Monitoring parallel file system usage in a high-performance computer cluster

Jaan Tollander de Balsch

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 2023-02-21

Supervisor

Prof. Petteri Kaski

Advisor

Dr. Sami Ilvonen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Jaan Tollander de Balsch

Title Monitoring parallel file system usage in a high-performance computer cluster

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Petteri Kaski

Advisor Dr. Sami Ilvonen

Date 2023-02-21 **Number of pages** 55+8 **Language** English

Abstract

Many high-performance computer clusters, rely on a system-wide, shared, parallel file system for large storage capacity and bandwidth. A shared file system is available across the entire system, making it user-friendly but prone to problems from heavy use. Such use can cause congestion and slow down or even halt the whole system, harming all users who use the parallel file system. In this thesis, we investigate whether monitoring file system usage in a production system at CSC can help identify the causes of slowdowns, such as specific users or jobs. The long-goal at CSC is to build an automatic, real-time monitoring and warning system that system administrators can use to make decisions on alleviating the slowdowns. Specifically, we monitor the usage of the Lustre parallel file system with Lustre Jobstats feature in the Puhti cluster, which is a petascale cluster with a diverse user base. We explain the necessary details of the Puhti cluster and our monitoring system to understand the Lustre file system usage data. During the thesis, we discovered issues in the data quality from Lustre Jobstats. The issues affected identifiers in the data, making some data unreliable and limiting our ability to build an automatic, real-time analysis. Nevertheless, we obtained a feasible data set for explorative data analysis. We demonstrate 24 hours of monitoring data by visually demonstrating file system usage patterns at low and high-level. Furthermore, we show that we can use file system usage data to identify causes of relative changes in I/O trends, particularly large relative increases. Finally, we explore ideas for future work on monitoring file system usage with reliable data from longer periods.

Keywords monitoring computer systems, observability, computer cluster, high-performance computing, parallel file system, Lustre, I/O behavior, time series analysis, exploratory data analysis

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Jaan Tollander de Balsch

Työn nimi Rinnakkaistiedostojärjestelmän käytön valvonta suurteholaskenta tietokoneklusterissa

Koulutusohjelma Tieto-, tietoliikenne- ja informaatiotekniikka

Pääaine Tietotekniikka **Pääaineen koodi** SCI3042

Työn valvoja Prof. Petteri Kaski

Työn ohjaaja TkT. Sami Ilvonen

Päivämäärä 2023-02-21 **Sivumäärä** 55+8 **Kieli** Englanti

Tiivistelmä

Monet tehokkaat tietokoneklusterit luottavat järjestelmän laajuiseen, jaettuun rinnakkaistiedostojärjestelmään suuren tallennuskapasiteetin ja kaistanleveyden saavuttamiseksi. Jaettu tiedostojärjestelmä on käytettävissä koko järjestelmässä, mikä tekee siitä käyttäjäystävällisen, mutta altis raskaan käytön aiheuttamille ongelmille. Tällainen käyttö voi aiheuttaa ruuhkautumista ja hidastaa tai jopa pysäyttää koko järjestelmän, mikä vahingoittaa kaikkia rinnakkaistiedostojärjestelmää käyttäviä käyttäjiä. Tässä opinnäytetyössä tutkimme voiko CSC:n tuotantojärjestelmän tiedostojärjestelmän käytön seuranta auttaa tunnistamaan hidastumisen syitä, kuten tiettyjä käyttäjiä tai töitä. CSC:n pitkäaikainen tavoite on rakentaa automaattinen, reaaliaikainen valvonta- ja varoitusjärjestelmä, jonka avulla järjestelmänvalvojat voivat tehdä päätöksiä hidastumisen lievittämiseksi. Tarkemmin sanottuna seuraamme Lustre rinnakkaistiedostojärjestelmän käyttöä Lustre Jobstats ominaisuudella Puhti-klusterissa, joka on monipuolisen käyttäjäkunnan omaava petascale-klusteri. Selitämme tarvittavat yksityiskohdat Puhti-klusterista ja valvontajärjestelmästämme Lustre-tiedostojärjestelmän käyttötietojen ymmärtämiseksi. Opinnäytetyön aikana havaitsimme ongelmia Lustre Jobstats:in tietojen laadussa. Ongelmat vaikuttivat tiedoissa oleviin tunnisteisiin, mikä teki joistakin tiedoista epäluotettavia ja rajoitti kykyämme luoda automaattinen, reaaliaikainen analyysi. Siitä huolimatta saimme käyttökelpoisen tietojoukon tutkivaa data-analyysiä varten. Esittelemme 24 tunnin seurantatietoja näyttämällä visuaalisesti tiedostojärjestelmän käyttötapoja matalalla ja korkealla tasolla. Lisäksi osoitamme, että voimme käyttää tiedostojärjestelmän käyttötietoja tunnistamaan syitä suhteellisiin muutoksiin I/O-trendeissä, erityisesti suurissa suhteellisissa nousuissa. Lopuksi tutkimme ideoita tulevaa työtä varten tiedostojärjestelmän käytön seuraamiseksi luotettavalla tiedolla pidemmältä ajalta.

Avainsanat tietokonejärjestelmien valvonta, havaittavuus, tietokoneklusteri, suurteholaskenta, rinnakkaistiedostojärjestelmä, Lustre, I/O käyttäytyminen, aikasarjaanalyysi, tutkiva data-analyysi

Acknowledgments

I sincerely thank my thesis advisor, doctor Sami Ilvonen and supervisor, professor Petteri Kaski. Sami's work with the monitoring system and writing advice were instrumental in completing this thesis. Petteri's guidance with writing and thoughtful comments improved the thesis significantly.

Furthermore, I thank Sebastian Von Alfthan for providing me the chance to work as part of a great team on a fascinating problem; Simon Westersund, who set up the monitoring system and proof-reading the thesis; Ulf Tigerstedt, who helped to manage the database; CSC - The IT Center for Science for a place to work with amazing and talented people; Aalto University for the many years of education; and the Aalto Scientific Computing system administrators, Simo Tuomisto and Mikko Hakala, for explaining their monitoring workflow.

Espoo February 21, 2023

Jaan Tollander de Balsch

Contents

1 Introduction

Persistent data storage is an essential part of a computing system. Many high-performance computing (HPC) systems, typically computer clusters, rely on a system-wide, shared, parallel file system for large storage capacity and bandwidth. A shared file system is available across the entire system, making it user-friendly but prone to problems from heavy use. Such use may lead to congestion in a parallel file system, which can slow down or even halt the whole system, harming all users who use the file system, not just the ones responsible for the problem. Heavy use may be intentional, such as data-intensive computing, or unintentional, such as unknowingly running a program that creates many temporary files or a program that uses the file system for communication between processes. In this thesis, we investigate whether monitoring file system usage in a production system can help identify the causes of slowdowns, such as specific users or jobs.

The professional literature typically refers to interaction with storage as *I/O*, an abbreviation for *Input/Output*. Generally, I/O refers to communication between a computer and the outside world, but we often use it to describe interactions with a storage device. A file system is a commonly used abstraction layer between the physical storage device and the user, but there are others, such as object storage. The term *storage I/O* is agnostic about the underlying abstraction layer. In this work, the I/O refers to storage I/O.

Traditionally, we measure the performance of an HPC system in standard linear algebra operations per second, focusing on the processor and memory [1], [2]. However, storage I/O performance is also becoming important in HPC system due to a rise in data-intensive workloads, such as data science and machine learning workflows, which relies on huge amounts of data. The system must transport this data between main memory and storage, making I/O performance essential and problems from heavy I/O more common. The increasing demand for better I/O performance in HPC systems makes studying it necessary. The HPC community has also established new benchmarks to measure I/O performance, such as the ones discussed in the IO500 benchmarks [3]. Research from institutions and companies such as Oak Ridge National Laboratory, Lawrence Berkeley National Laboratory, Virginia Tech, Cray, and Seagate is actively finding ways to improve I/O performance in HPC. For example, they research ways to improve parallel file systems [4], [5] and develop alternative storage solutions [6], [7].

Since parallel file systems are shared, and heavy usage can cause problems, educating users about how to use them correctly is crucial. Many HPC facilities have guidelines for performing file I/O on high-performance clusters. For example, Texas Advanced Computing Center (TACC)'s guidelines [8] advise avoiding overburdening the parallel file system with bad practices and moving the heavy I/O to local temporary storage. Furthermore, they list common bad practices and solutions for them, such as the following:

1) Using many small files instead of a few large files. Accessing the same amount

of data from many small files than fewer large files requires more file system operations.
2) Having too many files in a single directory instead of using subdirectories or local temporary storage.
3) Not striping large files; we should stripe large files. Striping a file refers to storing consecutive segments of a large file into multiple storage devices for improved performance. Automatic striping or tools to help users to stripe files with the correct stripe count may alleviate this problem.
4) Performing suboptimal file I/O patterns. For example, patterns that create large amounts of unnecessary file system operations, such as repeatedly opening and closing the same file.
5) Performing high-frequency file I/O instead of keeping data in memory or limiting the I/O frequency.
6) Accessing the same file from multiple processes simultaneously instead of creating copies of the file or using parallel I/O libraries.
7) Overlooking I/O patterns workloads; we should use I/O profiling tools.

Another solution is to use tools for throttling the I/O rate of jobs performing heavy I/O [9]. Users can proactively throttle their workloads, or administrators can throttle jobs with heavy I/O without and avoid suspending these jobs.

Even if users follow the guidelines, problems eventually occur. To identify when they occur, we must actively monitor file system performance. Furthermore, by measuring I/O performance and using statistical time-series analysis, we can identify variations in performance trends, such as short-transient or long-persistent ones, and changes in baseline performance over time [10]. However, we need more than performance monitoring to identify who or what is causing problems in a parallel file system. To identify causes, we can monitor file system health, capacity, and usage, track system changes, and use data from the resource manager. This thesis focuses on fine-grained file system usage monitoring to identify the causes of short-transient problems. *Fine-grained* refers to collecting statistics of each file system operation to identify who performs the operations, from which node, and to which storage unit. Fine-grained monitoring shows us detailed file system behavior instead of a single aggregate of its performance. This work is a part of the greater need for measuring and understanding I/O behavior in HPC systems [11], [12].

Problems from parallel file system usage concern the high-performance clusters at CSC – IT Center for Science, which provides ICT services for higher education institutions, research institutes, culture, public administration, and enterprises in Finland. At the time of writing, CSC operates three high-performance clusters, Puhti, Mahti, and the pan-European LUMI, which all use the *Lustre* parallel file system [13]. Especially the Puhti cluster is susceptible to service disruptions from heavy file system usage, which leads to lost productivity, lost computational resources, and increased administrative work. Monitoring file system usage will help us to identify the causes of the problems and take action faster to alleviate them.

Lustre has a feature called *Lustre Jobstats* [14] for collecting file system usage statistics

at a fine-grained level. Early experimental monitoring with Jobstats includes [15], [16]. More recently, Jobstats have been used to collect long-term, job-level I/O patterns for improving storage design [17]. We will use Jobstats to collect statistics at higher granularity than the previous studies. Commercial monitoring products also work with Lustre Jobstats, such as View for ClusterStor [18] and DDN Insight [19]. Unfortunately, these products did not meet our monitoring and analysis needs, which led us to develop a custom solution.

In this work, we monitor and analyze file system usage in the *Puhti* cluster. Our long-term goal at CSC is to build active monitoring and near real-time warning systems to identify who and what causes problems in the file system. This thesis takes steps towards achieving this goal. Currently, Puhti has system-level load monitoring from processor usage, file system capacity monitoring, and job information from the workload manager, which cannot identify the causes of the problems. When problems occur, system administrators have to determine the causes manually. However, the problem often disappears before they have identified the actual cause. Active monitoring of file system usage should help system administrators to identify the causes and take action as the issues occur, not afterward. It should also reduce the amount of manual work involved.

