools, placement groups, and OSDs could greatly help in understanding the impact of these variables and making adjustments to them. Ceph itself provides valuable information about these different components, such as the state of a placement group or the health of an OSD.

3.2. Implementation

To achieve the aforementioned goals, I have implemented the following process that divides the visualization of the storage layout into four steps: The first step involves obtaining the textual outputs from the server, which are then parsed and cut into usable tables, lists, or dictionaries. These refined formats can then be used as the basis for Plotly graphs, which in turn can be grouped and connected into an interactive dash application.

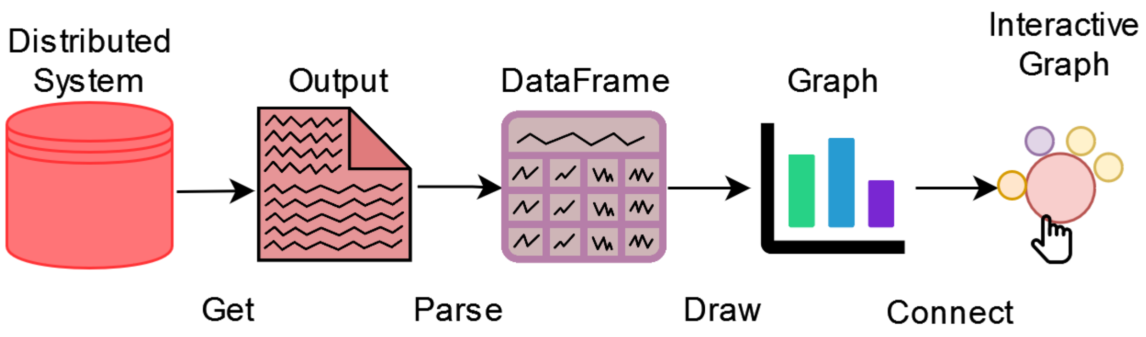


Figure 3.1.: The four steps I implemented to create interactive visualizations out of

3.2.1. Get Information

In many cases, there are simple commands to initially get the raw storage layout information from the distributed system. These can be as short as `"ceph osd tree"` in Ceph or `"lfs df"` in Lustre and provide an overview of all the storage devices. In these cases, it is sufficient to simply send the command to the server and redirect the response to a text file or, alternatively, to a variable to start parsing immediately.
Ceph also offers the command `"ceph pg dump"`, which does not just dump vast amounts of information about every single placement group, it also includes information about every pool and additional information for the OSDs that is not included in `"ceph osd tree"`. This, too, can be simply written to a text file.

But getting information about individual files and especially entire directories can be much more difficult:

Lustre: Gathering the file striping information of a single file can be easily done with the command `"lfs getstripe <file-name>"`. If more than one file is needed, multiple of these commands have to be sent.
But for a whole directory and its subfolders it is necessary to first collect the path of all files in it and then individually send `"getstripe"` commands to the server for every file. In this step, it can also be beneficial to save additional statistics for each file, like the size, owner, and date of last access/modification/change, as these details can sometimes explain unusual file distributions. The size information is also crucial for certain advanced visualizations. However, sending console commands for every file can quickly get out of hand for larger directories, as they consume a significant amount of resources and may overburden the system with excessive requests.

Ceph: Files in Ceph are typically split into numerous objects, which are then pseudo-randomly distributed across placement groups. It is necessary to first get the list of all objects in a pool with `"rados -p <pool-name> ls"`. A pool might contain just a dozen objects, but it can also easily include several hundred million objects. Afterwards, it is possible to find the objects belonging to a desired file and for each object the corresponding PG and OSDs through the command `"ceph osd map <pool-name> <object-name>"`.
This has the same problem as the directory analysis in Lustre, but now for each and every file. Sending thousands of `"osd map"` commands to the Ceph cluster for every single object a file is split into would put the system under immense stress, not to mention actually analyzing an entire directory in Ceph this way.

Regrettably, at least the Ceph system I used was more restrictive than Lustre, and I was unable to execute any of these commands myself. Instead, I had to rely on the administrator of the cluster to execute the commands and provide me with the results. As such, typical users may not have access to any of this fundamental information.

3.2.2. Parse Information

After receiving the raw data from the distributed system, it can be parsed into a format usable by the visualization library, which typically means a list, or a table format like pandas DataFrame. A DataFrame is generally preferred because it allows for querying the created table and selecting rows, columns, or cells depending on any desired conditions, thus being quite fast.
Though for some formats, such as the file striping in Lustre, the data is already in a dictionary-like format. In these cases, it can be more efficient in terms of speed and space to retain the dictionary layout and save it as, for example, a JSON file. While it is possible to reconstruct this striping data into tables, doing so would result in significantly larger tables with a lot of redundancies, which may not be desirable.

With that being said, here are the different ways I parsed the four output text files:

Lustre (df) :

```
UUID                    1K-blocks        Used   Available Use% Mounted on
home-MDT0000_UUID      1913028680    27181984  1724785628   2% /home[MDT:0]
home-MDT0001_UUID      1274014400    23198716  1143443144   2% /home[MDT:1]
home-MDT0002_UUID      1274014400    19334548  1147307312   2% /home[MDT:2]
home-MDT0003_UUID      1274014400    18176856  1148461364   2% /home[MDT:3]
home-OST0000_UUID     31118373528  8971718260 20562437828  31% /home[OST:0]
home-OST0001_UUID     31118373528  8943859432 20590728104  31% /home[OST:1]
home-OST0002_UUID     31118373528 10565464240 18968972084  36% /home[OST:2]
home-OST0003_UUID     31118373528 11108430144 18426154344  38% /home[OST:3]

filesystem_summary:  124473494112 39589472076 78548292360  34% /home
```

Figure 3.2.: Example output of the df command that lists the utilization and capacities of all MDTs and OSTs in the /home partition

The first command to parse is the "df" command, which displays the use of every MDT and OST in the system in a simple table-like format. This allows me to simply recreate the table by going through each line in the textual output, splitting it and storing each device property in a corresponding column.
In some systems, like the one I worked with, there is not just one table, but instead several for different partitions like /home, /work, or /fastdata. Confusingly, a single MDT or OST can belong to several of these at the same time, though its storage information remains the same. As such, it would be possible to either discard duplicate targets or entire partitions. I decided to simply store these, mostly redundant, copies as well, in case the exact partition of an OST ever becomes necessary in the future.

Lustre No PFL (getstripe):

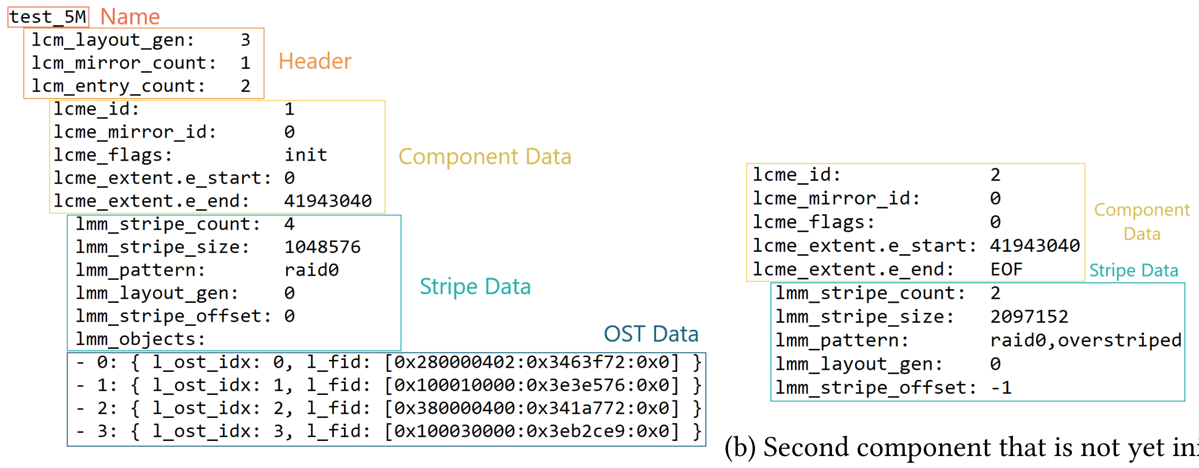
Nearly as simple is the striping layout of most files: To parse the header, so the first few lines with general information about the file striping, it is enough to split each line and store these

```
df.csv
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_pattern:        raid0
lmm_layout_gen:     0
lmm_stripe_offset: 1
        obdidx           objid           objid           group
             1        65267030       0x3e3e556               0
```

Figure 3.3.: Example output of the `getstripe` command that shows the layout of a single file without PFL

key-value pairs in a dictionary until hitting the OST-table. There, each line is then added as a dictionary to a list of all used OSTs. Parts of this information remain unused in this thesis, like the layout_gen, pattern, or stripe offset, but are still kept in the parsed result for convenience and possible future visualizations that might require them.

Lustre PFL (getstripe):



```
test_5M
  lcm_layout_gen:     3
  lcm_mirror_count:   1
  lcm_entry_count:    2
    lcme_id:             1
    lcme_mirror_id:      0
    lcme_flags:          init
    lcme_extent.e_start: 0
    lcme_extent.e_end:   41943040
      lmm_stripe_count:  4
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: 0
      lmm_objects:
      - 0: { l_ost_idx: 0, l_fid: [0x280000402:0x3463f72:0x0] }
      - 1: { l_ost_idx: 1, l_fid: [0x100010000:0x3e3e576:0x0] }
      - 2: { l_ost_idx: 2, l_fid: [0x380000400:0x341a772:0x0] }
      - 3: { l_ost_idx: 3, l_fid: [0x100030000:0x3eb2ce9:0x0] }
```

```
lcme_id:             2
lcme_mirror_id:      0
lcme_flags:          0
lcme_extent.e_start: 41943040
lcme_extent.e_end:   EOF
  lmm_stripe_count:  2
  lmm_stripe_size:   2097152
  lmm_pattern:       raid0,overstriped
  lmm_layout_gen:    0
  lmm_stripe_offset: -1
```

(a) Header and first component that is in use

(b) Second component that is not yet initialized and thus has no OSD data

Figure 3.4.: Exemplary progressive file layout with an active and an inactive component, the different parts are outlined

Files with a progressive file layout consist of several blocks: a first "header" block with three lines of general information about the file layout, and then as many blocks as there are components to the PFL. These blocks are simple key-value pairs that can be stored into dictionaries just like the non-PFL files but also contain additional information about the component. The "lmm_objects" show the OSTs this part of the file is stored on, though in a dictionary format, instead of the table format the standard files are stored in. Each OST-dictionary is added to a list, and all individual component-dictionaries are then also grouped in a list, creating a nested structure. These different parts can be seen in figure 3.4a.
However, not every component of the layout has to be active (marked by an "init" flag). If a file is small enough, then some components are not yet active and have no OST information, like in figure 3.4b.

Lustre Directory (getstripe):

The same procedure can also be done for a whole directory and all its sub-folders. As mentioned in the previous section, this requires a list of all files contained in a directory and separate "`getstripe`" commands. Each parsed file is then added to a list, creating a single large JSON file.