The scope of the thesis is to describe the necessary details of high-performance computing, Lustre parallel file system, and Puhti cluster for collecting fine-grained file system usage statistics and the monitoring system built around them. The thesis advisor and system administrators were responsible for developing, deploying, and maintaining the monitoring system on Puhti. Their effort was instrumental in initiating the thesis work, collecting the data, and helping with writing the thesis. Furthermore, performing explorative data analysis on the obtained monitoring data and visualizing and explaining the results belongs to the scope. The main contributions of the thesis are the resulting insights that help us build the real-time monitoring and warning system.

The thesis is structured as follows. In Section 2, we present a general overview of high-performance computing and specific software related to high-performance clusters. We also describe the configuration of the Puhti cluster from a storage perspective and explain the necessary system identifiers for fine-grained data. In Section 3, we describe the monitoring system and explain how we collect data, what data we collect, and how we store it. Section 4 presents methods and results from explorative data analysis on the collected monitoring data during this thesis. We explore issues with data quality and how they affected the thesis work, provide visualizations and explanations of the monitoring data, demonstrate that we can identify users who perform heavy I/O relative to others from the data, and present ideas for improving the analysis methods in the future. Section 5 concludes by discussing the general aspects of the thesis work, accomplished thesis goals, and perspectives for future work on monitoring file system usage and I/O behavior.

2 High-performance computing

Fundamentally, computing is about repeatedly applying a logical rule on a string of symbols to transform it for some goal-oriented task, such as solving a mathematical problem. In practice, we use computers for computing, which they do by leveraging physical processes. Contemporary computers represent data in digital form as binary digits called bits. Rules correspond to instructions to a computer processor that manipulates a string of bits in memory. Memory consists of multiple levels of volatile main memory and non-volatile storage organized hierarchically based on factors such as proximity to the processor, access speed, and cost. Models of computation include serial and parallel computing. Serial computing refers to performing one operation at a time. In contrast, parallel computing is about performing multiple independent operations simultaneously, intending to reduce run time, and performing larger calculations.

High-performance computing (HPC) relies on parallel computing to provide large computing resources for solving computationally demanding and data-intensive problems. These problems include simulations of complex systems, solving large computational models, data science, and training machine learning models. Examples of commercial and research applications of HPC include:

- Weather modeling for weather forecasting
- Climate modeling for understanding climate change
- Financial analytics for trading decisions
- Data analysis for oil and gas exploration
- Fluid simulation, such as airflow for cars and airplanes
- Molecular dynamics simulation for pharmaceutical design
- Cosmological simulation for understanding galaxy creation
- Biosciences such as next-generation sequencing for studying genomes

Most HPC systems are computer clusters. *Computer cluster* connects multiple computers, called *nodes*, via a high-speed network to form a more powerful system. It usually has a large amount of storage as well. Computer clusters are usually centrally managed by organizations such as companies or universities. They rely on administrators and software from the organization and various vendors to configure the machine, install software, orchestrate their services, and maintain them. The organizations may offer access to the machine as a service with billing based on the usage of computer resources, such as the amount of time, memory, processors, and storage requested.

2.1 Linux operating system

An operating system (OS) is software that manages computer resources and provides standard services for application programs via an application programming interface (API). At the time of writing, practically all high-performance computer clusters use the *Linux* operating system [20]. Linux derives from the family of UNIX operating systems and closely follows the POSIX standard.

The Linux kernel [21] is the core of the Linux operating system, written in the C programming language. The kernel is the central system that manages and allocates computer resources such as processors, memory, and other devices. Its responsibilities include process scheduling, memory management, providing a file system, creating and terminating processes, access to devices, networking, and providing an application programming interface for system calls, making the kernel services available to programs. System calls enable user processes to request the kernel to perform specific actions for the process, such as manipulating files. They also provide separation between kernel space and user space. Library functions, such as functions in the C standard library, implement a caller-friendly layer on top of system calls for performing system operations. [22]

Linux implements a universal file I/O model, which means that it represents everything from data stored on disk to devices and processes as files. It uses the same system calls for performing I/O on all types of files. Consequently, users can use the same file utilities to perform various tasks, such as reading and writing files or interacting with processes and devices. The kernel only provides one file type, a sequential stream of bytes.

The kernel provides an abstraction layer called *Virtual File System (VFS)*, which defines a generic interface for file-system operations for concrete file systems such as ext4, Btrfs, or FAT. VFS allows programs to use different file systems uniformly using the operations defined by the interface. The interface contains the system calls such as `open()`, `close()`, `read()`, `write()`, `mknod()`, `unlink()` and others. Linux Man Pages [23, Sec. 2] provides in-depth documentation for system calls. We demonstrate the relationship between different system calls with code examples in Appendix A.

Linux is a multiuser system, which means that multiple users can use the computer at the same time. The kernel provides an abstraction of a virtual private computer for each user, allowing multiple users to operate independently on the same computer system. Linux systems have one special user, called the super user or root, which can access everything on the system. System administrators can use the super user for administrative tasks and system maintenance.

A *Linux distribution* comprises some version of the Linux kernel combined with a set of utility programs such as a shell, command-line tools, a package manager, and, optionally, a graphical user interface.

2.2 Client-server application

A *client-server application* is an application that consists of two processes, a client and a server. The *client* requests a server to perform some service by sending a message. The *server* listens for the client's messages, examines them, performs the appropriate actions, and sends a response message back to the client. The client and server may reside in the same or separate host computers connected by a network. They communicate with each other by some Interprocess Communication (IPC)

mechanism. Usually, the client application interacts with a user, while the server application provides access to a shared resource. Commonly, there are multiple instances of client processes communicating with one or a few instances of the server process.

2.3 Lustre parallel file system

A *parallel file system* is a file system designed for clusters. It stores data on multiple networked servers to facilitate high-performance access. It makes the data available via a global namespace such that users do not need to know the physical location of the data blocks to access a file. *Lustre* is a parallel file system that provides a POSIX standard-compliant file system interface for Linux clusters. The Lustre file system is implemented as a set of kernel modules designed using the client-server architecture. A kernel module is software that extends the kernel, in this case, to provide a new file system. [13], [24, Secs. 1–2]

Nodes running the Lustre client software are known as *Lustre clients*. The Lustre client software interfaces the virtual file system with *Lustre servers*. For Lustre clients, the file system appears as a single, coherent, synchronized namespace across the whole cluster. Lustre file system separates file metadata and data operations and handles them using dedicated Lustre servers. Each Lustre server connects to one or more storage units called *Lustre targets*.

Metadata Servers (MDS) provide access to file metadata and handle metadata operations for Lustre clients. Lustre stores the metadata, such as filenames, permissions, and file layout, on one or more storage units attached to an MDS, called *Metadata Targets (MDT)*. On the other hand, *Object Storage Servers (OSS)* provide access to and handle file data operations for Lustre clients. Lustre breaks file data into one or more objects; it stores each object on one or more storage units attached to an OSS, called *Object Storage Targets (OST)*. Finally, the Management Server (MGS) stores configuration information for the Lustre file system and provides it to the other components. Lustre file system components are connected using Lustre Networking (LNet), a custom networking API that handles metadata and file I/O data for the Lustre file system servers and clients. LNet supports many network types, including high-speed networks used in HPC clusters. Lustre has a feature called *Lustre Jobstats* for collecting file system operations statistics from a Lustre file system. We discuss how we use Jobstats in Section 3.

2.4 Slurm workload manager

Typically, the nodes on a cluster are separated into a front and back end. The front end consists of login and utility nodes, and the back end consists of compute nodes. Login nodes are meant for login into the system and performing light, interactive tasks, and compute nodes are meant for performing heavy computing workloads. Clusters rely on a workload manager to allocate access to computing resources, schedule, and run programs on the back end. The programs may instantiate an

interactive or batch process. A *batch process* is a computation that runs from start to finish without user interaction compared to interactive processes such as an active terminal prompt or a text editor which respond to user input.

Slurm is a workload manager for Linux clusters [25], [26]. Unlike Lustre, Slurm operates in the user space, not kernel space. Slurm provides a framework for starting, executing, and monitoring work on the allocated nodes with requested computing resources such as nodes, cores, memory, and time. Resource access may be exclusive or nonexclusive, depending on the configuration. Slurm maintains a queue of jobs waiting for resources to become available and can perform accounting for resource usage. We refer to a resource allocation as a *job*, which may contain multiple steps that may execute sequentially or in parallel. Administrators can group nodes into Slurm partitions to set different policies, such as queuing policies and maximum resource allocations. A node may belong to more than one partition.

2.5 Puhti cluster at CSC

In order to build a monitoring system for CSC's Puhti cluster, we need to understand certain aspects of its hardware and software configuration. Puhti is a Petascale system, referring to the peak performance above 10^{15} floating point operations per second. It has over five hundred unique monthly users and a diverse user base, making it prone to problems from heavy use of the parallel file system, and thus interesting for studying. Puhti is a Finnish noun that means having energy. We explain the hardware configuration of Puhti, including the nodes, processors, memory, storage, and network. Then, we cover the system configuration, such as the operating system, specific names and identifiers, and storage areas.

Table 1: Prefixes of units in base ten and base two.

Value	Prefix	Value	Prefix
1000^1	kilo (k)	1024^1	kibi (Ki)
1000^2	mega (M)	1024^2	mebi (Mi)
1000^3	giga (G)	1024^3	gibi (Gi)
1000^4	tera (T)	1024^4	tebi (Ti)
1000^5	peta (P)	1024^5	pebi (Pi)

We use units of bytes and bits and base ten and base two prefixes, as shown in Table 1. One byte (B) is eight bits (b). Units of memory size use bytes with base two prefixes, such as gibibytes (GiB), storage size uses bytes with base ten prefixes, such as gigabytes (GB), and network bandwidth uses bit rates with base ten prefixes, such as gigabits per second (Gb/s).

Table 2: Nodes on the Puhti cluster. Puhti has different node categories based on their function in the cluster. Lustre nodes serve the Lustre file system, service nodes form the front end and serve utility functions, and compute nodes form the back end of the cluster. Each node category contains different node types, and each node of a given type has identical resources, such as processor, memory, and local storage. The node count describes how many nodes of a given type Puhti contains.

Node category	Node type	Node count	Memory (GiB per node)	Local storage (GB per node)
Lustre	MDS (virtual)	2		
Lustre	OSS (virtual)	8		
Service	Utility	3	384	2900
Service	Login	2	384	2900
Service	Login-FMI	2	384	2900
Compute	CPU, M	484	192	-
Compute	CPU, M-IO	48	192	1490
Compute	CPU, M-FMI	240	192	-
Compute	CPU, L	92	384	-
Compute	CPU, L-IO	40	384	3600
Compute	CPU, XL	12	768	1490
Compute	CPU, BM	6	1500	5960
Compute	GPU	80	384	3600

The *Puhti* cluster has various *service nodes* and 1002 *compute nodes* as seen in Table 2. The services nodes consist of *utility nodes* for development and administration, *login nodes* for users to log in to the system, and MDS and OSS nodes for the Lustre file system. The compute nodes consist of 922 CPU nodes and 80 GPU nodes. Each login and compute node consists of two Intel Xeon Gold 6230 Central Processing Units (CPUs) with 20 cores and 2.1 GHz base frequency. In addition to CPUs, each GPU node has four Nvidia Volta V100 Graphical Processing Units (GPUs), and each GPU has 36 GiB of GPU memory. We type nodes based on how much Random-access Memory (RAM) and *fast local storage* they contain and whether they contain GPUs. Fast local storage is a Solid State Disk (SSD) attached to the node via Non-Volatile Memory Express (NVMe) for processes to perform I/O intensive work instead of relying on the system-wide storage.

The system-wide storage on Puhti consists of a Lustre parallel file system, introduced in Section 2.3. At the time of writing, Puhti has Lustre version 2.12.6 from DataDirect Networks (DDN). Puhti's Lustre configuration contains two virtualized MDSs and eight virtualized OSSs with an SFA18KE controller. Each MDS has two MDTs. All four MDTs share 20 of 800 GB NVMe via Linux Volume Manager (LVM). Each OSS has three OSTs. Each OST is connected to 30 of 9 TiB SAS HDD. The total storage capacity of the file system is 4.8 PiBs since part of the total capacity is reserved for redundancy. Storage is connected using Mellanox InfiniBand EDR links.

The cluster connects nodes via a network with an fat-tree topology. Each node connects to one of 28 L1 switches in the network, and each L1 switch connects to all 12 L2 switches. The connections use Mellanox InfiniBand HDR technology offering 100 Gb/s bandwidth for each node. Figure 1 shows a simplified, high-level overview of the network.

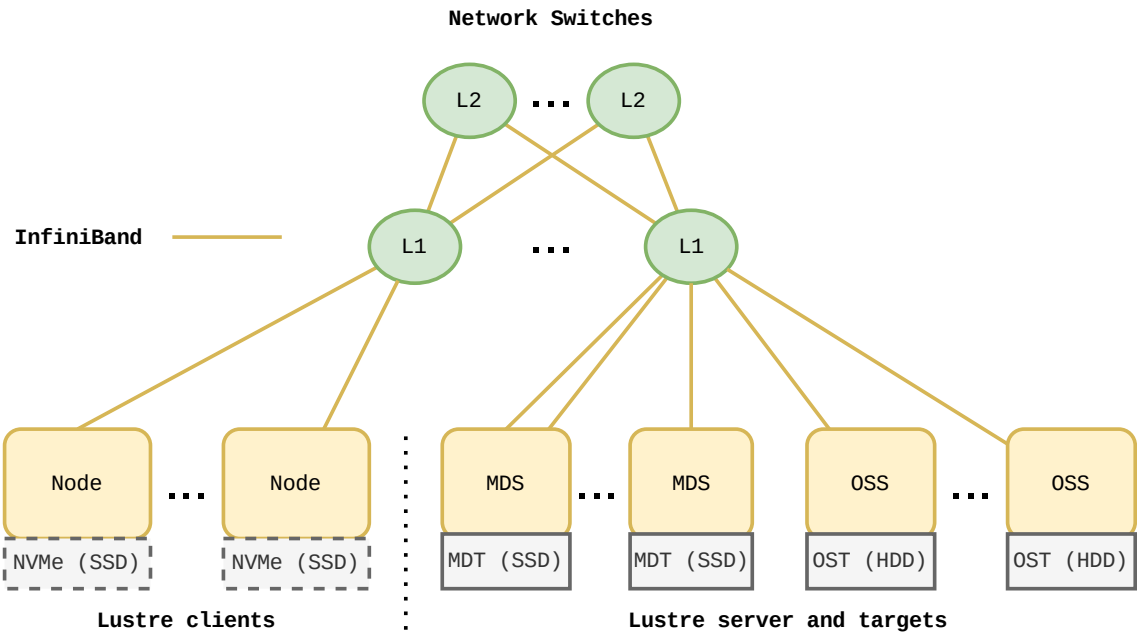

Figure 1: Puhti's configuration from a storage perspective. Rounded rectangles on the left illustrate compute, utility, and login nodes, whereas the dashed rectangles below are the optional attached local storage. Rounded rectangles on the right illustrate the Lustre servers, where the rectangles below are the appropriate Lustre targets. The lines represent the network connections, and the circles represent the network switches. Three dots between nodes or switches indicate that there are many of them.

As mentioned in Section 2.1, most high-performance clusters use the Linux operating system. Puhti also uses Linux as its operating system, specifically the RedHat Enterprise Linux Server (RHEL) distribution which transitioned from version 7.9 to 8.6 during the thesis writing.

Each Lustre server and target has a name in the Lustre file system. We record file system usage statistics for each target. Table 3 lists the names of Lustre targets for the corresponding Lustre server in Puhti. We denote set such that curly braces `{...}` denote a set, ranges such as `{01-04}` expand to `{01,02,03,04}`, and products such as `{a,b}{c,d}` expand to `{ac,ad,bc,bd}`. Furthermore, we add curly braces to elements outside them, such as `a{c,b}` is `{a}{c,b}` and expand them as a product.

Table 3: Names of Lustre servers and Lustre targets in Puhti. For example, `scratch-MDT0000` is the name of one of the MDTs, and `scratch-OST000f` is the name of one of the OSTs. The prefix `scratch-` is the mount point for Lustre directories under the root directory, `/scratch/`. We should not confuse it with the Scratch storage area discussed later, which is mounted under the scratch directory `/scratch/scratch/`.

Node category	Node Type	Index	Targets
Lustre	MDS	1	`scratch-MDT{0000,0001}`
Lustre	MDS	2	`scratch-MDT{0002,0003}`
Lustre	OSS	1	`scratch-OST{0000,0001,0002}`
Lustre	OSS	2	`scratch-OST{0003,0004,0005}`
Lustre	OSS	3	`scratch-OST{0006,0007,0008}`
Lustre	OSS	4	`scratch-OST{0009,000a,000b}`
Lustre	OSS	5	`scratch-OST{000c,000d,000e}`
Lustre	OSS	6	`scratch-OST{000f,0010,0011}`
Lustre	OSS	7	`scratch-OST{0012,0013,0014}`
Lustre	OSS	8	`scratch-OST{0015,0016,0017}`

Each node in Puhti is a Lustre client of the shared Lustre file system. We can identify nodes based on their *node name*, which is part of the hostname before the first dot, for example, `<nodename>.bullx`. Table 4 lists the names of service and compute nodes. We can use node names to separate file system operations at a node-specific level.

Table 4: Names of service and compute nodes in Puhti that have Lustre Jobstats enabled. For example, `puhti-login11` is the name of one of the login nodes, and `r01c21` is the name of one of the compute nodes.

Node category	Set of node names
Service	`puhti-login{11-16}`
Service	`puhti-fmi{11-12}`
Service	`puhti-ood1-{production}`
Service	`puhti-ood2-{testing,production}`
Compute	`r{01-04}c{01-48}`
Compute	`r{01-04}g{01-08}`
Compute	`r{05-07}c{01-64}`
Compute	`r08m{01-06}`
Compute	`r{09-10}c{01-48}`
Compute	`r{11-12}c{01-72}`
Compute	`r{13-18}c{01-48}`
Compute	`r{13-18}g{01-08}`

Puhti separates its file system into storage areas such that each storage area has a

dedicated directory. It shares the same Lustre file system across Home, Projappl, and Scratch storage areas with different uses and quotas.

- Home is intended for storing personal data and configuration files with a fixed quota of 10 GB and 100 000 files per user.
- Projappl is intended for storing project-specific application files such as compiled libraries with a default quota of 50 GB and 100 000 files per project.
- Scratch is intended for short-term data storage in the cluster with a default quota of 1 TB and 1 000 000 files per project.

As a general guideline, jobs should use the Scratch area for storing data. They should access the Home or Projappl areas only to read or copy configuration or application-specific files at the beginning of the job.

Puhti also has two local storage areas, Local scratch, and Tmp. They are intended for temporary file storage for I/O heavy operations to avoid burdening the Lustre file system. Users who want to keep data from local storage after a job completion must copy it to scratch since the system regularly cleans the local storage areas.

- Local scratch, mounted on a local SSD, is indented for batch jobs to perform I/O heavy operations. Its quota depends on how much the user requests for the job.
- Tmp, mounted on RAMDisk, is intended for login and interactive jobs to perform I/O heavy operations such as post and preprocessing data, compiling libraries, or compressing data.

In CSC systems, users have a user account that can belong to one or more *projects*. We use projects for setting quotas and accounting for computational resources and storage. Puhti associates each user account with a *user* and each project with a *group*. We can use user IDs to measure file system usage at the user level. However, we should retrieve the group ID from the workload manager's accounting as a project ID. RHEL 7 and 8 reserve user IDs from 0 to 999 for system processes. We refer to the users with IDs from 0 to 999 as *system users* and other users as *non-system users*. It is helpful to separate the file system operations performed by system users from the non-system users. We are more interested in the file system usage of non-system users that system users.

Puhti uses the Slurm workload manager, introduced in Section 2.4. At the time of writing, the version was 21.08.7, but it is updated regularly. It has partitions with different resource limits, set by administrators, as seen in Table 5. When we submit a job to Slurm, we must specify which partition it will run, the project used for billing, and the resource we want to reserve. We present concrete examples of Slurm job scripts for Puhti in Appendix B. Slurm schedules the job to run when sufficient resources are available using a fair share algorithm. It also performs accounting of details about the submitted jobs.

Table 5: Slurm partitions on Puhti at the time of writing. Each partition has a name and resource limits such as time, task, and node limit that a job can request on the partition. Node types, listed in Table 2, dictate the nodes where a job may run. Typically, memory and local storage limits are the same for the node type.

Partition name	Time limit	Task limit	Node limit	Node type
`test`	15 minutes	80	2	M
`interactive`	7 days	8	1	M-IO, L-IO
`small`	3 days	40	1	M, L, M-IO, L-IO
`large`	3 days	1040	26	M, L, M-IO, L-IO
`longrun`	14 days	40	1	M, L, M-IO, L-IO
`hugemem`	3 days	160	4	XL, BM
`hugemem_longrun`	14 days	40	1	XL, BM
`fmitest`	1 hour	80	2	M-FMI
`fmi`	12 days	4000	100	M-FMI
`gputest`	15 minutes	8	2	GPU
`gpu`	3 days	80	20	GPU

Slurm sets different job-specific environment variables for each job such that programs can access and use them. An important environment variable is the *Slurm Job ID*, accessed via `SLURM_JOB_ID`, which we use as an identifier to collect job-specific file operations. We can combine the data from Slurm's accounting with the file system usage using the job ID. For example, we can use the information about the project, partition, and local storage reservation of a job. Project information might help identify if members of a particular project performs heavy I/O.

3 Monitoring system

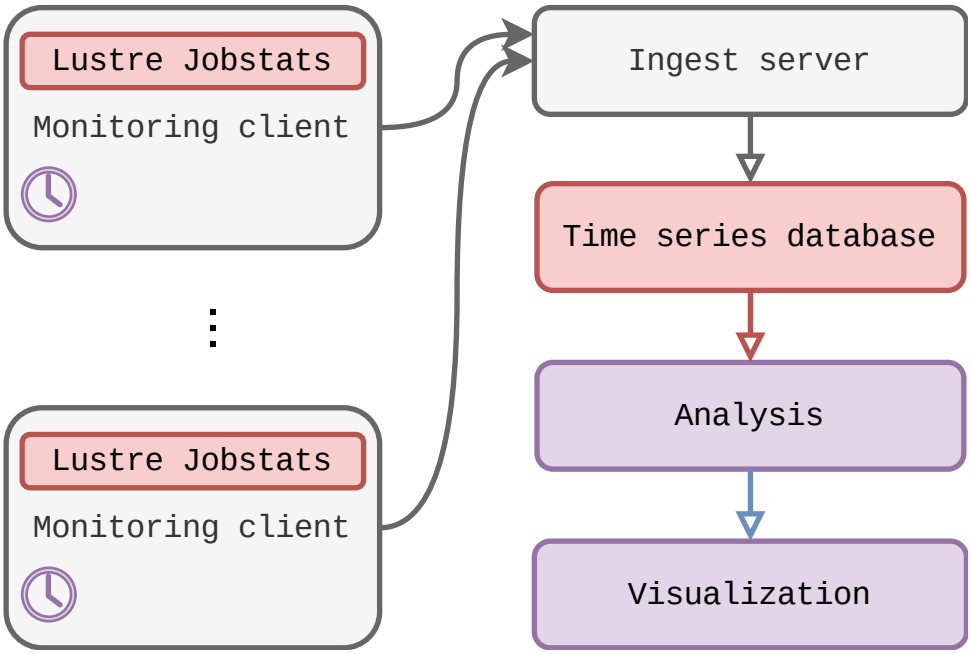

Figure 2: A high-level overview of the monitoring system and analysis. Rounded rectangles indicate programs, and arrows indicate data flow.