Ceph (osd tree):

```
ID   CLASS  WEIGHT     TYPE NAME                 STATUS  REWEIGHT  PRI-AFF
 -1         220.21967  root default
-20           5.23975      room 035
-25                 0          row dell
-40                 0              host hulk
 -5                 0              host marvel
-81                 0              host node1
-49                 0              host node2
-37                 0              host node3
-73                 0              host node4
-46                 0              host node5
 -9           5.23975          host jarvis
 54    ssd    0.87329              osd.54            up   1.00000  1.00000
 55    ssd    0.87329              osd.55            up   1.00000  1.00000
 56    ssd    0.87329              osd.56            up   1.00000  1.00000
 57    ssd    0.87329              osd.57            up   1.00000  1.00000
 84    ssd    0.87329              osd.84            up   1.00000  1.00000
 85    ssd    0.87329              osd.85            up   1.00000  1.00000
-85          85.58130      room 242
-11          43.66296          host mark3
 60   nvme    1.81929              osd.60            up   1.00000  1.00000
 61   nvme    1.81929              osd.61            up   1.00000  1.00000
```

Figure 3.5.: OSD Tree

In contrast to the rather simple procedures before, recreating the storage hierarchy from a Ceph tree is quite a bit more complex. To visualize it as a hierarchical graph like a tree map, it is necessary to have a list of all components and a second list with their respective parents. For this I created the following algorithm:

```python
tree = open("tree.txt")

# starting with the root, initialize the outputs and helper variables
root = tree.pop().split()
names, parents = [root[2]], [""]
hierarchy, weight = [root[2]], [root[1]]

for line in tree:
    attributes = line.split()
    # if only 4 columns => bucket (ID, Weight, Type, Name)
    if len(attributes) == 4:
        # no weight => bucket is already deleted, skip
        if attributes[1] == 0: continue

        names.append(attributes[3])
        # remove completely read buckets
        while weights[-1] < 0.1:
            hierachy.pop(-1)
            weights.pop(-1)

        # add the parent and adjust its weight, add new bucket to
            ↪ helper lists
        parents.append(hierarchy[-1])
        hierarchy.append(attribute[3])
        weights[-1] -= attributes[1]
        weights.append(attributes[1])

    # otherwise => leaf (ID, Class, Weight, Type, Name, Status,
       ↪ Reweight, Pri-Afff))
    else:
        names.append(attributes[3])
        parents.append(hierarchy[-1])
        weights[-1]-= attributes[2]

```

Listing 3.1: OSD Tree Parsing Algorithm

The first step is to open the text document and initialize the output lists for both the names of the leafs or buckets, and their respective parents, as well as the helper lists "hierarchy" and "weights". They are necessary to keep track of previous buckets and the amount of weight that was "used up" when descending the tree.
Each output and helper list is immediately filled with the root information of the first line, as the root bucket has no parent and is an exception that needs to be handled separately.

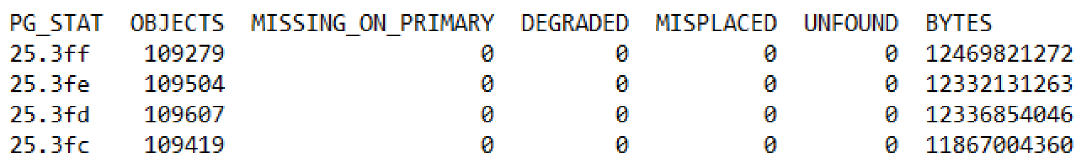
From this point on, the algorithm goes through each line sequentially. If it is a bucket, recognizable by a smaller number of attributes, it is crucial to first check its weight. Deleted buckets have a weight of zero and can remain in the tree for some time, so they have to be skipped. Otherwise, its name is added to the "names" list.
Then any buckets in the hierarchy that were completely processed are removed from both helper lists. As mentioned in chapter 2.1.3, each OSD has a weight according to its storage capabilities. A bucket's weight is therefore the sum of all its children, be they OSDs or other buckets. By subtracting the weight of each child as they are being read from the parent's weight, it can be assured that a bucket has been fully read when its weight reaches zero. Or at least close to zero, as floating point arithmetic means that rounding errors exist and a bucket's weight is not *exactly* the sum of its children.
After the helper lists were cleaned, the parent is the latest bucket in the hierarchy list and can simply be added to the parent list. Afterwards, both the name of the current bucket and its weight can be appended to the helper lists to correctly assign its children to it while subtracting its weight from its parent.

If the line is not a bucket, it is instead an OSD whose name can simply be added to the name list, with the last added bucket in the hierarchy as their parent. Similar to the buckets, the weight of this OSD is also subtracted from the last entry in the weight list, in order to correctly assign OSDs to their correct bucket.

The end results are two lists, one for the name of each element and one for their respective parent, which can later be used by Plotly.

Ceph (pg dump): The command "ceph pg dump" is a powerful tool, showing not just extensive placement group data, but also various additional information about the OSDs the placement groups are stored on and the pools they belong to.

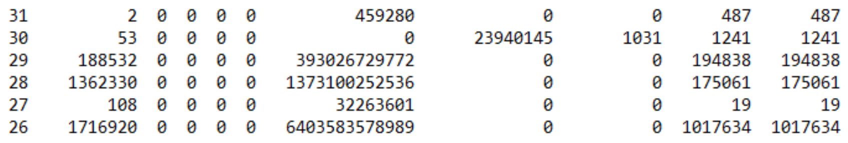
The first part consists of a large table containing all the placement groups for each pool, shown in figure 3.6. Parsing this table is very similar to the Lustre `"df"` output, as it is enough to



```
PG_STAT  OBJECTS  MISSING_ON_PRIMARY  DEGRADED  MISPLACED  UNFOUND  BYTES
25.3ff    109279                   0         0          0        0  12469821272
25.3fe    109504                   0         0          0        0  12332131263
25.3fd    109607                   0         0          0        0  12336854046
25.3fc    109419                   0         0          0        0  11867004360
```

Figure 3.6.: A fraction of the total placement group data

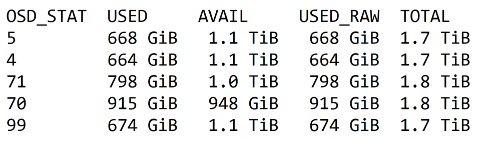
simply iterate through each entry and extract the relevant information to a DataFrame. I only utilize a fraction of the available information because much of the table is either uninteresting in regards to the storage layout or adds redundancy in case of errors.
In a healthy cluster, some columns contain just zeros and are only interesting in the case of errors. They can be important to note, but are rather uneventful in normal situations. Moreover, certain columns are rather cryptic and lack explanations in the documentation.



```
31        2  0  0  0  0         459280         0       0      487      487
30       53  0  0  0  0              0  23940145    1031     1241     1241
29   188532  0  0  0  0   393026729772         0       0   194838   194838
28  1362330  0  0  0  0  1373100252536         0       0   175061   175061
27      108  0  0  0  0       32263601         0       0       19       19
26  1716920  0  0  0  0  6403583578989         0       0  1017634  1017634
```

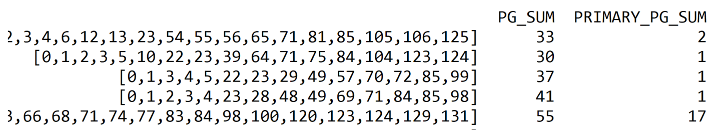
Figure 3.7.: Pool summary

Shown in figure 3.7 is the second part, which provides a summary of some of the above placement group information, but this time for each pool these PGs are in. It shows how many bytes and objects a pool has in total, how many of its PGs have any errors, and more. This pool summary is also stored line by line in another data frame.



```
OSD_STAT  USED     AVAIL     USED_RAW  TOTAL
5         668 GiB   1.1 TiB   668 GiB  1.7 TiB
4         664 GiB   1.1 TiB   664 GiB  1.7 TiB
71        798 GiB   1.0 TiB   798 GiB  1.8 TiB
70        915 GiB   948 GiB   915 GiB  1.8 TiB
99        674 GiB   1.1 TiB   674 GiB  1.7 TiB
```

(a) First five columns of OSD information



```
                                                       PG_SUM  PRIMARY_PG_SUM
2,3,4,6,12,13,23,54,55,56,65,71,81,85,105,106,125]        33               2
    [0,1,2,3,5,10,22,23,39,64,71,75,84,104,123,124]       30               1
             [0,1,3,4,5,22,23,29,49,57,70,72,85,99]       37               1
             [0,1,2,3,4,23,28,48,49,69,71,84,85,98]       41               1
3,66,68,71,74,77,83,84,98,100,120,123,124,129,131]        55              17
```

(b) other 3 columns, with the heartbeat peers on the left

Figure 3.8.: First five entries in the OSD part, the heartbeat peers are shortened for brevity's sake

Last but not least, there is a third table about the object storage daemons. While the "tree" command provides the type and location in the cluster map for the OSD, this table goes beyond that by providing additional information. It shows the total, available, and used bytes, as well as the heartbeat peers. These OSDs have established a peer-to-peer contact, enabling them to share the state of the system, monitor each other's state, and "hear each others heartbeat", so to speak. The table also displays the number of PGs saved on this OSD and how many of those are primary placement groups instead of replicated copies.
This, just like the other tables, is also simply read line by line, recreating the columns to create a DataFrame out of it.

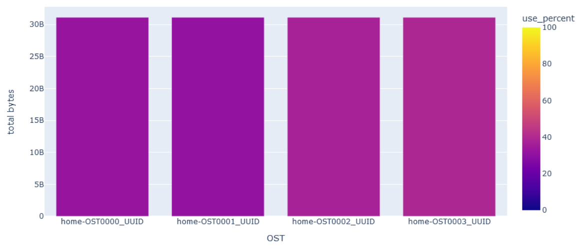
3.2.3. Visualize Information

Once the raw data has been parsed and transformed into a usable format, it is then possible to use a visualization library such as Plotly to create diverse graphical representations of the data.

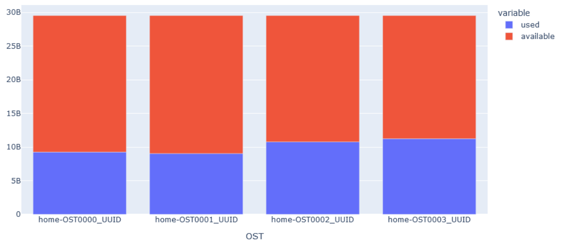
Plotly provides two options for visualizations: a "default" version that allows users to choose one of over 20 basic graphs and customize it as needed, and an "express" version that is already customized for most common use-cases. Both take the form of functions that need either the columns of a DataFrame or a list for each data dimension to generate graphs. For example, the function to create a bar chart might look like this:

```
figure = plotly.express.bar(DataFrame, x="OST-ID", y="bytes-Used", color="Component")
figure.show()
```

For the **general storage overview**, a bar chart is utilized where each bar represents a storage device and their height the total size. They are colored according to their utilization (used bytes divided by total bytes), which allows users to quickly spot outliers. The total size is particularly important for systems with a varied collection of storage devices. The variety can easily lead to an uneven distribution, such as smaller SSDs overfilling quicker than larger hard disks.
Another approach involves dividing the bar into two sections, a lower bar representing used storage and an upper bar representing the remaining available storage. This design leverages skeuomorphism by mimicking the fill-levels of a glass, leading to more intuitive comprehension. But comparing fill-percentages becomes more difficult in very heterogeneous storage environments due to the varying device sizes, as it requires more attention to compare the height of four bars rather then simply discerning two colors.



(a) Storage device overview, color for fill level



(b) Storage overview, blue for used bytes, red for available bytes

Figure 3.9.: Storage device overview examples

Lustre For visualizing the striping layout of a **single file** I choose a simple stacked bar chart. This graph illustrates how much of the file is on which storage device and, when a progressive file layout is applied, how each component is distributed by differentiating them by color. While it would be feasible to show each individual stripe as a separate stack in the bar chart, these slices typically have a fixed size of 1MB. As a file might have 1GB on a single OST, cutting the bar into 100 pieces makes it very hard to read and render, with very little relevant information gained.
Unlike most other graphs, where the data can be taken directly from the columns of a DataFrame, for the Lustre file striping data, it is actually necessary to compute the location. For a "normal" file this process is relatively straight forward. By accessing the file's size with "`stat(<file-path>)`", it can be divided among the OSTs it is stored on. Files with a progressive file layout consist of several components that all need to be processed similarly.
Multiple file distributions are visualized in much the same way, but in this case, the color indicates the file the slices belong to, and not the individual component.