This section explains how our monitoring system works and how we collect file system usage statistics from the Lustre file system in the Puhti cluster using Lustre Jobstats. We built the monitoring system as a client-server application, consisting of a monitoring client, an ingest server, and a time series database, illustrated in Figure 2. We do not monitor the usage of the local storage areas because monitoring their usage is complicated. Since they are not part of the Lustre file system, we cannot use Lustre Jobstats. A more practical option is to combine the reservations for local storage from Slurm accounting using job identifiers from Lustre Jobstats data.

Subsection 3.1 covers the settings we use for the entry identifiers for collecting fine-grained statistics. In Subsection 3.2, we explain the different file system operations statistics we can track, how we query them, and the output format. In Subsection 3.3, we explain when Lustre Jobstats resets the statistics it collects. Subsection 3.4 explains how to compute average file system usage rates from the statistics, which we will use in the analysis. The statistics we collect from Jobstats form multiple time series. We explain how we store time series data in the *time series database* in Subsection 3.5. In Subsection 3.6, we explain how the *monitoring client* collects the usage statistics from Lustre Jobstats on each Lustre server and sends them to the *ingest server*. Due to various issues, we had to modify the monitoring client during the thesis. These changes affected the analysis and required significant changes in the analysis code and methods. We explain the initial and modified versions of the monitoring client. Subsection 3.7 explains how we detect and handle initial statistics entries. Subsection 3.8 explains how the ingest server processes the data from the

monitoring clients and inserts it into the time series database.

The thesis advisor and system administrators were responsible for enabling Lustre Jobstats, developing the monitoring client and ingest server, installing them on Puhti, and maintaining the database. We adapted the Monitoring client and Ingest server codes from a GPU monitoring program written in the Go language [27], which used InfluxDB [28] as a database. We changed the database to TimescaleDB [29]. We take the precise design of programs as given and explain them only at a high level. The thesis work focused on the analysis and visualization parts we explain in Section 4.

3.1 Entry identifier format

We can enable Jobstats by specifying a formatting string for the *entry identifier* using the `jobid_name` parameter on a Lustre client as explained in the Lustre Manual [24, Sec. 12.2]. We can configure each Lustre client separately and specify different configurations for different clients. We can use the following format codes.

- `%e` for *executable name*.
- `%h` for *fully-qualified hostname*.
- `%H` for *short hostname* or *node name*, which removes everything from the fully-qualified hostname after the first dot, including the dot.
- `%j` for *job ID* from environment variable specified by `jobid_var` setting.
- `%u` for *user ID* number.
- `%g` for *group ID* number.
- `%p` for *process ID* number.

The formatting affects the resolution of the statistics. Using more formatting codes results in higher resolution and leads to a higher rate of data accumulation.

The formatting for Lustre clients on login nodes includes the executable name and user ID.

- `jobid_name="%e.%u"`

The formatting for Lustre clients on compute and utility nodes includes job ID, user ID, and node name. We set the job ID to Slurm job ID.

- `jobid_name="%j:%u:%H"`
- `jobid_var=SLURM_JOB_ID`

For Puhti, we listed the node names in Table 4.

3.2 File system statistics

Each Lustre server keeps statistics for all of its targets. We can fetch the statistics and print them in a text format by running the `lctl get_param` command with an argument that points to the desired jobstats. We can query jobstats from the Lustre server as follows:

```
lctl get_param <server>.<target>.jobstats
```

The text output is formatted as follows.

```
<server>.<target>.job_stats=
job_stats:
- job_id: <entry_id_1>
  snapshot_time: <snapshot_time_1>
  <operation_1>: <statistics_1>
  <operation_2>: <statistics_2>
  ...
- job_id: <entry_id_2>
  snapshot_time: <snapshot_time_2>
  <operation_1>: <statistics_1>
  <operation_2>: <statistics_2>
  ...
```

The server (`<server>`) parameter is `mdt` for MDSs and `odbfilter` for OSSs. The target (`<target>`) contains the name of the Lustre target of the query. For Puhti, we listed them in Table 3.

After the `job_stats` line, we have a list of entries for workloads that have performed file system operations on the target. The output denotes each *entry* by dash `-` and contains the *entry identifier* (`job_id`), *snapshot time* (`snapshot_time`), and various operations with statistics. The value of snapshot time is a timestamp as a Unix epoch when the statistics of one of the operations are last updated. Unix epoch is the standard way of representing time in Unix systems. It measures time as the number of seconds elapsed since 00:00:00 UTC on 1 January 1970, excluding leap seconds.

In Table 6, we list the MDT and OST operations for which Jobstats keeps statistics. Each operation (`<operation>`) contains a line of statistics (`<statistics>`), formatted as key-value pairs separated by commas and enclosed within curly brackets:

```
{ samples: 0, unit: <unit>, min: 0, max: 0, sum: 0, sumsq: 0 }
```

The samples (`samples`) field counts how many operations the job has requested since Jobstats started the counter with an implicit unit of requests (`reqs`). The statistics fields are minimum (`min`), maximum (`max`), sum (`sum`), and the sum of squares (`sumsq`) for either measure of latency, that is, how long the operation took, with unit microseconds (`usecs`) or for a measure of transferred bytes with unit bytes (`bytes`). These fields contain nonnegative integer values. The samples, sum, and sum of squares increase monotonically except if reset. Statistics of an entry that has not performed any operations are implicitly zero. We collect the value from the `samples` field from all operations, except `read_bytes` and `write_bytes`, where we collect the value from the `sum` field. In this thesis, we did not look at the latency values.

Table 6: This table lists all operations tracked by the Jobstats for each Lustre target. The light gray operation names indicate that the operation field is present in the output, but the values were always zero. Thus, we did not include them in our analysis.

MDT	OST	Operation	Explanation of the operation
MDT	-	`open`	Opens a file and returns a file descriptor.
MDT	-	`close`	Closes a file descriptor that releases the resource from usage.
MDT	-	`mknod`	Creates a new file.
MDT	-	`link`	Creates a new hard link to an existing file. There can be multiple links to the same file.
MDT	-	`unlink`	Removes a hard link to a file. If the removed hard link is the last hard link to the file, the file is deleted, and the space is released for reuse.
MDT	-	`mkdir`	Creates a new directory.
MDT	-	`rmdir`	Removes an empty directory.
MDT	-	`rename`	Renames a file by moving it to a new location.
MDT	OST	`getattr`	Return file information.
MDT	OST	`setattr`	Change file ownership, change file permissions such as read, write, and execute permissions, and change file timestamps.
MDT	-	`getxattr`	Retrieve an extended attribute value
MDT	-	`setxattr`	Set an extended attribute value
MDT	OST	`statfs`	Returns file system information.
MDT	OST	`sync`	Commits file system caches to disk.
MDT	-	`samedir_rename`	File name was changed within the directory.
MDT	-	`crossdir_rename`	File name was changed from one directory to another.
-	OST	`read`	Reads bytes from a file.
-	OST	`write`	Writes bytes to a file.
MDT	OST	`punch`	Manipulates file space.
-	OST	`get_info`	-
-	OST	`set_info`	-
-	OST	`quotactl`	Manipulates disk quotas.
MDT	OST	`read_bytes`	Output from `read` operation.
MDT	OST	`write_bytes`	Output from `write` operation.
-	OST	`create`	-
-	OST	`destroy`	-
-	OST	`prealloc`	-

3.3 Entry resets

If Jobstats has not updated the statistics of an entry within the *cleanup interval* by looking at whether the snapshot time is older than the cleanup interval, it removes the entry. We refer to the removal as *entry reset*. We can specify the cleanup interval in the configuration using the `job_cleanup_interval` parameter. The default cleanup interval is 10 minutes.

We detect the resets by observing if any counter-values have decreased. This method does not detect reset if the new counter value is larger than the old one, but it is uncommon because counter values typically grow large. We might underestimate the counter increment in this case when calculating the difference between two counter values.

3.4 Computing rates

We can calculate a *rate* during an interval from two counter values by dividing the difference between the counter values by the interval length. We treat the previous counter value as zero if we detect a reset. For Jobstats, a rate during an interval tells us how many operations, on average, happen per time unit during an interval. For example, if the previous counter of write operations for a job is $v_1 = 1000$ at time $t_1 = 0$ seconds, and the current value is $v_2 = 2000$ at time $t_2 = 120$ seconds, it performed $v_2 - v_1 = 1000$ write operations during the interval of $t_2 - t_1 = 120$ seconds. Therefore, on average, the job performed $1000/120 \approx 8.33$ write operations per second during the interval. We explain theoretical details about computing rates in Appendix C.

3.5 Storing time series data

We can use a time series database to efficiently store and handle time series data from multiple distinct time series. *Time series data* has distinctive properties that allow optimizations for storing and querying them. TimescaleDB documentation [30] characterizes these properties as follows:

1) *time-centric* meaning that records always have a timestamp.
2) *append-only* means that we almost always append new records and rarely update existing data or backfill missing data about old intervals.
3) *recent* new data is typically about recent time intervals.

There are different options for choosing time series databases. A key differentiation between time series databases is whether they can handle a growing number or a fixed number of distinct time series. We used TimescaleDB because it can handle data with a growing number of distinct time series. TimescaleDB expands PostgreSQL for storing and analyzing time series data and can scale well to an increasing amount of distinct time series without drastically declining performance. Initially, we used InfluxDB but found out it did not scale well for our use case.

Table 7: A record in a time series database consists of a time series identifier, timestamp, metadata, and time series data.

Field	Type	Value
`time_series_id`	ID type	Unique identifier for an individual time series.
`timestamp`	Date-time with timezone	Timestamp with UTC timezone.
`<metadata>`	Data type	One or more metadata values related to the time series identifier.
`<data>`	Data type	One or more observed values.

An instance of a time series database consists of time series tables with a schema as in Table 7. The *time series identifier* (`time_series_id`) is an ID type such as an integer type, and the *timestamp* (`timestamp`) is a date-time with Coordinated Universal Time (UTC) timezone. We recommend using UTC instead of local time zones to avoid problems with daylight saving time and date-time instead of Unix epoch because date-times are human-readable.

In our implementation, a time series table is a TimescaleDB hyper table with appropriate indices for efficient queries and chunks with a proper time interval for improved performance. We set a compression policy to compress data that is older than a specified time to reduce storage and a retention policy for dropping data that is older than a set time to limit data accumulation and for regulatory reasons. We stored all data on a single time series table. In the future, we may experiment with separate tables for MDT and OST data to improve performance since they mainly contain different fields.

3.6 Monitoring client

The monitoring client calls the appropriate command, as explained in Section 3.2, at regular observation intervals to collect statistics. The observation interval should be less than half of the cleanup interval for reliable reset detection. Smaller observation interval increases the resolution but also increase the rate of data accumulation. We used a 2-minute observation interval and a 10-minute cleanup interval.

Initially, we computed the difference between the two counters on the monitoring clients and stored them in the database. Since we used a constant interval, the differences were proportional to the rates explained in Section 3.4, making database queries easy and fast. However, computing the differences in the monitoring clients makes the design more complex and error-prone. Another problem that affected the initial design was a problem with data quality, which we discuss in detail in Section 4.1. Due to some bug or configuration issue, some node names were in the fully-qualified format instead of the short hostname format. Because we assumed that all node names would follow the short hostname format, we parsed the metadata from

the entry identifiers with a short hostname and a fully-qualified hostname as the same. The mistake made us identify two different time series as the same, resulting in wrong values when computing rates. We fixed it by modifying our parser to disambiguate between the two formats. To identify and solve these problems, we switched to collecting the counter values in the database and computing the rates afterward. This approach simplifies the monitoring system and supports variable interval lengths. However, the approach makes queries and analysis more computationally intensive.

The monitoring client works as follows. First, it parses the target and all entries from the output using Regular Expressions (Regex). Then, it creates a data structure for all entries with the timestamp, target, parsed entry identifier, snapshot time, and statistics listed in Table 6. An example instance of a data structure using JavaScript Object Notation (JSON) looks as follows:

```
{
  "timestamp": "2022-11-21T06:02:00.000+00:00",
  "entry_id": "11317854:17627127:r01c01",
  "target": "scratch-OST0001",
  "job_id": 11317854,
  "user_id": 17627127,
  "node_name": "r01c01",
  "executable_name": null,
  "snapshot_time": 1669010520,
  "read": 7754,
  "write": 4284,
  "...": "..."
}
```

Finally, the monitoring client composes a message of the data by listing the individual data structures and sends it to the ingest server via Hypertext Transfer Protocol (HTTP). Our implementation used the InfluxDB line protocol for communication because we designed the code initially for InfluxDB. Due to the scaling problem, we use TimescaleDB and suggest using a more efficient line protocol for communication instead.

3.7 Handling initial entries

We must manually add an initial entry filled with zeros when an entry appears that was not present in the previous observation interval in Jobstats. Otherwise, we lose data from the first observation interval when a new time series appears, or an old one resets. During this thesis, we added the initial entries to the data during the analysis. In the future, we should backfill them into the database to simplify the analysis.

To detect initial entries, the monitoring client must keep track of previously observed identifiers, concatenation of target and entry identifier (`<target>:<entry_id>`), and the previous observation timestamp. When the client encounters an identifier not

present in the previous observation interval, it creates a new instance of a data structure with the new target and entry identifier, the previous timestamp, the missing value for snapshot time, and zeros for statistics. For example, if the previous data structure is the first observation, we have the following:

```
{
  "timestamp": "2022-11-21T06:00:00.000+00:00",
  "target": "scratch-OST0001",
  "entry_id": "11317854:17627127:r01c01",
  "job_id": 11317854,
  "user_id": 17627127,
  "node_name": "r01c01",
  "executable_name": null,
  "snapshot_time": null,
  "read": 0,
  "write": 0,
  "...": "..."
}
```

The monitoring client sends this to the ingest server as explained in Section 3.6.

3.8 Ingest server

The ingest server is responsible for maintaining a connection to the database, listening to the monitoring clients' messages, and parsing them. The server creates an insert statement for every message and sends it into the time series database to add the data.

We did not explicitly set a time series identifier. Instead, we identified different time series as distinct tuples of the target, node name, job ID, and user ID. For login nodes, which do not have a job ID, we generated a unique, synthetic job ID using the executable name and user ID values. Also, we created a unique, synthetic job ID for other entries for which it was missing. We dropped other entries that did not conform to the entry identifier format we had set, as described in Section 3.1.

We could use the concatenated target and entry identifier string as the time series identifier (`<target>:<entry_id>`). It is optional since we can identify individual time series from the metadata alone. However, it might reduce ambiguity about identifying an individual time series.

4 Results

This section presents the results from analyzing the data obtained from monitoring file system usage on the Puhti cluster. Unfortunately, due to issues with data quality from Lustre Jobstats on Puhti, we did not reach all the thesis goals set in Section 1. We could not perform a reliable analysis of the monitoring data from the initial monitoring client and had to discard it. Furthermore, the data quality issue prevented us from developing a reliable, automated analysis and visualization of the real-time monitoring data. Also, we did not have time to gather enough reliable data to correlate file system usage with slowdowns after discarding the initial data. In Subsection 4.1, we discuss the data quality issues regarding the entry identifiers and investigate the entry identifiers from a large sample of consecutive Jobstats outputs.

Later, we obtained new data from the modified monitoring client. However, due to the nature of the issue, we had to discard some of the obtained data. The remaining data seems plausible, and we derive insights from this data and formulate ideas for future work. Regarding the research questions from Section 1, the data indicates that we can identify users who perform more file system operations than others on the cluster, often orders of magnitude more. However, the data quality issues reduce the reliability of the identification. We demonstrate different aspects of this data from compute nodes taken at 2-minute intervals for 24 hours on 2022-10-27. We omitted data from login and utility nodes in this analysis due to a lack of time to verify the correctness of the data. Subsection 4.2 shows counter values and computed rates of three jobs to illustrate different I/O patterns. In Subsections 4.3 and 4.4, we show the total rates of each operation for each Lustre target to visualize larger-scale I/O patterns across the whole data set. Subsection 4.5 shows how fine-grained measurements allow us to break the total rate down into its components which we can use to identify users who perform heavy I/O relative to others. Furthermore, we demonstrate how a minority of users can cause the majority of an I/O load. In Subsection 4.6, we explore ideas for improving the analysis in the future.

We performed explorative data analysis on batches of monitoring data using the Julia language [31], [32] and tabular data manipulation tools from the DataFrames.jl [33] package. To visualize our results, we used Plots.jl [34] with GR [35] as the backend. We obtained a database dump from a selected period into Apache Parquet [36] files, a file format that can efficiently handle and compress tabular data. We limited the file size to be manageable on a local computer by dumping data from different days to separate files. We preprocessed the monitoring data by computing rates from the counter values, inferring initial entries from the data and discarding unwanted data. The processed data consists of rows of timestamp and metadata values and the average rate of each operation from the previous to the current timestamp. The metadata values are categorical; that is, they take values from a fixed set of possible values, such as the names of Lustre targets from Table 3, node names from Table 4, valid user identifiers, and valid job identifiers. We describe the theoretical aspects of computing rates from counters and sums and densities for rates in Appendix C.

4.1 Entries and issues

Table 8: Examples of various observed entry identifiers. The examples show correct entry identifiers, identifiers with missing job IDs, and various malformed identifiers.

Format	Observed entry identifier
Correct	`wget.11317854`
Correct	`11317854:17627127:r01c01`
Missing job ID	`:17627127:r01c01`
Malformed	`wget`
Malformed	`wget.`
Malformed	`11317854`
Malformed	`11317854:`
Malformed	`113178544`
Malformed	`11317854:17627127`
Malformed	`11317854:17627127:`
Malformed	`11317854:17627127:r01c01.bullx`
Malformed	`:17627127:r01c01.bullx`
Malformed	`:1317854:17627127:r01c01`

We found that some of the observed entry identifiers did not conform to the format on the settings described in Section 3.1. Table 8 demonstrates correct entry identifiers, an entry identifier with missing job ID, and different malformed entry identifiers we observed.

The first issue is missing job ID values. Slurm sets a Slurm job ID for all non-system users running jobs on compute nodes, and the identifier should include it. However, we found many entries from non-system users on compute nodes without a job ID. Due to these issues, data from the same job might scatter into multiple time series without reliable indicators making it impossible to provide reliable statistics for specific jobs. The issue might be related to problems fetching the environment variable's value. This issue occurred in both MDSs and OSSs on Puhti.

The second, more severe issue is that there were malformed entry identifiers. The issue is likely related to the lack of thread safety in the functions that produce the entry identifier strings in the Lustre Jobstats code. A recent bug report mentioned broken entry identifiers [37], which looked similar to our problems. Consequently, we cannot reliably parse information from these entry identifiers, and we had to discard them, which resulted in data loss. This issue occurred only in OSSs on Puhti. We obtained feasible values for correct entry identifiers, but we are still determining if the integrity of the counter values is affected by this issue. Next, we look at Figures 3 and 4, which show the number of entries per Lustre target and identifier format for system and non-system users in a sample of 74 Jobstats outputs taken every 2-minutes from 2022-03-04.

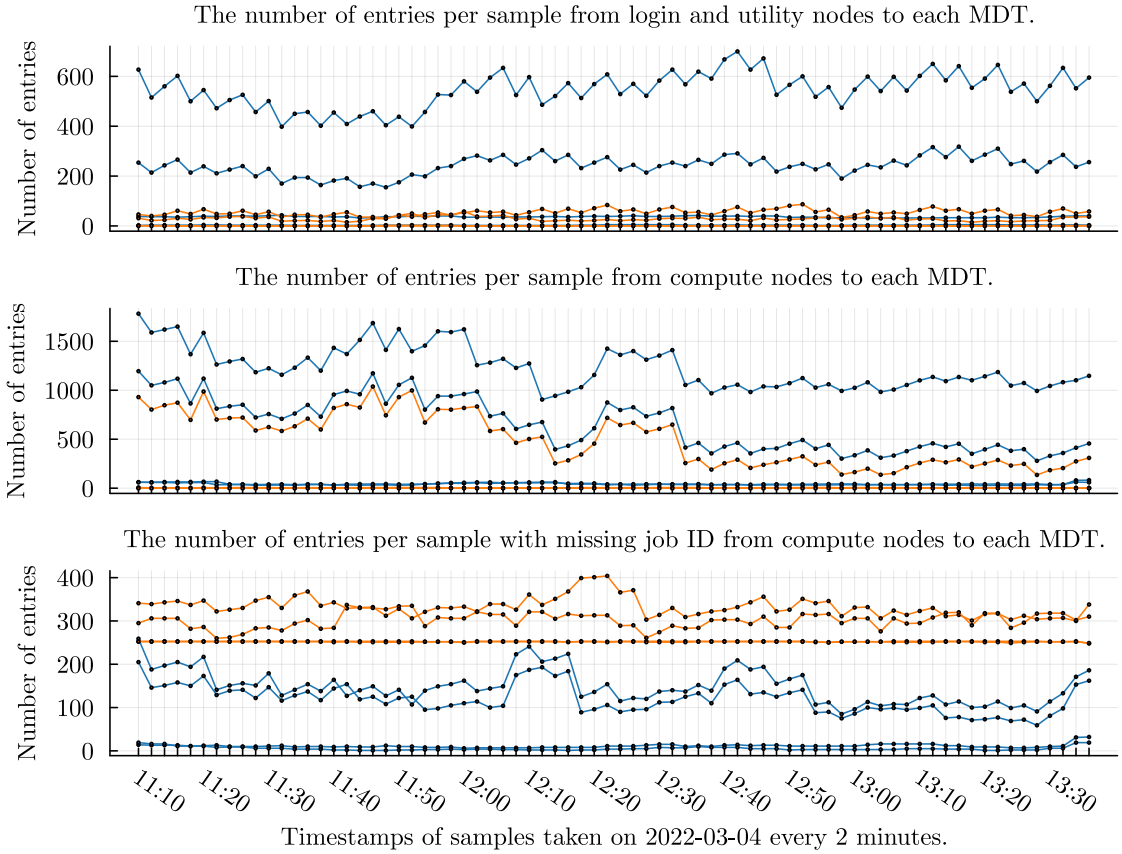

Figure 3: The number of entries for each of the four MDTs during a sample of Jobstats outputs taken every 2 minutes during an interval on 2022-03-04. Each subplot shows a different identifier format; line color indicates non-system users and system users; and each line shows a different MDT for a given user type. We can see many missing job IDs compared to intact ones for non-system users, many entries for system users, and an unbalanced load between MDTs. The first subplot shows the number of correct entries for login and utility nodes, and the second subplot shows them for compute nodes. The third subplot shows the number of missing job IDs on compute nodes, which is substantial compared to the correct identifiers in the second subplot. There are no malformed entries on MDTs.