To show how multiple files **overlap** on certain OSTs, especially for I/O considerations, a 2-D heat map is used. This visualization requires more preparation, as it is necessary to do the

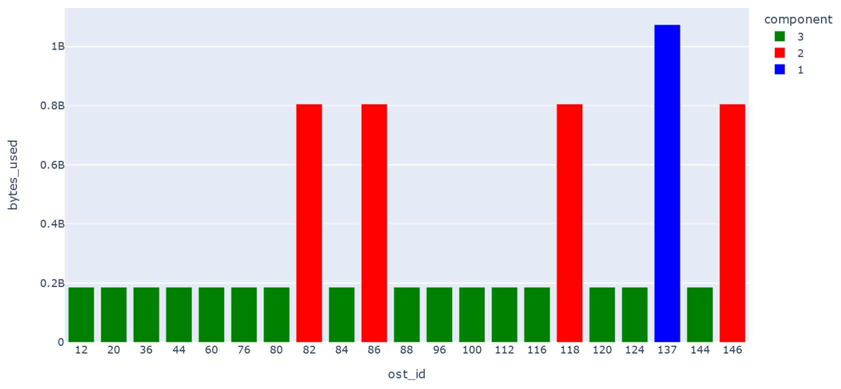


Figure 3.10.: Single file distribution example

above procedure of getting the individual storage distribution for all files in each group. Then the distribution between the two groups can be compared, storing the overlap for each OST. This provides a good metric for measuring potential congestion between two groups of files.

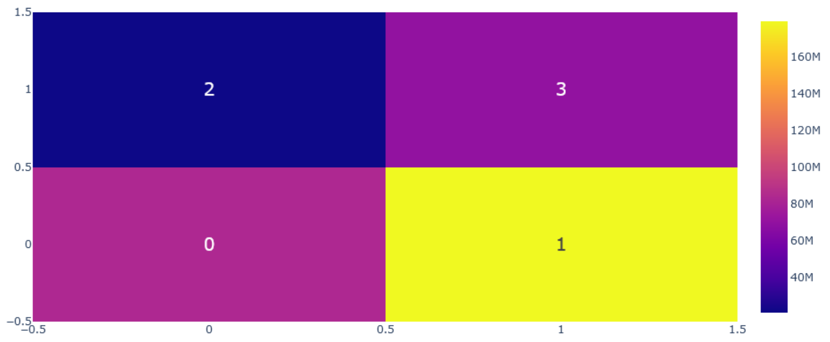


Figure 3.11.: Heat map of a pair of three files, that overlap the most on OST 1

To visualize the **additional statistics** of a directory, scatter plots are used for the *three time measurements* (*atime, ctime, mtime*), showing the amount of files last accessed, changed, or modified on any given day. This approach aggregates the data and is effective even on thousands of files spanning decades.
The *ownership* is shown through pie charts, as they are sufficient for a quick overview to see which users or groups own most of the files. While a pie chart does not support in-depth analyses, such investigations are seldom necessary in this case. For the *size* a histogram is used, to show how many files lie in a certain byte range. The exact number for each data point is less important than the overall picture to spot larger trends. A histogram excels at aggregating massive amounts of data into a digestible size, which is important for very large directories.

Ceph: For the Ceph storage overview, a bar chart is utilized similar to Lustre, but in addition to this, a tree map. This allows for an understanding of the physical storage layout, meaning which OSDs are in the same host/row/room as others. That data also theoretically exists in Lustre and any other distributed systems. They distribute their storage devices, so it is always interesting to know where they are and which devices are next to each other, but it is not always as easy to access and central to the entire system as in Ceph.

For the placement groups, I use a scatter plot, with the size in bytes as the x-axis and the number of contained object as the y-axis. This creates bands or clouds that almost automatically sort

PGs into pools. The number of placement groups per pool defines how many objects each PG on average has, while the size can vary wildly depending on the specific objects that happen to end up in the PG. The color of the scatter plot represents the corresponding pool, though another option would be to color them green, yellow, or red depending on the state of the PG, as monitoring the health of a placement group is also crucial. For the pools, I have opted for a bar chart, as this allows any interesting available information to be on the y-axis, with each pool on the x-axis. Although using color to incorporate more dimensions would be possible, I decided against it to keep the graph as simple and readable as possible.

3.2.4. Connect Information

Though these individual graphs are already sufficient for some tasks and allow for a better understanding of the storage layout, adding additional interactivity between them allows us to explore the data even more.

Plotly has the advantage that it already offers a limited amount of interaction in a graph out of the gate, as it is possible in most graphs to zoom in, pan across, and hover over specific data points for more additional information. This can be enhanced by explicitly defining the parts that should be shown.

Using Dash, it is possible to define a HTML layout of boxes, place graphs or menus in them, and define their interactive behaviour through callbacks, completely changing which data graphs display and how they do so.

As this level of complexity is sometimes neither necessary nor helpful, I decided to only create Dashboards for the Ceph overview and Lustre directory distribution. The other singular graphs are arranged in notebooks, which allow for a more "user-controlled" interactivity where one can freely change code and create the connectivity by hand, which is more difficult and hands-on than a Dashboard, but also provides more flexibility and allows users to more freely add or change the existing tools to their needs.

Lustre directory overview: To improve the directory visualization for Lustre, I combine the main distribution over the OSTs with the additional statistics with three graphs: one for the three times, one for both the group and user ownership, and finally one for the size. As the first two have several versions of basically the same information, I use buttons to allow users to switch between a-, c-, and m-time, as well as between the user and group ownership, as showing every statistic at the same time would necessitate very small graphs that are no longer readable.
The user can then change for which files they want to view the statistics by selecting a group of bars, so OSTs, in the main graph. With this selection, the program queries all the stored file layouts for the selected storage devices and returns the aforementioned statistics graphs with the updated data. Users can easily select OSTs that fall outside of the general pattern and see how the files there compare to the rest. Are they newer, or perhaps a lot smaller or larger than the rest? Were they added by a different person than the others? All these potential clues can be quickly examined with this Dashboard.

Ceph overview: For Ceph I chose to connect the placement groups, pools, and storage devices. To better customize the graphs, each has a drop-down menu that allows the user to change the value displayed on the y-axis for the two bar charts of the OSDs and pools, and the color of the scatter plot for the PGs.

The largest graph is a tree map for the OSDs that initially shows the device type of each OSD, but when clicking on a bar from the OSD bar chart, it shows where its heartbeat peers are, to show how the OSDs themselves are connected to each other. Initially, I aimed for a network graph to show the connections, but as each OSD can easily have dozens of peers, this network was essentially unreadable. When clicking on a placement group, the tree map shows the primary and secondary OSDs it is stored and replicated on. The hierarchical representation from the tree map allows for a quick assessment of the distribution, and whether it is spread around enough to avoid most concurrent failures.

Creating this interactivity is the most difficult step, as especially multidirectional connections like those in the Ceph overview have the potential for unintended behaviour. Some graphs have built-in interactivity with themselves, which leads to certain callbacks breaking them. This can happen with tree maps, where clicking on a node zooms in on it. Using a click-events to change the whole graph creates both the new updated graph and the zoomed in view of the clicked node, rendering both unusable. This problem prevents some more intuitive interactivity and necessitates ugly workarounds, like needing a separate OSD bar chart to change the OSD tree map.

Summary

In this chapter we have seen the visualization goals are: For a generic distributed system an overview over the storage device utilization and file or folder distribution. For Lustre specifically it is the file striping over the selected OSTs and for Ceph the relationship between PGs, pools, and OSDs.
To achieve these the process I implemented can be split into four steps: First the raw textual data has to be retrieved through various commands, then parsed into usable lists or tables. These can be visualized into specific graphs by Plotly and in the end be connected into an interactive Dashboard.

Chapter 4.

Related Works

In this chapter I will showcase a selection of similar projects that other people worked on and explain how they differ from what I have done. They range from more general works on storage topologies, to specific Dashboards for Ceph and Lustre.

4.1. Interactive Analysis of Large Distributed Systems with Scalable Topology-based Visualization

One of the few scientific papers on the visualization of distributed systems was written by Lucas M. Schnorr, Arnaud Legrand, and Jean-Marc Vincent in 2013, called "Interactive Analysis of Large Distributed Systems with Scalable Topology-based Visualization" [Schnorr et al., 2013]. It presents a scalable approach for analyzing the topology of a distributed system, but unlike this thesis, it focuses on the transmission of data across the network instead of its storage.

It uses a very reduced set of visualization tools: monitored entities are represented with nodes, edges between them indicate a relationship between the two objects. Typically, hosts are shown as squares and links between them as diamonds, with their size being defined by the available resource capacity. These forms are mostly white but filled with another color, like black, to a certain percentage according to their utilization.

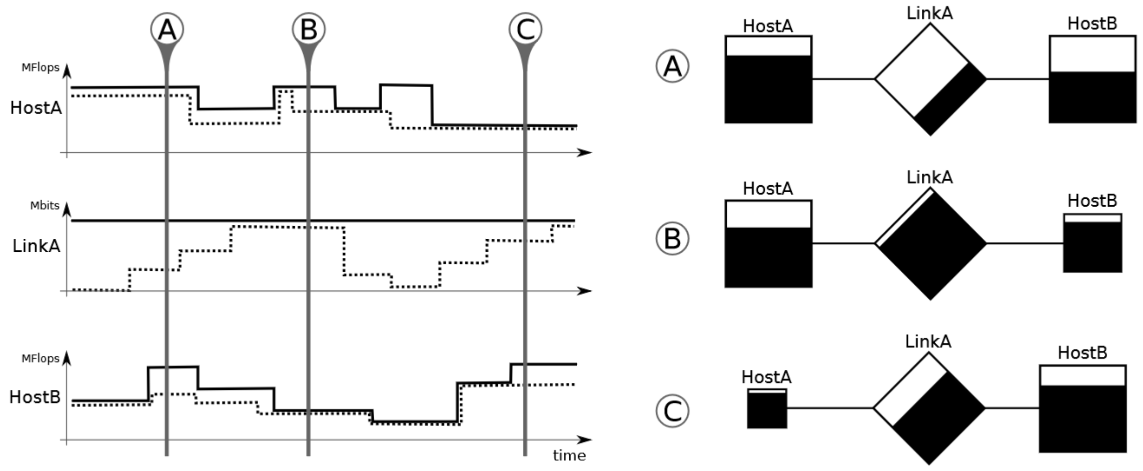


Figure 4.1.: Exemplary network analysis at three different timestamps from [Schnorr et al., 2013]

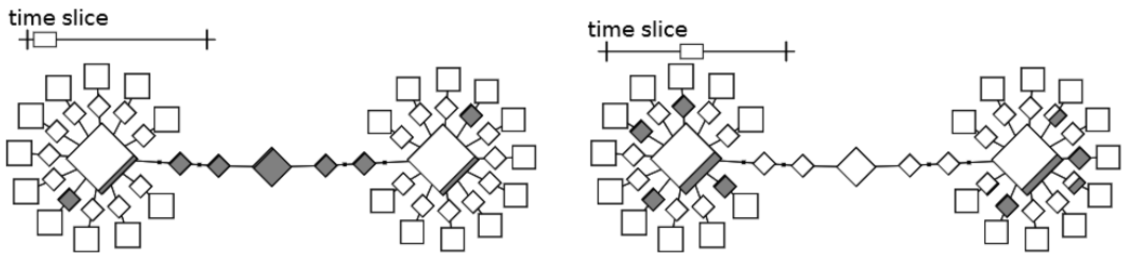


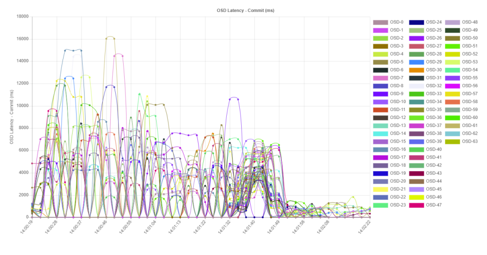
Figure 4.2.: A more complex example for a connection between networks from [Schnorr et al., 2013]

The rather limited nature of this visualization approach means that even complex topologies can be easily understood and congestions or bottlenecks identified. It also allows users to change the scaling between the data and their graphical representations, as well as define their position and distances when needed.