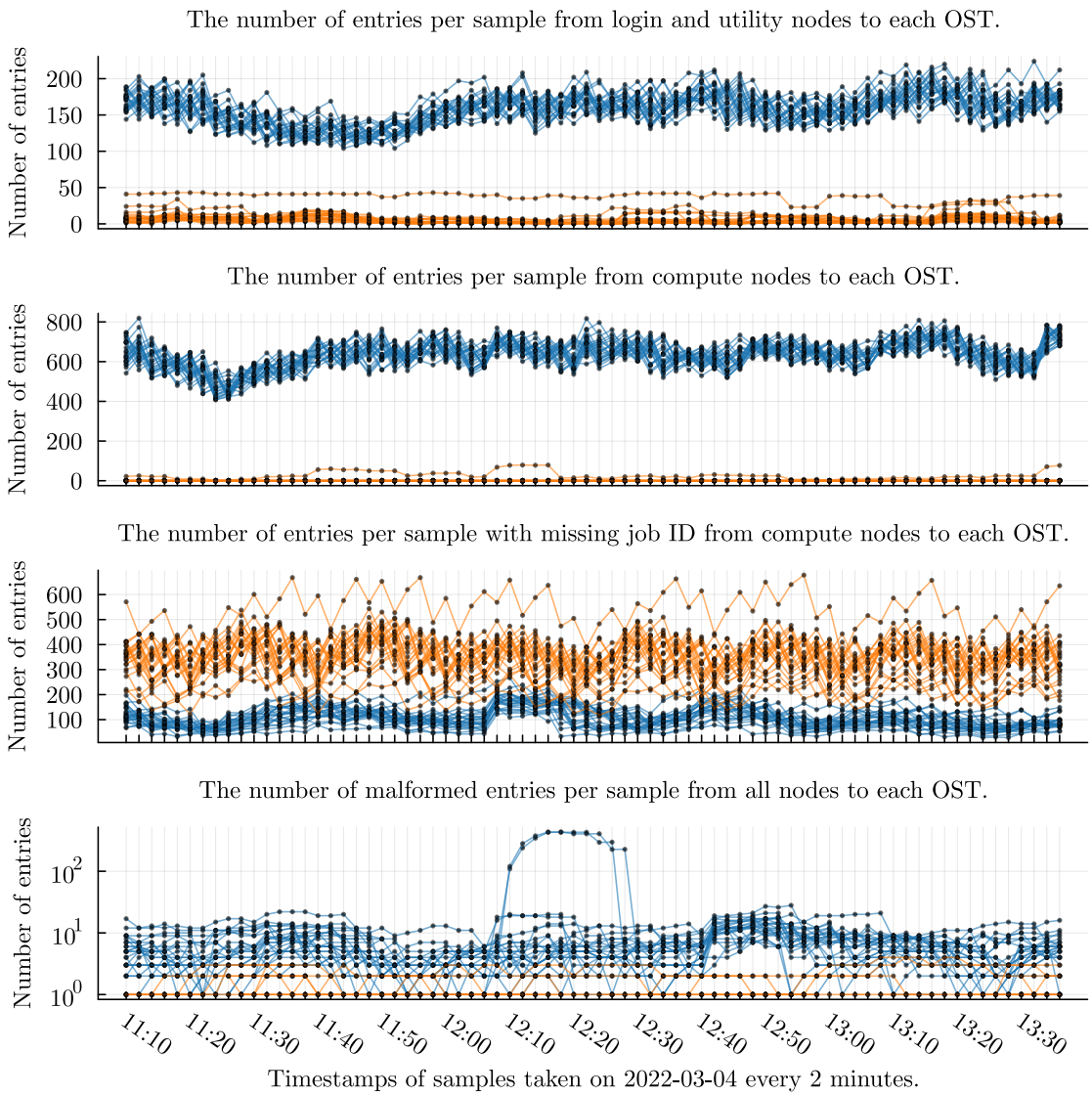

Figure 4: The number of entries for each of the 24 OSTs during a sample of Jobstats outputs taken every 2 minutes during an interval on 2022-03-04. Each subplot shows a different identifier format; line color indicates non-system users and system users; and each line shows a different OST for a given user type. We can see many missing job IDs compared to intact ones for non-system users, many entries for system users, systematic generation of malformed entry identifiers, and a balanced load between OSTs. The first subplot shows the number of correct entries for login and utility nodes, and the second subplot shows them for compute nodes. The third subplot shows the number of missing job IDs on compute nodes, which is substantial compared to the correct identifiers in the second subplot. The fourth subplot shows the number of malformed identifiers for all nodes. We can see that Jobstats on Puhti systematically produce missing job IDs and malformed identifiers. Furthermore, there is a large burst of malformed identifiers from 12:06 to 12:26, indicating that in some conditions, Jobstats produces large amounts of malformed identifiers.

We can see that only two of the four MDTs handle almost all of the metadata operations. Of the two active MDTs, the first one handles more operations than the second one, but their magnitudes correlate. The load across MDTs is unbalanced because MDTs are assigned based on the top-level directory. Each MDT is assigned to a different storage area, such as Home, Projappl, or Scratch, covered in Section 2.5, and the load is unbalanced because the usage of these storage areas varies. On the contrary, the load across OSTs is balanced because the files are assigned OSTs equally with a round-robin policy unless the user explicitly overwrites this policy.

We see that the number of entry identifiers with missing job IDs is substantial compared to the number of correct identifiers for non-system users. We also observe that Jobstats systemically generates malformed identifiers on the OSSs, and in certain conditions, maybe due to heavy load on the OSS, it can create many of them.

Entries from non-system users are the most valuable ones for analysis. However, we see that system users generate many entries. By inspecting the data, we found that only two system users, root and job control, generate these entries in Puhti. Furthermore, most of these entries contain little valuable information; for example, many have a single `statfs` operation. Also, entries from system users usually did not have a job ID as their processes do not run via Slurm, although sometimes they do have a job ID.

Regarding data accumulation, each entry corresponds to one row in the database. Therefore, reducing the number of entries reduces storage size and speeds up queries and the analysis. We should discard or aggregate statistics of system users to reduce the accumulation of unnecessary data. In general, correct entry identifiers would reduce unnecessary data accumulation.

4.2 Counters and rates

We explore I/O patterns at the most fine-grained level by visualizing data from three selected jobs. Figures 5, 6, and 7 show us different patterns of counter values and rates for `write` operations for different jobs during 24 hours of 2022-10-27. The figures demonstrate the fine-grained nature of the monitoring data and entry resets discussed in Section 3.

The x-axis displays time and the y-axis displays the accumulated amount of operations for counters and the operations per second for the rate. Each line displays operations from one Lustre client to one Lustre target. The figures in this subsection display a single node job; thus, each line shows `write` operations from the same compute node to a different OST. We say that a job is active during a period that performs any file system operations; otherwise, it is inactive.

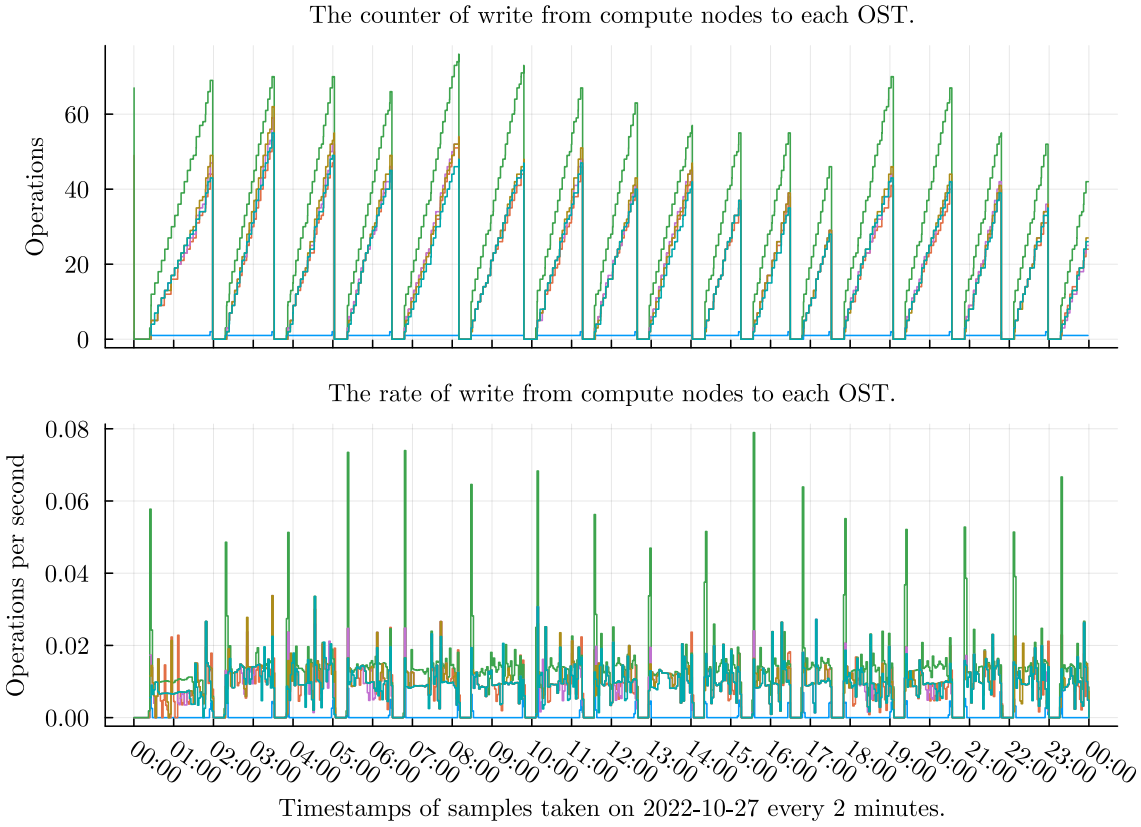

Figure 5: The counter and rate of `write` operations from one job on a single compute node. The top subplot shows the counter values, and the bottom subplot shows the rates computed from the counter values in the first plot. The subplots share the same x-axis. The counter values follow a typical saw-tooth pattern for almost linearly increasing counter values that reset periodically due to inactivity. In the active periods, we see a higher write amount of writes in the beginning, then quite near constant write rate until the job becomes inactive. The lines follow a similar pattern indicating that the job performs a similar write pattern for each OST except for the ones whose rate is near zero.