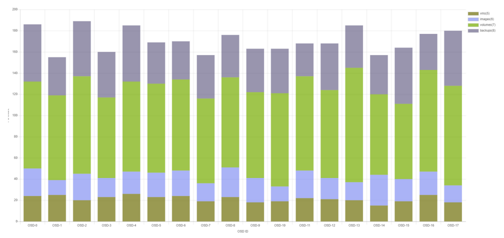
In conclusion, while the visualizations proposed by the paper also revolve around distributed systems and allow for some interactivity (defining the analyzed time slice), they do not concern themselves with the storage layout but instead with the bandwidth necessary for the transmission of the data. They also restrict themselves to a rather minimalist representation and a single graph type. This thesis on the other hand, tries to capture many different types of data with an appropriate graph type and puts a bigger focus on color.

4.2. sonar4ceph

A purely Ceph-based work is "sonar4ceph" [JungJungIn, n d], that allows users to track and show a variety of information about the relationship between placement groups, object storage daemons, and pools, their respective performance, latency, as well as bandwidth utilization. Similar to the work in this thesis, it shows the hierarchy of the OSDs with a tree map, though it also includes the used and available space, as well as warnings about the health of the cluster. Other graphs focus on the relationship between placement groups, pools, and OSDs, though always in singular graphs without any interactivity between them.



(a) Line chart for the latency of each OSD



(b) Stacked bar chart showing the number of PGs per pool and OSD

Although the graphs are aesthetically pleasing and professional, the readability can on occasion be poor, especially when too many entities are drawn and colored over each other, like what happened to the line chart for OSD latency in figure 4.3a. Other diagrams find clever ways to display many interconnected metrics together. Figure 4.3b shows the relationship between OSDs, pools, and placement groups through a single stacked bar chart mostly well, but makes it difficult to compare the PG count of especially the two top most pools between OSDs as the bars constantly change their position.

Of the related works, it is perhaps the one closest to this thesis, though many parts of it focus more on the performance of the cluster than strictly its storage. The visualizations do not offer any interactivity between them. Instead, many different metrics are combined into single (individual) graphs with varying degrees of success.

4.3. Ceph Dashboard

The built-in web-based Ceph Dashboard [Ceph-Foundation, nd] shows the overall state and health of a Ceph cluster, including but not limited to information about the OSDs, Pools, PGs, latency, and throughput. It is a mix of text-based status windows and graphical summaries to assert the health of the cluster and to quickly notice damages or problems. Besides the storage, it also allows administrators to have a look at the monitors and managers of their cluster.

As it is part of Ceph as a Manager Daemon module, it has access to every piece of information in Ceph. In terms of the amount of data shown, it is superior to this thesis, where only a few outputs were available and could be used for parsing and visualizing data. But in contrast, the Ceph Dashboard displays no interactivity to connect the various graphs and puts a heavy focus on performance instead of storage layout. As the name implies, it is also explicitly built only for Ceph and does not work on any other distributed system.

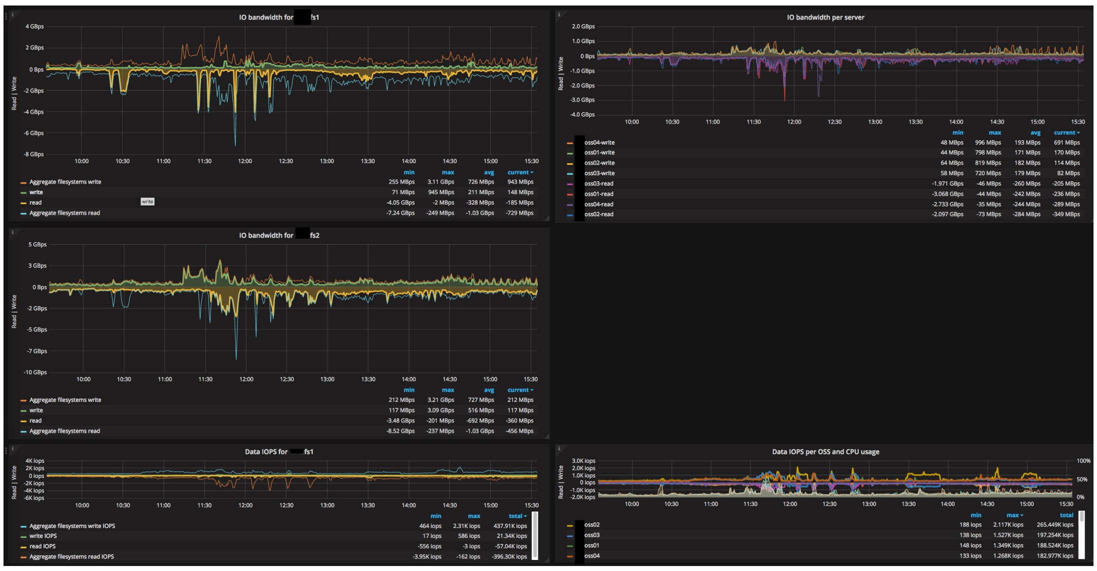
4.4. Lustre Overview Panel from Grafana Labs

Although there are several projects that focus on visualizing the storage layout of Ceph, the same cannot be said about Lustre. While some institutions and companies develop their own visualization tools, little public information is available regarding them. One of the few works regarding Lustre is an overview panel from Grafana Labs [Grafana, 2019].

This dashboard is more directed towards administrators who want a real-time overview of their Lustre system, with a focus on pending (metadata) operations, IO bandwidth, and job statistics. It is not concerned about the storage layout of the data, like this thesis, but instead with the many read and write operations on the stored data, making it somewhat of the flipside of this work.

Summary

As we have seen in this chapter, while there are some previous works on the visualization of distributed systems, most are either more focused on the transmission of data than the storage. Even the ones that do so often lack in depth in this department and do not offer any interactivity for a deeper understanding.



IO bandwidth for [redacted] fs1

	min	max	avg	current ▾
Aggregate filesystems write	255 MBps	3.11 GBps	726 MBps	943 MBps
write	71 MBps	945 MBps	211 MBps	148 MBps
read	-4.05 GBps	-2 MBps	-328 MBps	-185 MBps
Aggregate filesystems read	-7.24 GBps	-249 MBps	-1.03 GBps	-729 MBps

IO bandwidth per server

	min	max	avg	current ▾
[redacted] oss04-write	48 MBps	996 MBps	193 MBps	691 MBps
[redacted] oss01-write	44 MBps	798 MBps	171 MBps	170 MBps
[redacted] oss02-write	64 MBps	819 MBps	182 MBps	114 MBps
[redacted] oss03-write	58 MBps	720 MBps	179 MBps	82 MBps
[redacted] oss03-read	-1.971 GBps	-46 MBps	-260 MBps	-205 MBps
[redacted] oss01-read	-3.068 GBps	-44 MBps	-242 MBps	-236 MBps
[redacted] oss04-read	-2.733 GBps	-35 MBps	-244 MBps	-289 MBps
[redacted] oss02-read	-2.097 GBps	-73 MBps	-284 MBps	-349 MBps

IO bandwidth for [redacted] fs2

	min	max	avg	current ▾
Aggregate filesystems write	212 MBps	3.21 GBps	727 MBps	212 MBps
write	117 MBps	3.09 GBps	516 MBps	117 MBps
read	-3.48 GBps	-201 MBps	-692 MBps	-360 MBps
Aggregate filesystems read	-8.52 GBps	-237 MBps	-1.03 GBps	-456 MBps

Data IOPS for [redacted] fs1

	min	max	total ▾
Aggregate filesystems write IOPS	464 iops	2.31K iops	437.91K iops
write IOPS	17 iops	586 iops	21.34K iops
read IOPS	-556 iops	-3 iops	-57.04K iops
Aggregate filesystems read IOPS	-3.95K iops	-162 iops	-396.30K iops

Data IOPS per OSS and CPU usage

	min	max ▾	total
[redacted] oss02	188 iops	2.117K iops	265.449K iops
[redacted] oss03	138 iops	1.527K iops	197.254K iops
[redacted] oss01	148 iops	1.349K iops	188.524K iops
[redacted] oss04	133 iops	1.268K iops	182.977K iops

Figure 4.4.: Part of the Grafana Dashboard showing various IO-bandwidth and Data IOPS metrics

Chapter 5.

Evaluation

We have seen why visualizing the storage layout of distributed systems matters, how both the distributed systems and visualizing works, as well as how the tools I designed work to achieve this goal and how they compare to existing ones. Now we are going to evaluate these tools, seeing how they function. But before any of that, we first have to familiarize ourselves with the distributed system on which I conducted my tests:

5.1. Test Environments

5.1.1. Levante-DKRZ (Lustre)

The High Performance Computing (HPC) system "Levante" [DKRZ, nd] is an integral part of the Deutsches Klimarechenzentrum (DKRZ) in Hamburg and was recently installed between 2021 and 2022. As the name implies, it is mainly used for various climate simulations, with its most important being its contributions to the World Meteorological Organizations "Intergovernmental Panel on Climate Change" report. It is also employed on dozens of other research topics, whether these are paleoclimate models for improved isotope analysis in the alps, the interactions between agriculture methods and the climate, or future climate change predictions on the Arctic ice.

Levante has over 370.000 cores in 2832 compute nodes, made up mostly of AMD 7763 processors of varying sizes with up to 14 Petaflops of peak performance. It also offers a total of 130 petabytes of storage. The file system is organized into different partitions, as seen in the "df" paragraph of section 3.2.2. Users have access to three partitions: a fast HOME file system where each user can store important files such as shell scripts or source code. A WORK file system to store the data and various model output that a project uses and produces, as well as a SCRATCH file system that allows for the temporary storage of large amounts of data, but only for up to 2 weeks. Other partitions are POOL/DATA for common computing resources, as well as FASTDATA, and SW, which are more restrictive.

5.1.2. Ants-OVGU (Ceph)

Ants is the HPC cluster at the Otto-von-Guericke-University Magdeburg (OVGU) for university scientific use, mostly for parallelized applications with a high network and memory demand. It is made up of 26 AMD Epyc processors of a variety of specifications, and 6 Intel Xenon processors, totaling 700 cores. These are assigned to nine partitions for various groups to use. Across three rooms Ants has 136 OSDs that offer over 200 terabytes of storage.

5.2. Test Results

In the following, I will go over the various visualizations I created and the different use-cases they may or may not be adequate for.

5.2.1. Storage Layout Overview

General Overview: The first step to any analysis of the storage layout is perhaps to get an overall view of the storage devices, which works very similarly for all distributed systems and already provides an early warning for potential problems. As mentioned in section 3.2.3 I tested two versions of displaying the storage utilization: In figure 5.1 and 5.2 the utilization is denoted with a color scale, which works reasonably well.
Plotly allows for a wide selection of color scales, of which I prefer one called "thermal". It is relatively intuitive and is used by some thermal image cameras that users might have experience with. Most importantly in this case are the wide range of colors that allow users to differentiate even slight differences in the storage device usage. I also experimented with a temperature scale (blue-red) that has the advantage of being very intuitive, as most people have seen this color scale in weather reports or thermometers. But as it is only comprised of two colors, slight differences, like in figure 5.1, are difficult to spot.
Figure 5.3 shows the amount of used bytes in blue and the available amount in red above it, which together make up the total size of this storage device. In theory at least. In practice, this can vary a substantial amount, especially in Ceph, resulting in a "bumpy" graph that is harder to read.

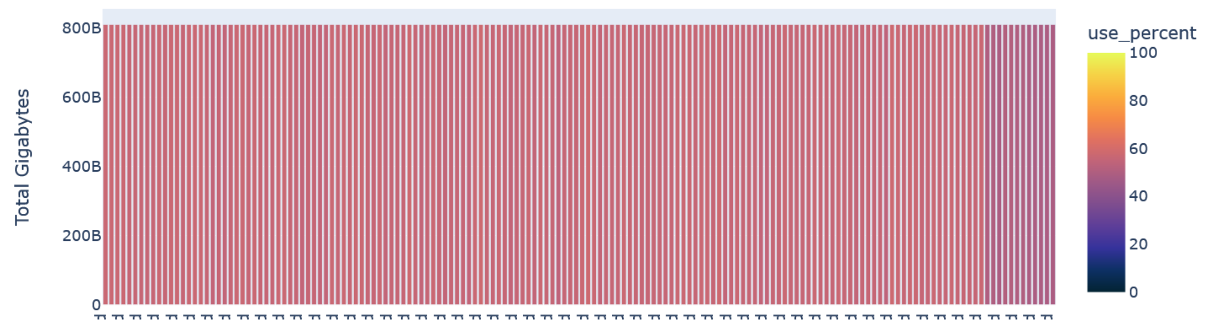


Figure 5.1.: Lustre Overview

Because the Lustre system is so large and split into several partitions, I have only shown one of them here.

While both graphs are not particularly complex, they do offer a good universal baseline. And as we will see, irregularities at this level often reflect on other areas. Maldistributions here already give a hint for why more specific problems might arise. For Lustre, in figure 5.1, we can already see that the last twelve OSTs are less utilized than the others. This could be because they were recently added and not completely filled yet. Though this is still a rather minor deviation and not necessarily a major problem.

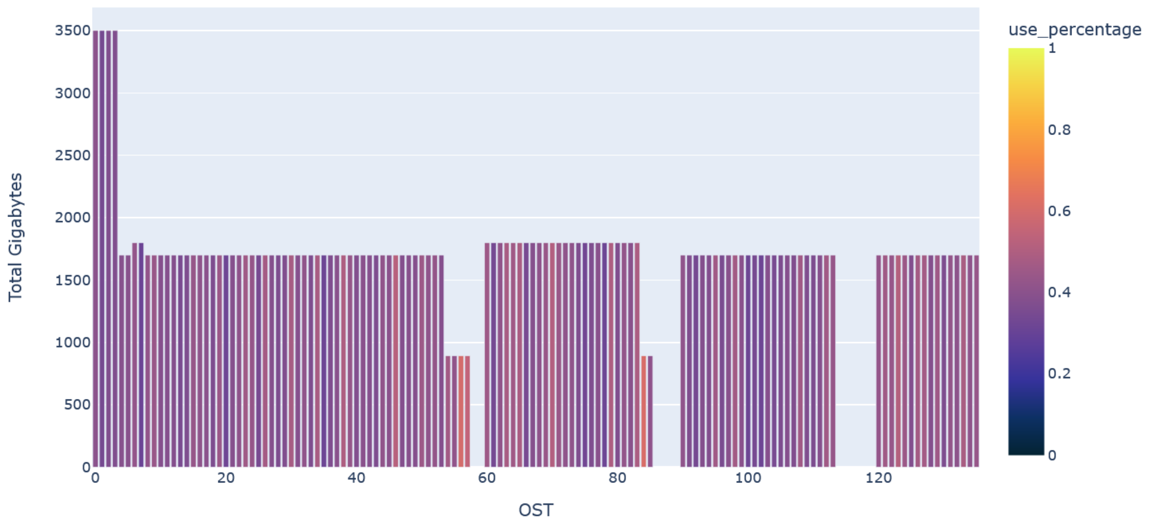
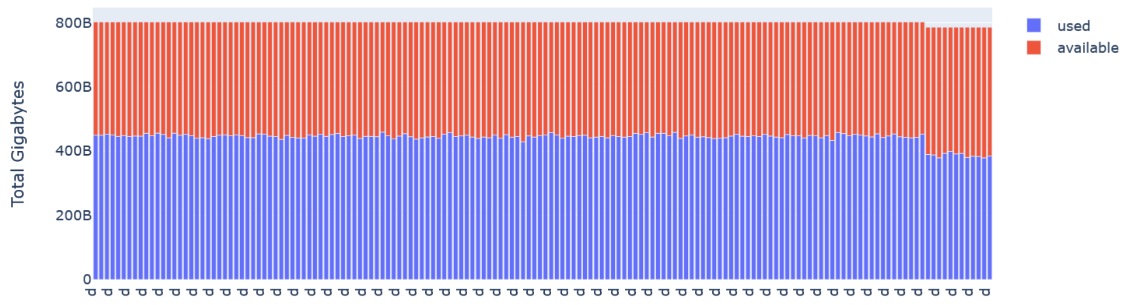
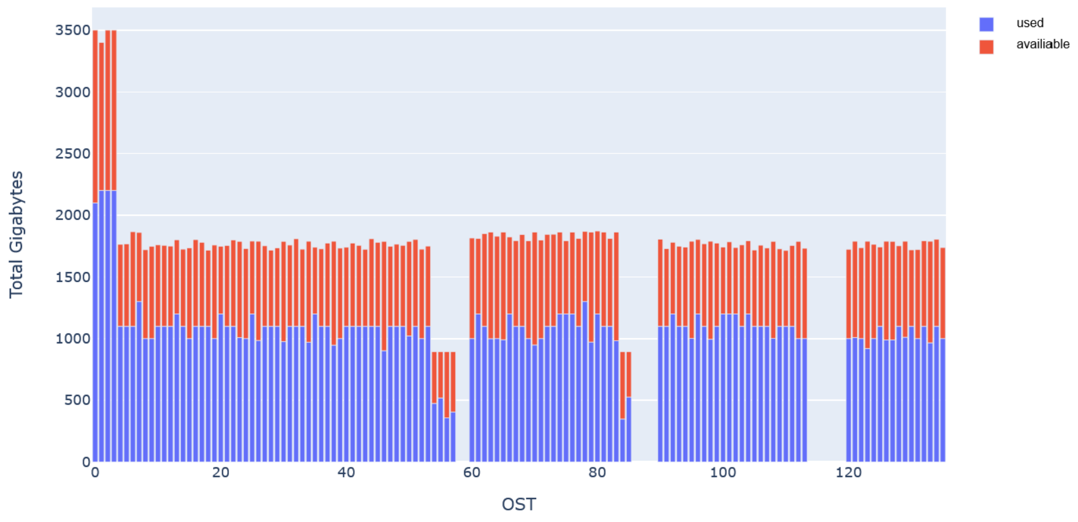


Figure 5.2.: Ceph Overview



(a) Lustre



(b) Ceph

Figure 5.3.: Stacked bar chart overview

Similar observations can be made in figure 5.2 with Ceph, where some of the smaller OSDs are quite a bit more crowded than the rest, although this general view is not very helpful for finding potential causes.

It is also noteworthy that there is a difference in the level of detail between Lustre and Ceph. While Lustre does not provide significantly more overview information than this, Ceph, on the other hand, offers a wealth of information that can be visualized, principally the pools and placement groups, as well as the storage device hierarchy/physical location.
This is mainly due to the information available to me for each system: I got the Lustre storage layout via user APIs, while the Ceph information comes from an administrator.

Ceph specific overview: To analyze the Ceph storage layout in more detail, I built an interactive Dash application that includes placement group and pool data, as outlined in sections 3.2.3 and 3.2.4 . The initial view can be seen in figure 5.4: most prominent is a tree map at the top displaying the OSD topology and class of each storage device. In this case, a SSD is colored orange, a NVMe purple, and a bucket green. At the bottom left is the OSD bar chart from figure 5.2, in the middle a scatter plot for the placement groups, and to the right is another bar chart for the pool data.

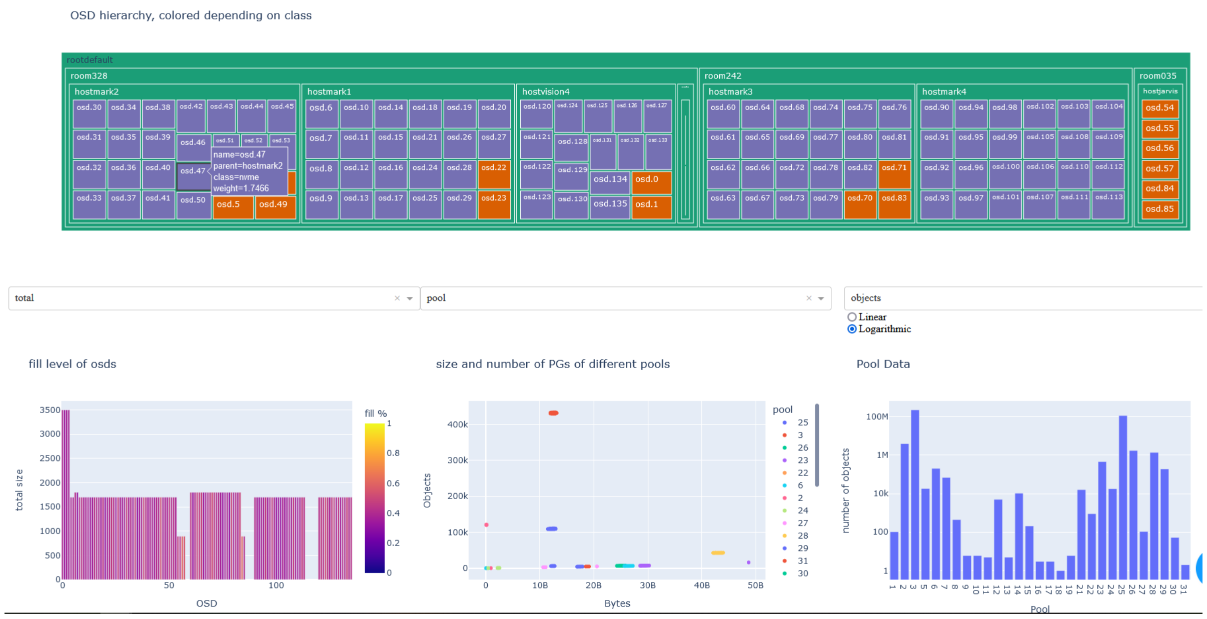


Figure 5.4.: Initial Dashboard showing the relationship between OSDs, PGs and pools

When a bar in the OSD bar chart is clicked, it is colored yellow in the tree map, and all its heartbeat peers that it is connected with are colored in blue, shown in figure 5.5
It must be done in this roundabout way via the bar chart since, as mentioned in section 3.2.4, the tree map already has intrinsic interactivity. Adding a click event that would allow users to directly select an OSD in the tree map to see its peers would lead to rendering errors. Another possible solution I tried was to use a hover event for the tree map, but this would not work for any OSD in the center of the graph, as the hover callback would fire on the first OSD touched by the mouse.