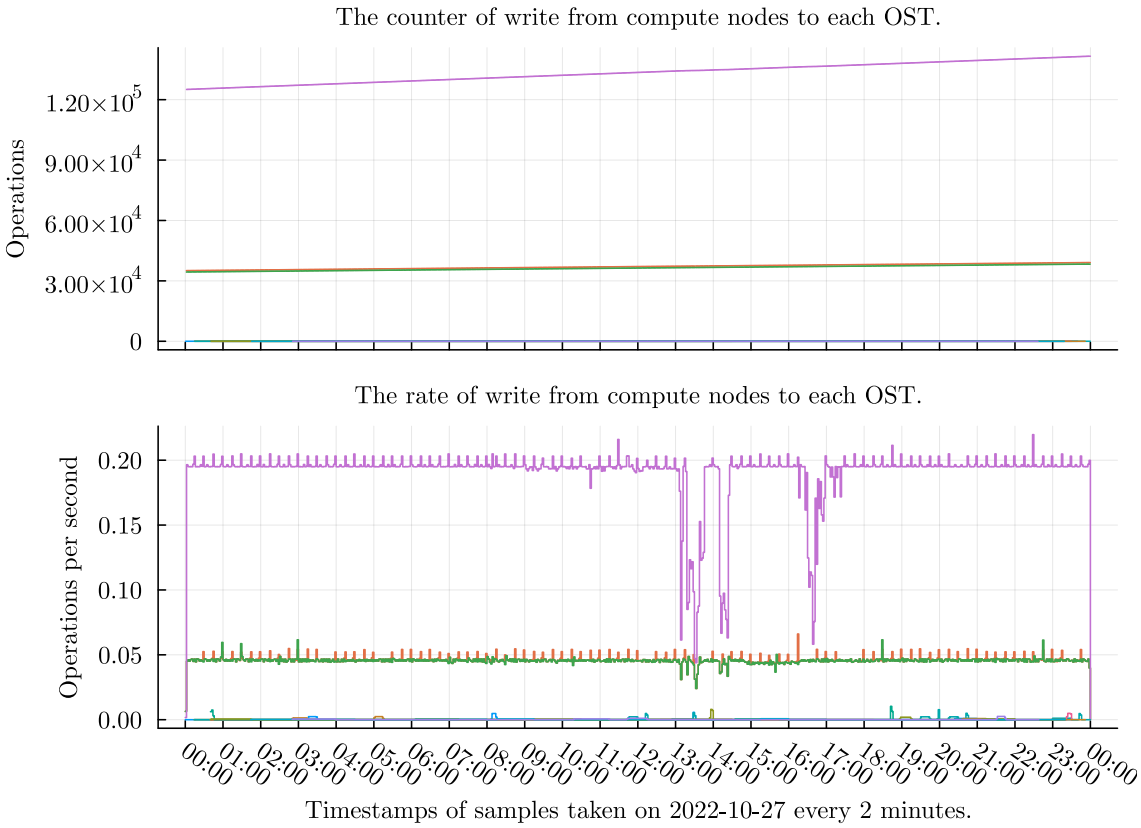

Figure 6: The counter and rate of `write` operations from one job on a single compute node. The top subplot shows the counter values, and the bottom subplot shows the rates computed from the counter values in the first plot. The subplots share the same x-axis. The counter values increase almost linearly, indicating that the job performs writes consistently during the whole period. The rate over the whole period is almost constant with some small fluctuations. We can see that the job performs almost 75% of the operations to one OST, almost 25% to two other OSTs, and almost none to the others.

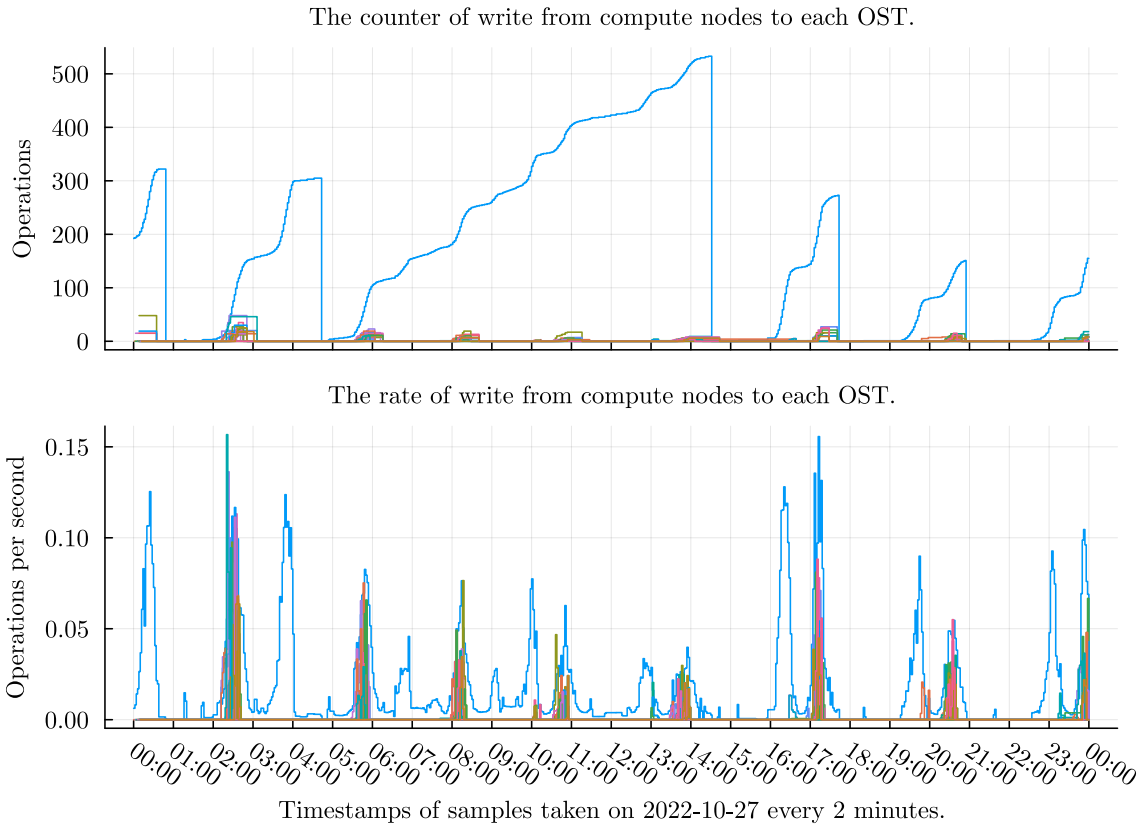

Figure 7: The counter and rate of `write` operations from one job on a single compute node. The top subplot shows the counter values, and the bottom subplot shows the rates computed from the counter values in the first plot. The subplots share the same x-axis. One of the counter values increases in a wave-like pattern that resets periodically; the other counter seems to increase in a burst-like manner for short periods before resetting. By looking at the rates, we can see that the rates fluctuate for all OSTs. Furthermore, most of the time, the job performs writes to one OST and sometimes to multiple OSTs in a burst.

4.3 Metadata rates

We explore trends of different metadata operations by visualizing the data we obtained from metadata servers. Figures 8, 9, 10, 11, 12, 13, and 14 show the total rates for all operations from compute nodes to each of four MDTs during 24 hours of 2022-10-27. Comparing loads between MDTs is not interesting because Lustre assigns each storage area to one MDT. We use a logarithmic scale due to large variations in the magnitude of the rates. Because some rates in the plots are zero, but the logarithmic axis does contain zero, we omit zeros from the plot. The plots share the same x-axis, making them easier to compare.

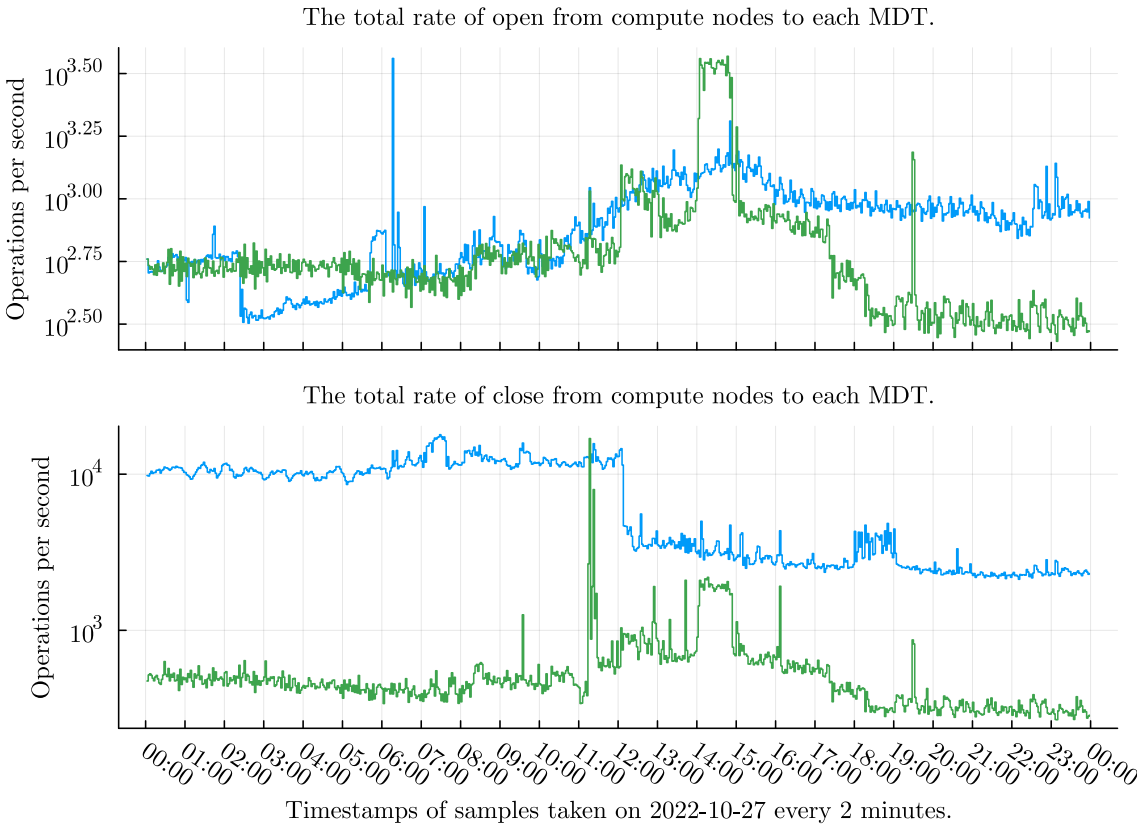

Figure 8: Total rates of `open` and `close` operations from compute nodes to each MDT. We can see that the `open` rate is quite consistent, and `close` has a large drop around 12.00. Large changes in rates are usually caused when a single job that performs heavy I/O stops. We can also see that the rate of `close` is greater than the rate of `open`. It is impossible to perform more `close` and `open` operations because we always need to open a file before closing it. We suspect that Lustre clients cache some operations, and Jobstats does not include cached operations in the statistics, explaining the differing `close` and `open` rates.