When a PG in the scatter plot is clicked, it shows the primary and secondary OSDs it is stored on and replicated on in the tree map, as shown in figure 5.6. This allows for a quick check to make sure that a PG was indeed stored as intended and is spread out enough.
A problem with Plotly can be seen here: the tree map hides some elements in the initial zoomed-out view, so that certain OSDs are only visible if zoomed in. A second blue OSD is hidden

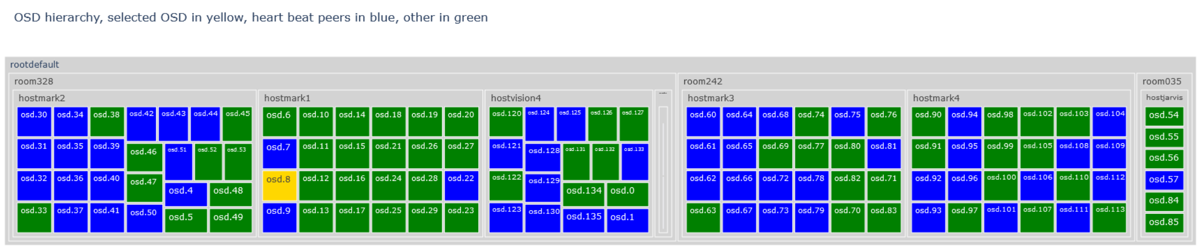


Figure 5.5.: OSD tree map with selected OSD

inside the grey bar in the center of the tree map. There is only a limited amount of screen space available for a graph, and Plotly decides itself which elements it minimizes and which it shows fully. All three graphs have a drop-down menu. The menus for the two bar charts allow users

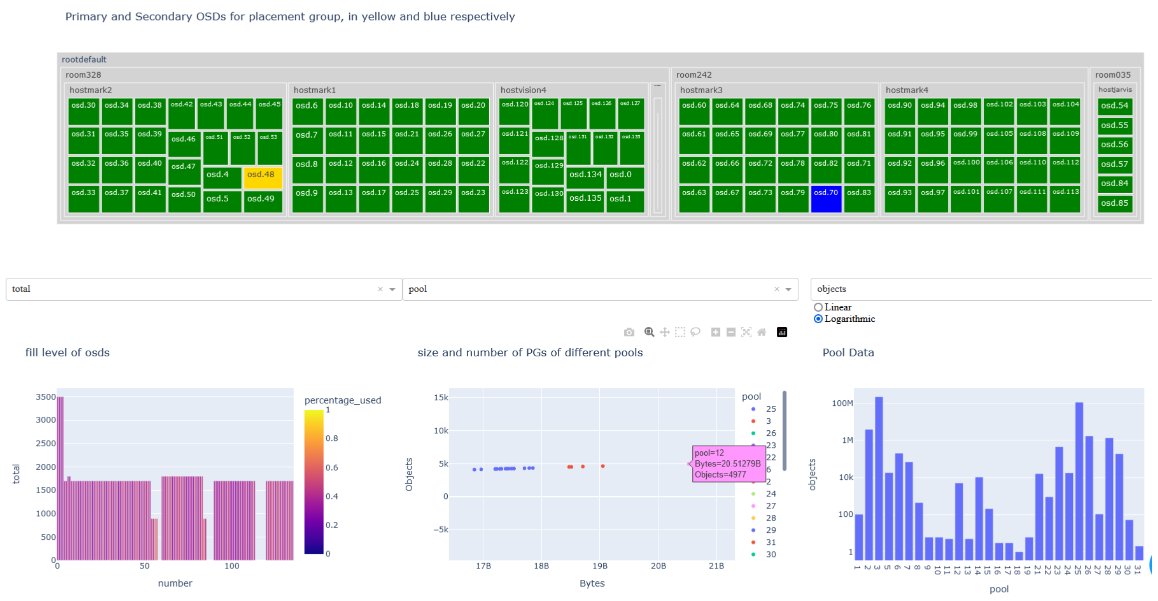
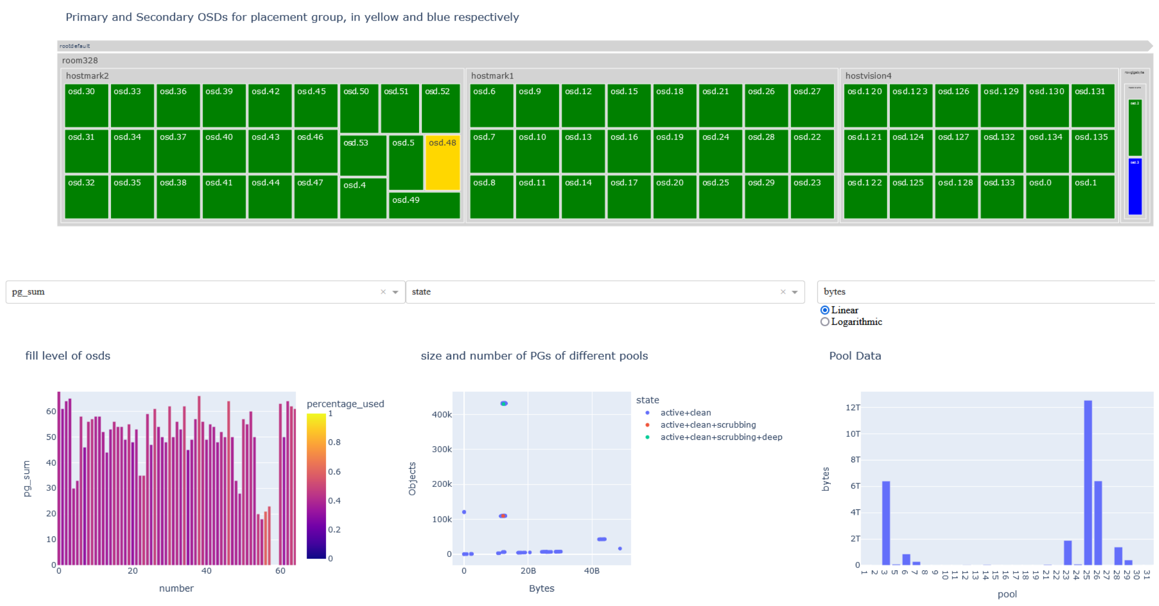


Figure 5.6.: Ceph Dash with selected PG

to select which column of the original table they want displayed on the y-axis. The drop-down menu of the scatter plot changes the color of the PGs. It can either be the pool they belong to or the "state" of the PG, which can be a combination of 32 states. The normal and optimal one is "active+clean", while a damaged placement group might be "degraded" or "inconsistent" and one currently being checked for problems could be "active+clean+scrubbing+clean".

5.2.2. Individual File Distribution

An important aspect is the distribution of a single file that a user might be interested in. It is also the building block of other, more complex visualizations that show the distribution of multiple files and their relationships.
Figure 5.8 shows the visualization of the typical layout of a file in the "/pool/data" partition of Levante. While this most basic use-case is in and of itself rather uneventful, it is a very common layout. Most small files do not need to be striped across a plethora of devices and are instead most efficiently stored in this simple form. That makes it a useful and frequent



pg_sum

state

bytes

[✓ Linear] Logarithmic

Figure 5.7.: Ceph Dash with various y-axis changes

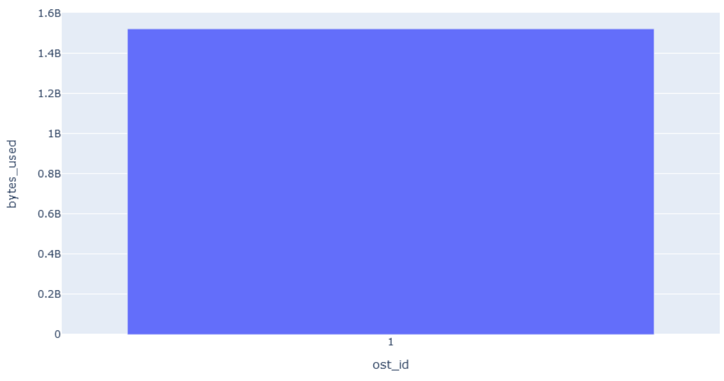
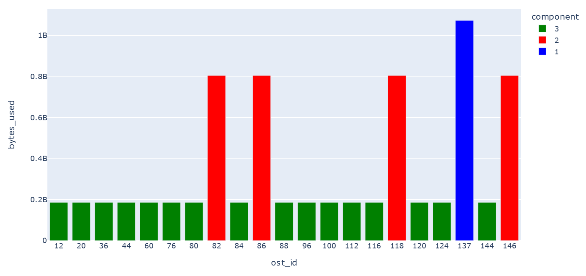


Figure 5.8.: In the most rudimentary case, a file is simple stored on a single OST

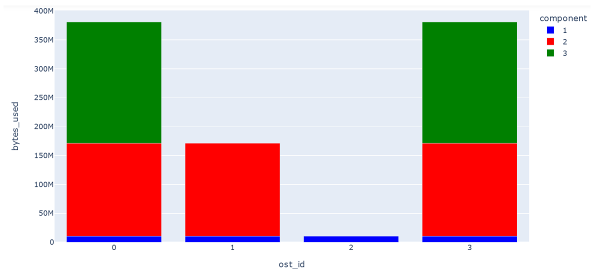
use-case. In these simple cases, a visualization is not always needed, as it might be easy enough to understand from just text. Getting, parsing, and visualizing is work that takes time and is not always in proportion to the insight gained.

A lot more interesting are the two PFL examples in figure 5.9, that both consist of three components but have very different effects. In figure 5.9a the first gigabyte is striped across a single device, enough for small files and necessitating only a small overhead. The next 3 gigabytes are striped across 4 different OSTs, until the rest of the file is striped across 16 devices, which is more than enough for most typical files. This ensures that even if the file grows significantly larger, it will still be balanced and spread relatively evenly across many devices. The opposite is a layout in figure 5.9b that starts by striping data across four devices, then three, and finally just two devices. The larger the file grows, the more the imbalance grows as well, resulting in much slower access than would be possible. The visualization can help in showing the results of mistakes like these that might be overlooked in text form.

These were examples from Lustre. Ceph, on the other hand, is object-based, and showing the location of a file, or even an entire directory, would be rather difficult. Especially as I personally did not have access to the system and many users might neither. Though, at least in theory,



(a) A very efficient PFL



(b) A sub-optimal PFL

Figure 5.9.: Two exemplary progressive file layouts

this procedure is possible, if not entirely quick, efficient, or easy.
All objects in a pool can be listed with the Rados command "rados -p <pool-name> ls", and every object belonging to a desired file would need to be identified. For each object, the ceph command "ceph osd map <pool-name> <object-name>" would need to be sent. The outputs from these commands would need to be parsed for the PG this object is placed in and the OSDs it is stored and replicated on. All of these would have to be gathered and could then be shown in a similar manner to the one I implemented in Lustre.

5.2.3. Multiple File Distribution

Visualizing the file layout of several files is relatively simple. Similar graphs to figure 5.9 are layered on top of each other, though with each file having a different color, instead of each layout component. This is to reduce the complexity of the graph and keep it readable. In this use-case the focus also lies on the location of a collection of files, and less on each individual file layout.
A relatively small example can be seen in figure 5.10. I created several files with varying sizes and layouts, that are stored across the four OSTs of the HOME partition. The inefficient layout from figure 5.9b can be seen in purple and seems relatively good when compared to the light green file "test_1453" that lies entirely on OST 1. Due to this, OST 1 is overloaded, while OST 2 remains nearly untouched.

This visualization is only easily readable for a small number of files. Beyond that, the sheer volume of files and colors becomes overwhelming, and other visualization approaches might be better, like showing the entire directory.
One such case can be seen in Figure 5.11: The hundreds of file layouts are simply too many and make it nearly impossible to decipher the actual distribution of any one file. Which is unfortunate, as there is quite an interesting problem to be seen here: The graph mostly consists of large bars of 1 gigabyte size, but between them are numerous smaller stripes of just a few megabytes, so small that they can not be colored and appear grey.
The reason for this can be seen when one looks at the file sizes: They are all around 1.2 gigabytes. The progressive file layout stripes the first 1 gigabyte across a single OST, and the next 3 gigabytes across four OSTs. This is a good progressive file layout for larger files that are several gigabytes big, where there is actually a significant amount of data to divide across four OSTs. However, in this specific situation, the rest is so small that there is basically no speed benefit from utilizing these four OSTs, it just creates a larger overhead cost for Lustre. Changing the standard PFL to only start utilizing more OSTs after perhaps 2 gigabytes could

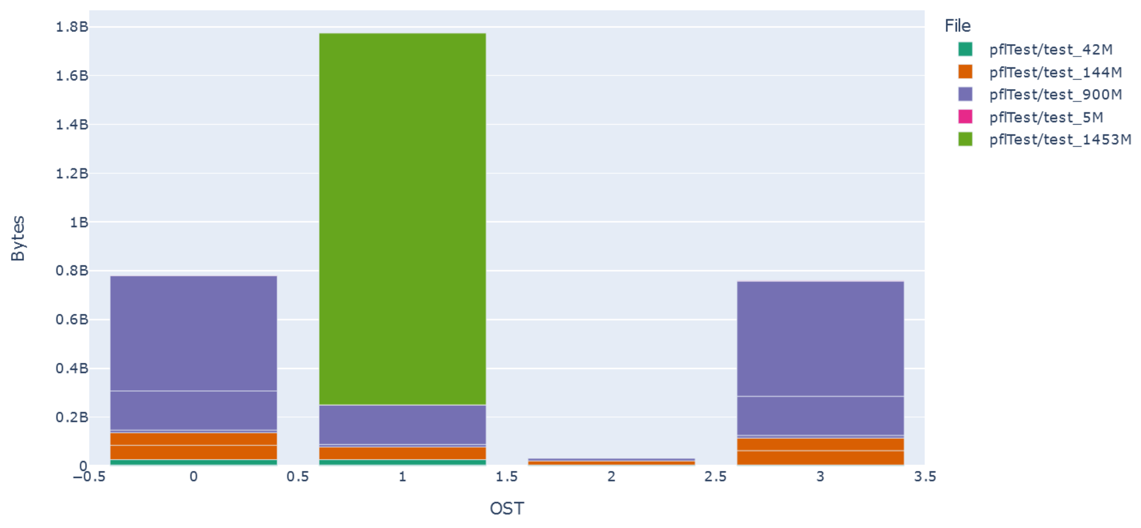


Figure 5.10.: The file distribution of to many files

help immensely.
Additionally, the hundreds of files are stored randomly across 158 OSTs, which leads to a somewhat even distribution. Some storage devices like OST 2 are nevertheless significantly more crowded than others, while some, especially OST 77, remain nearly empty. Such imbalance is to be expected from a random selection and is still within an acceptable range.

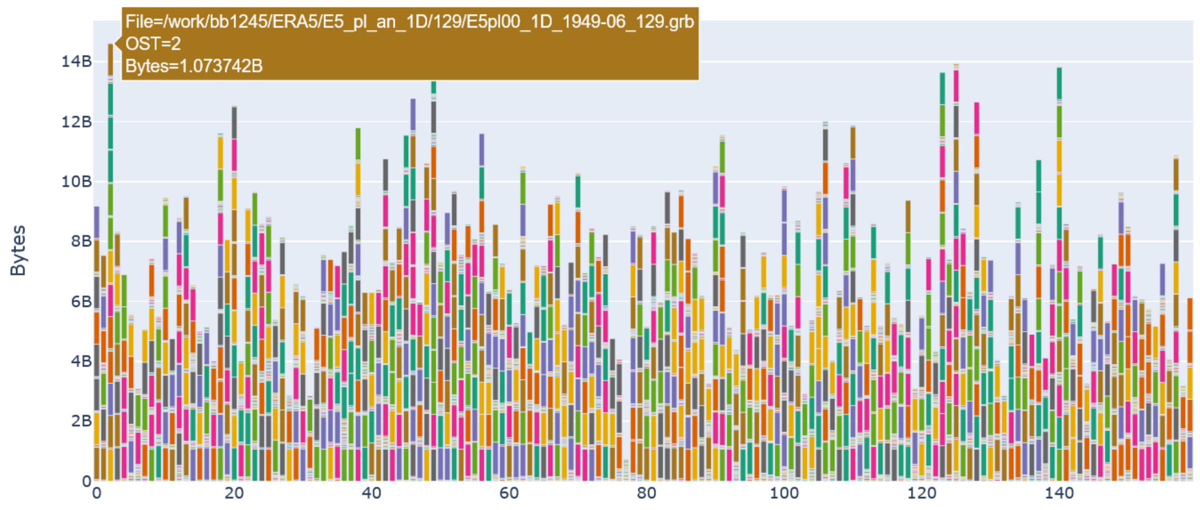


Figure 5.11.: The file distribution of 970 files

5.2.4. Multiple File Overlap

The aforementioned visualization works well when the distribution of files is of particular interest. But many applications may read and write from and to several files, so avoiding conflicts between two groups of files might be advantageous. For that, a heat map is very helpful, an example of which can be seen in figure 5.12. It shows the overlap between two groups of files on each OST, utilizing a thermal scale. As we can see, in this hypothetical scenario, hot spots like OST 2, 18, 20, and 110 might experience conflicts and congestion, while other OSTs like 6 or 31 are underutilized.

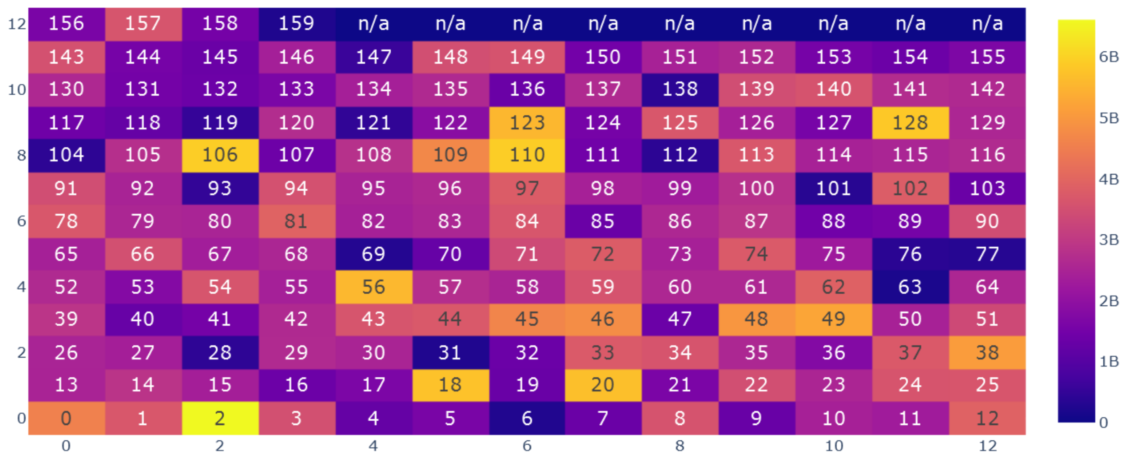


Figure 5.12.: Heat map of the overlap between two groups of files

A slight difficulty I encountered with heat maps in Plotly is the need for a mostly square shape to show every element at once in a compact form. If given just a list of values, Plotly also only shows a very long, one-dimensional heat map which makes comparing and spotting values rather difficult. But for a square heat map, a square matrix of all the values is needed. Should the number of elements not have an integer square root, dummy elements are needed to fill the square matrix, shown with "n/a" in the heat map. This is not a huge problem, but it might give the impression on first glance that there are some OSTs in the top row that are underutilized.

5.2.5. Directory Structure

Finally, the last use-case I looked at was the the analysis of an entire directory and all files in it and its subfolders. To tackle this problem, I built the Dash application seen in figure 5.13 and 5.14. It shows the fill level of each OST as a bar chart, with the color denoting the number of files from the analyzed directory that are stored on that particular OST. Below this bar chart are three more graphs that show different statistics of the directory files, as described in section 3.2.4.
As an example, I analyzed the directory "/pool/data", which contains over half a million files. Most striking is the nearly perfect distribution across most OSTs, except for the last twelve, which remain nearly untouched by the directory. These are also the OSTs that are slightly less filled, which was already visible in figure 5.1 and 5.3a.
For an initial search of potential causes, the Dash application shows the gathered file statistics in three graphs. These show, that most files were last modified in March 2018, though some date back over two decades. Nearly all of them were created by a single user and are rather small.

While these statistics are for all the files, it is possible to select only the outliers in the top bar chart, as seen in figure 5.14, which limits the statistic graphs to only files from those selected OSTs. While most of these files are still relatively small and most belong to the same user as before, a far greater part of them were created by a different user. Most striking is perhaps the time component, which shows most files on the underutilized OSTs to be very young. Nearly all of them are from a single date in 2023, with nearly every other file from various days in

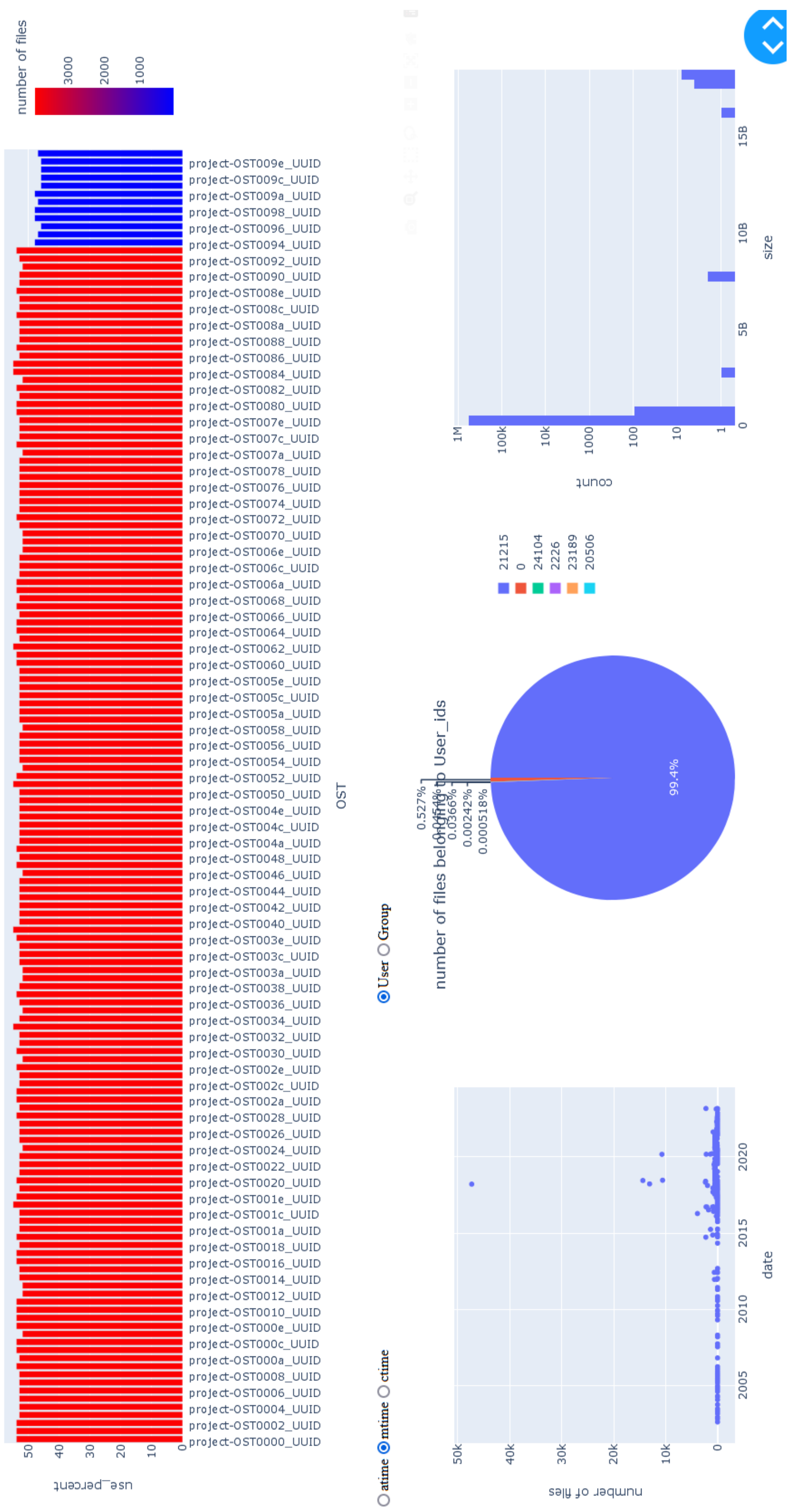


Figure 5.13.: Inintial Lustre Dashboard

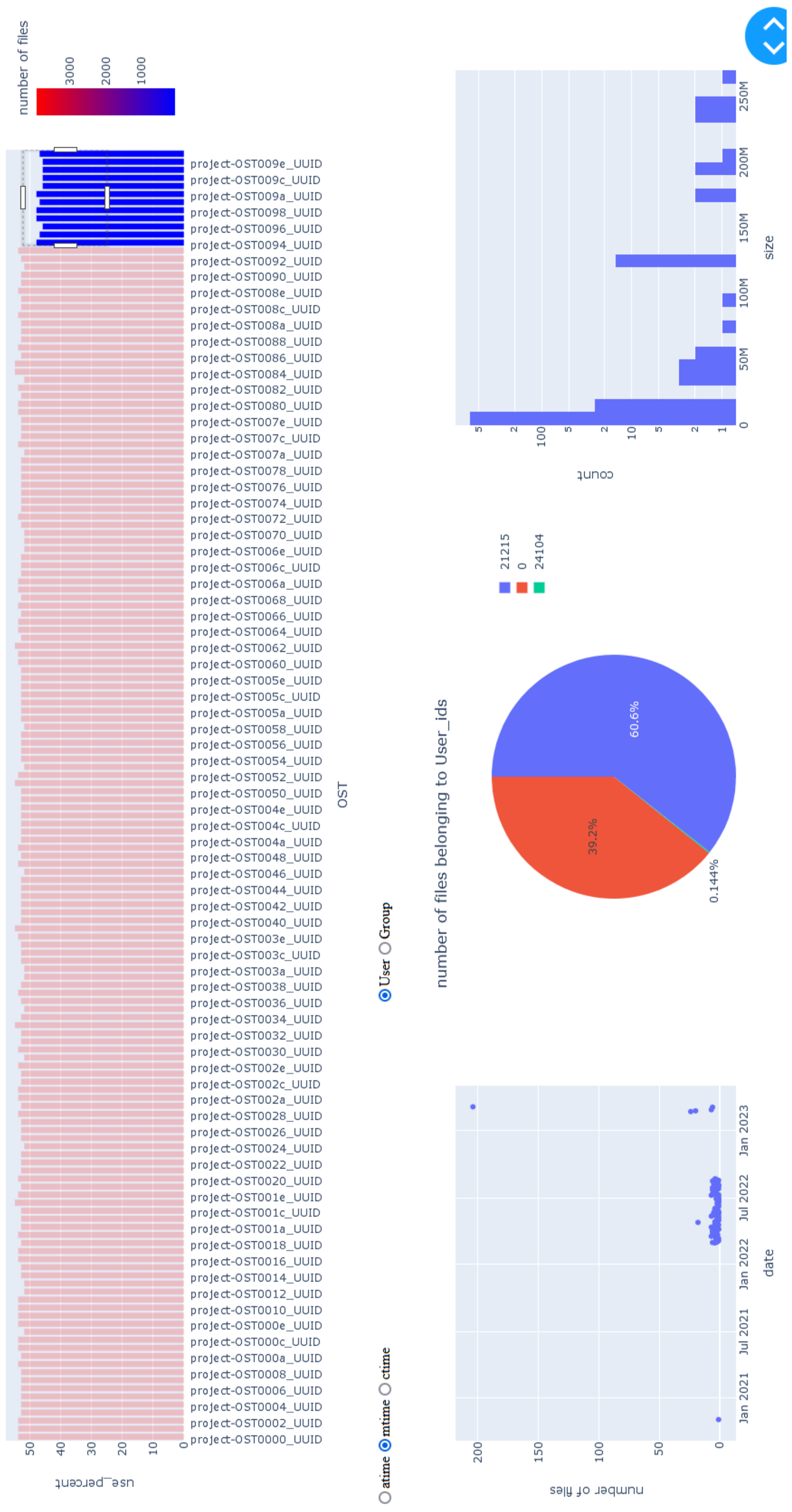


Figure 5.14.: Selection of the last twelve OSTs

2022. This is in stark contrast to the rest of the files, which tend to be much older and spread across a far wider time frame.

Thus, a possible cause might be that the last twelve OSTs were a recent addition to the cluster and older files were not (yet) moved to them. New files, on the other hand, are placed mostly randomly across all available OSTs by Lustre and are slowly starting to populate them.
To prove this hypotheses, more information about the storage devices might be helpful, like when they were added to the cluster. Such information is most likely only accessible to the administrator of the cluster. Another hint would be the statistics of every other file on these twelve OSTs to see if they were also recently added. This would require gathering the file layout and statistics of each and every file in the entire system, which would undoubtedly overwhelm it.

Summary

In this chapter we have seen a broad selection of visualizations for various use cases: First a very basic overview of the storage device utilization that works for most distributed systems. System specific additions, like a cluster map, pool and placement group information for Ceph are necessary for a truly fruitful analysis. Secondly single file visualizations work quite well in Lustre and successfully reveal maldistributions. These can be extended to cover the layout of multiple files and even to compare the overlap between two groups of files. Ceph, unfortunately, is much more restrictive with its access. Finally, for large amounts of files a separate directory visualization can be helpful for finding issues and providing first hints for potential causes.

Chapter 6.

Conclusion and Future Work

6.1. Conclusion

Overall, this thesis was successful in creating interactive visualizations for the storage layout of distributed systems. I was able to develop nearly universal visualizations that work beyond just a single system, though by themselves they are quite limited and need to be joined by more system-specific graphs for any deeper analysis. Specialized charts about Lustre file striping were surprisingly successful and could showcase very interesting cases of unbalanced storage distributions.
Unfortunately, I could not work with Ceph directly, which mostly excluded me from the file and object side of it. However, even the more easily available PG, OSD and pool data were quite intricate and interconnected, which lend themselves well to an interactive visualization.

Especially when working on these interactive relationships, I noticed the limitations of the visualization libraries I chose, Plotly and Dash, and was more often than not working against them, instead of with them. Simpler interactive behaviour, like the OST-selection for the Lustre directory visualization, was relatively easy once I understood the concepts behind Dash. But when I attempted a more omnidirectional interactivity for Ceph, where the same graph modifies other graphs but can also itself be modified, it became quite complicated and prone to bugs. Nevertheless, these libraries did offer me an unprecedented level of comfort in quickly creating various graphs with many useful features, like hover tool tips or color bars. These would otherwise have been much more time-consuming or impossible to create with other visualization libraries.

6.2. Future Works

6.2.1. Utilize more available information

The first potential avenue for future works is to utilize more aspects that Lustre and especially Ceph have to offer: While I have utilized most commands that Lustre offers about the storage layout, there are many parts of these commands that I did not utilize, be it for time constraints, lack of knowledge, deviation from the topic of storage layout, or incorrect assumptions about the usefulness of certain information. Through this, it might be possible to delve deeper into the physical storage layout and the actual location of files and stripes on a storage device, as this can influence how fast I/O operations are done.

This is even more so the case for Ceph, where I did not have first-hand access to the outputs and used just a small part of all the data that were available to me. A huge part of Ceph that I omitted are the placement rules that define how and where a file is placed and replicated. They are very simple to access, but I could not find a satisfactory solution on how best to integrate them with the overall storage layout in a way that could really benefit users. There is also a plethora of placement group information available through the command `pg dump` that seemed unimportant or simply not understandable. While the documentation for Ceph commands is quite thorough, there is very little information about their outputs, much of which one has to guess or cobble together.

6.2.2. Physical Layout

While I focused more on a high-level overview and the overall distribution of files, an equally interesting aspect is the hardware side and the actual physical layout on the disks. Such information is more difficult to come by and not necessarily available to the average user, but it is nevertheless crucial for optimal storage. The fragmentation of a disk or the position of its read-and-write head for each fragment can matter a lot as well

6.2.3. Increase Usability

Increasing the usability of the tools could be another path, not only to increase the efficiency with which an administrator might use them, but also to give more casual users of distributed systems the tools to make the most out of their supercomputer. Not every climate scientist working at the DKRZ might want to or have the time to learn the intricacies of Lustre. Still, how they choose to store files has a direct effect on the speed of their simulations, whether they know it or not. Good default settings are invaluable and give serviceable results in most cases, though by their very design they leave a lot of room for optimization in specific situations. Giving each user an accessible way of optimizing their work could help the entire system utilize its resources as much as possible.

6.2.4. Changes over time

An interesting direction for future research would involve incorporating a time component. This could mean illustrating how storage use changes over time, how specific files grow and move, to identify potential bottlenecks and hot spots. Such an approach might help in planning for future system expansions and overhauls, determining which parts should be prioritized to keep up with increasing requirements. Some key considerations might be the rate at which storage devices fill up, typical user behaviour, and the life cycle of a file. These might be particularly interesting for improving the PFL in Lustre, as it is specifically designed to accommodate changes in a file’s size.

Bibliography

[BMF, 2023] BMF (2023). Sollwerte des Haushaltsjahres 2023. (Cited on page 10)

[Ceph-Foundation, nd] Ceph-Foundation (n.d.). Ceph Dashboard. retreived from `https://docs.ceph.com/en/quincy/mgr/dashboard/`, (Accessed 2023-06-12). (Cited on page 27)

[Ceph-Foundation, nd] Ceph-Foundation (n.d.). Introduction to Ceph. retrieved from `https://docs.ceph.com`.May 13, 2023. (Cited on page 5)

[Der Bundeswahlleiter, 2023] Der Bundeswahlleiter (2023). Bundestagswahl 2021 069:Magdeburg. (Cited on page 10)

[DKRZ, nd] DKRZ (n.d.). HLRE-4 Levante - DKRZ High-Performance Computing System. retreived from `https://www.dkrz.de/en/systems/hpc/hlre-4-levante?set_language=en`, (Accessed 2023-05-16). (Cited on pages 1 and 29)

[Gluster, nd] Gluster (n.d.). Gluster documentation. (Cited on page 8)

[Grafana, 2019] Grafana (2019). Grafana Lustre Overview. (Cited on page 27)

[Hadoop, nd] Hadoop (n.d.). HDFS Architecture Guide. (Cited on page 8)

[Heer et al., 2010] Heer, J., Bostock, M., and Ogievetsky, V. (2010). A tour through the visualization zoo. *Commun. ACM*, 53(6):59–67. (Cited on page 9)

[Herold and Breuner, 2018] Herold, F. and Breuner, S. (2018). Introduction to BeeGFS. (Cited on page 8)

[JungJungIn, n d] JungJungIn (n. d.). Sonar 4 CEPH (sonar4ceph). (Cited on page 26)

[Lustre, 2010] Lustre (2010). *Lustre manual.* Lustre, https://doc.lustre.org/lustre_manual.xhtml, version 2.x edition. Retrieved May 13, 2023. (Cited on pages 3 and 5)

[Lustre, nd] Lustre (n.d). Introduction to Lustre. (Cited on pages 3 and 4)

[Metcalfe and Boggs, 1976] Metcalfe, R. M. and Boggs, D. R. (1976). Ethernet: Distributed Packet Switching for Local Computer Networks. *Commun. ACM*, 19(7):395–404. (Cited on page 3)

[Plotly, nd] Plotly (n.d.). Dash User Guide. (Cited on page 9)

[Schach et al., 2020] Schach, D., Tröger, J., Endt, C., and Erdmann, E. (2020). Corona-Fälle in Europa: Mehr Infizierte als jemals zuvor. Zeit Online. retreived from `https://www.zeit.de/wissen/2021-08/coronafaelle-europa-verlauf-coronavirus-pandemie-animation-datenanalyse`. (Cited on page 10)

[Schnorr et al., 2013] Schnorr, L., Legrand, A., and Vincent, J.-M. (2013). Interactive Analysis of Large Distributed Systems with Scalable Topology-based Visualization. (Cited on pages 25 and 26)

[Tufte, 1983] Tufte, E. R. (1983). *The Visual Display of Quantitative Information.* Graphics Press. (Cited on pages 9 and 10)

[Weil, 2019] Weil, S. (2019). Intro to Ceph. Tech Talk. (Cited on page 6)

[Weil et al., 2006] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. (2006). CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing.* (Cited on pages 5, 6, and 7)

Statement of Authorship

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

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

Magdeburg, June 13, 2023

Lennart Börchers