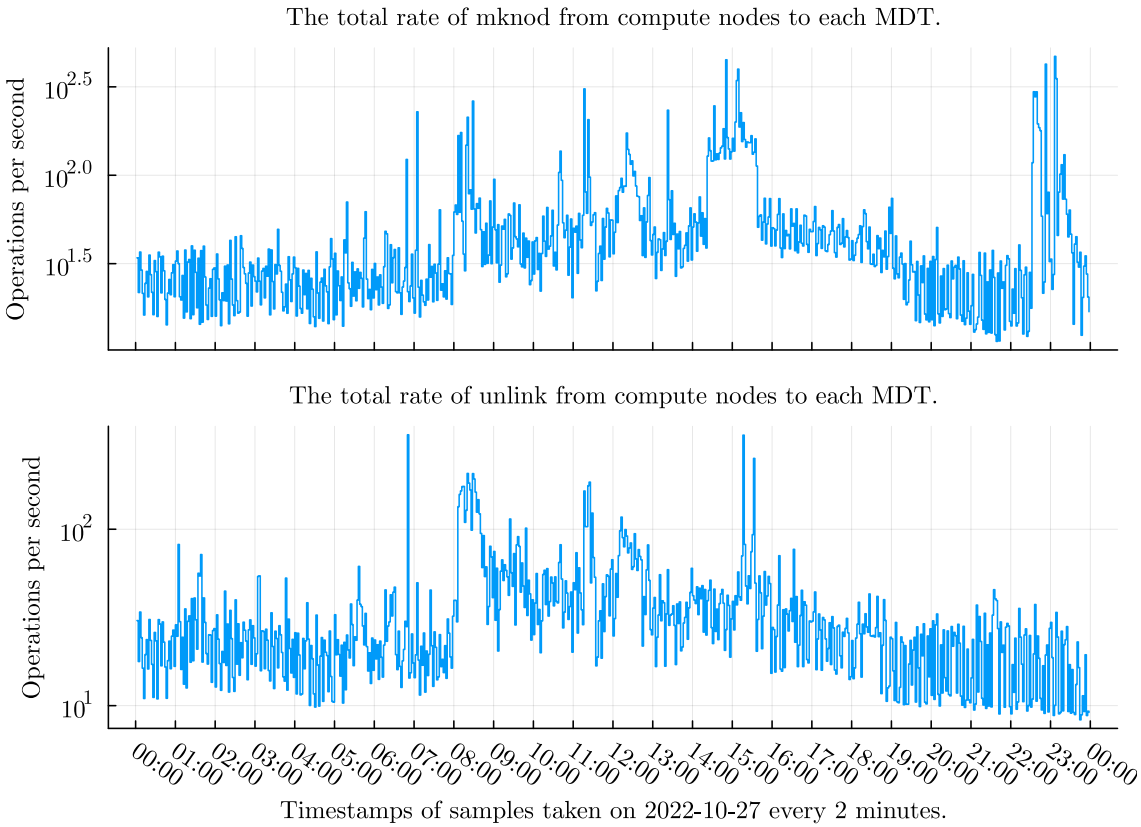

Figure 9: Total rates of `mknod` and `unlink` operations from compute nodes to each MDT. We can see the file creation rate by looking at the rate of `mknod` operations and the file removal rate by looking at the rate of `unlink` operations. The values in these plots do not show large variations. Elevated file creation and removal rates may indicate the creation of temporary files on the Lustre file system, which is undesirable.

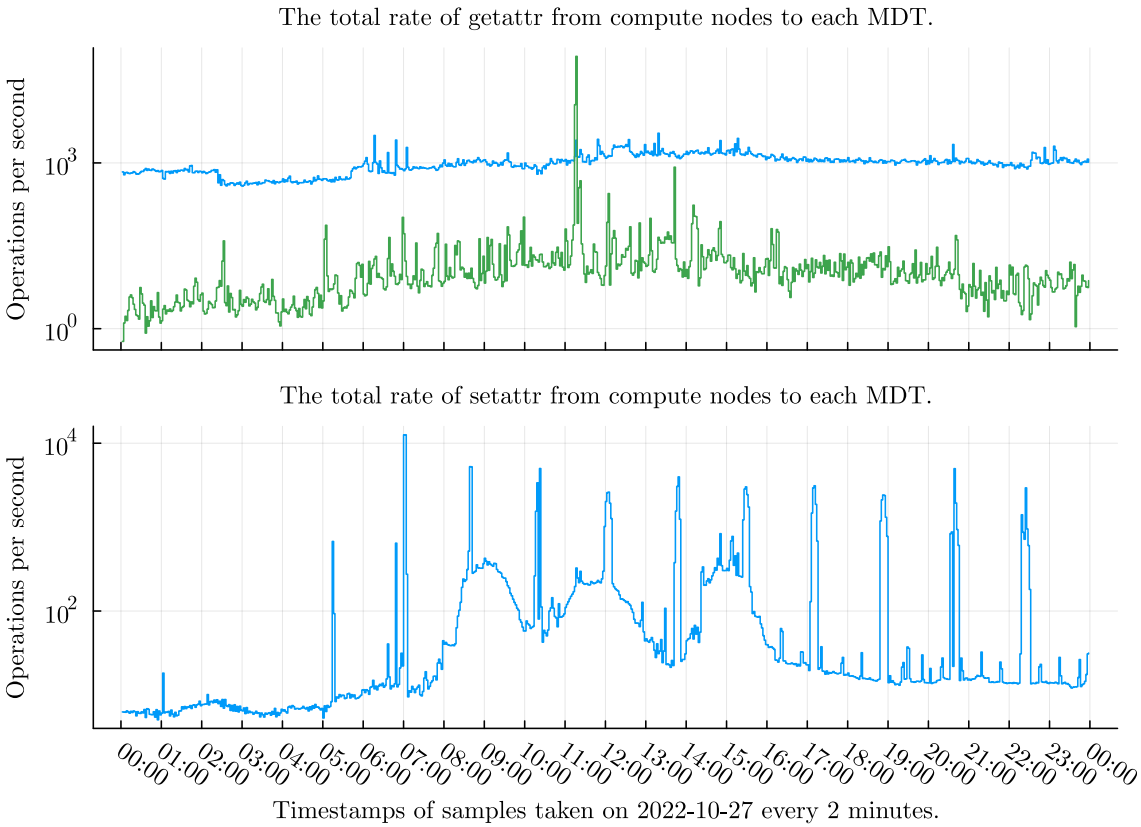

Figure 10: Total rates of `getattr` and `setattr` operations from compute nodes to each MDT. We can see that the `getattr` rate is consistent, but the `setattr` has large spikes. These rates indicate the frequency of querying and modifying file attributes, such as file ownership, access rights, and timestamps. For example, these rates may be elevated due to the creation of temporary files. In Figure 20, we inspect the contribution of different users to the `setattr` rate on MDT.

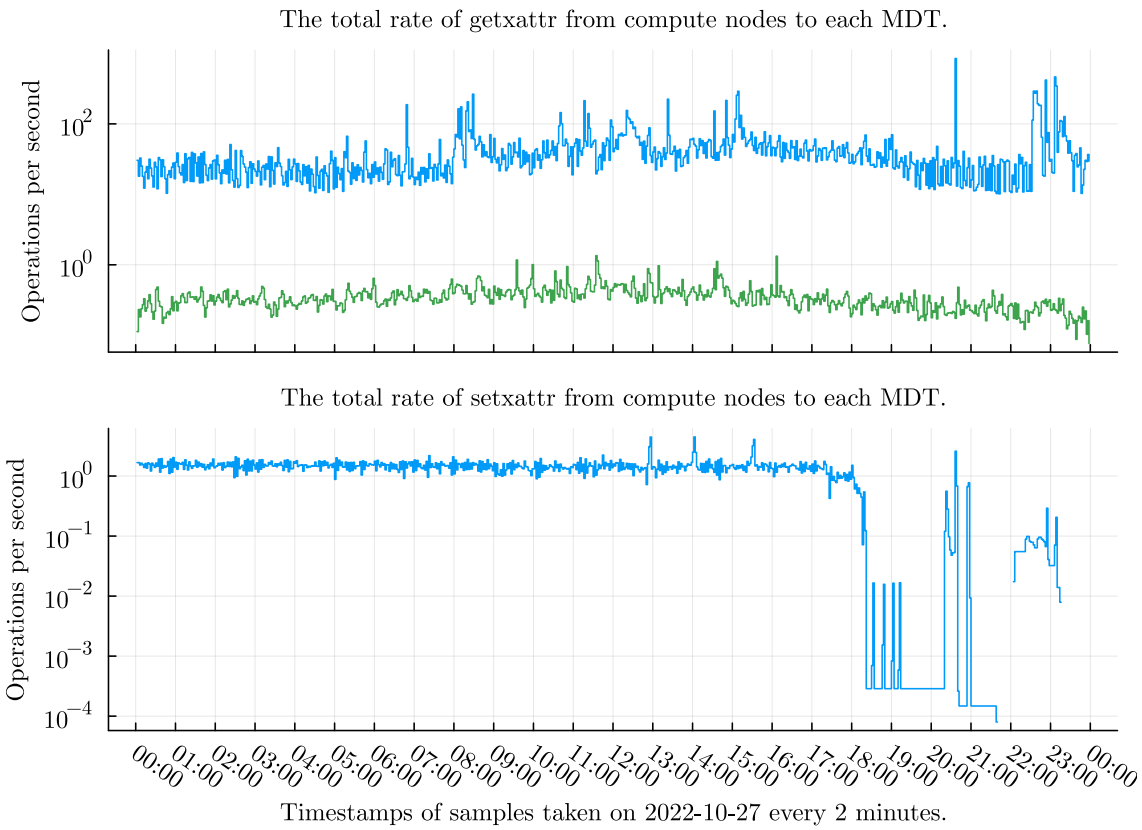

Figure 11: Total rates of `getxattr` and `setxattr` operations from compute nodes to each MDT. We can see that `getxattr` rates are consistent throughout the period. The magnitude of the `setxattr` rate is small, only a couple of operations per second, from 00.00 to 18.00, after which the rate falls to near zero.

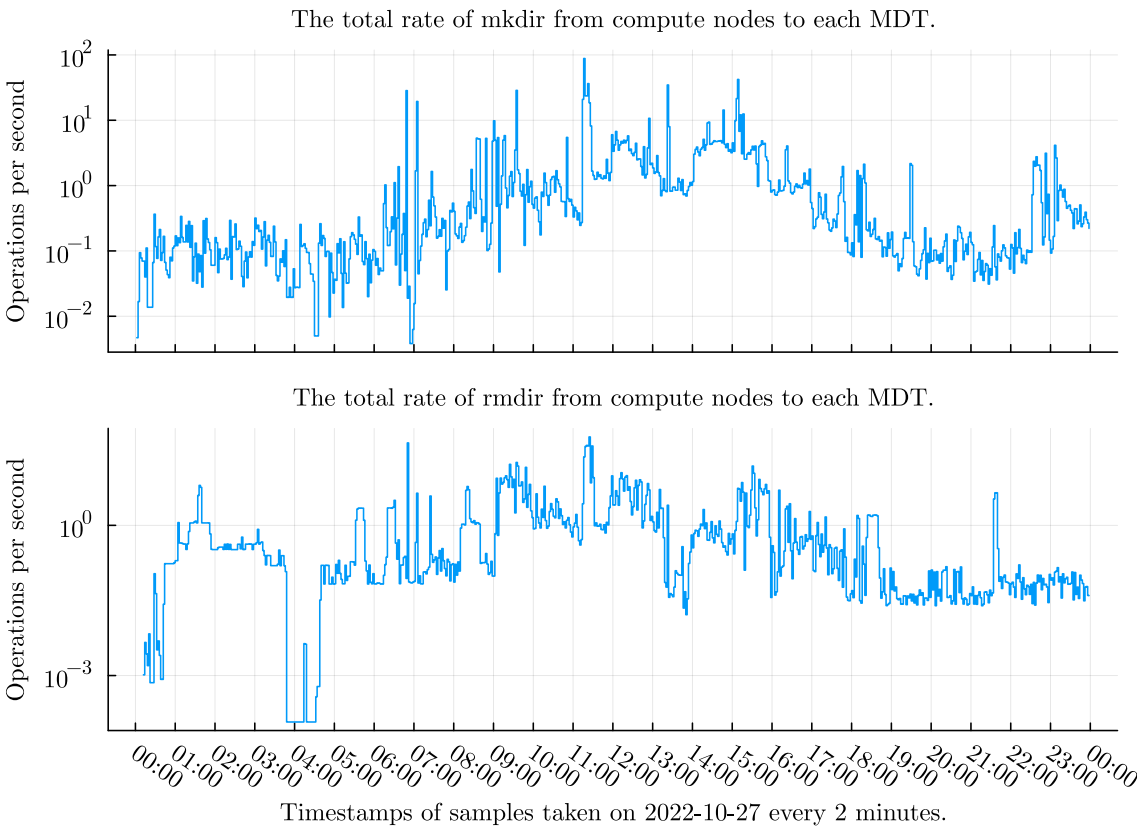

Figure 12: Total rates of `mkdir` and `rmdir` operations from compute nodes to each MDT. Both rates are consistent throughout the period, and the magnitude is relatively small, for example, compared to file creation and removal in Figure 8.

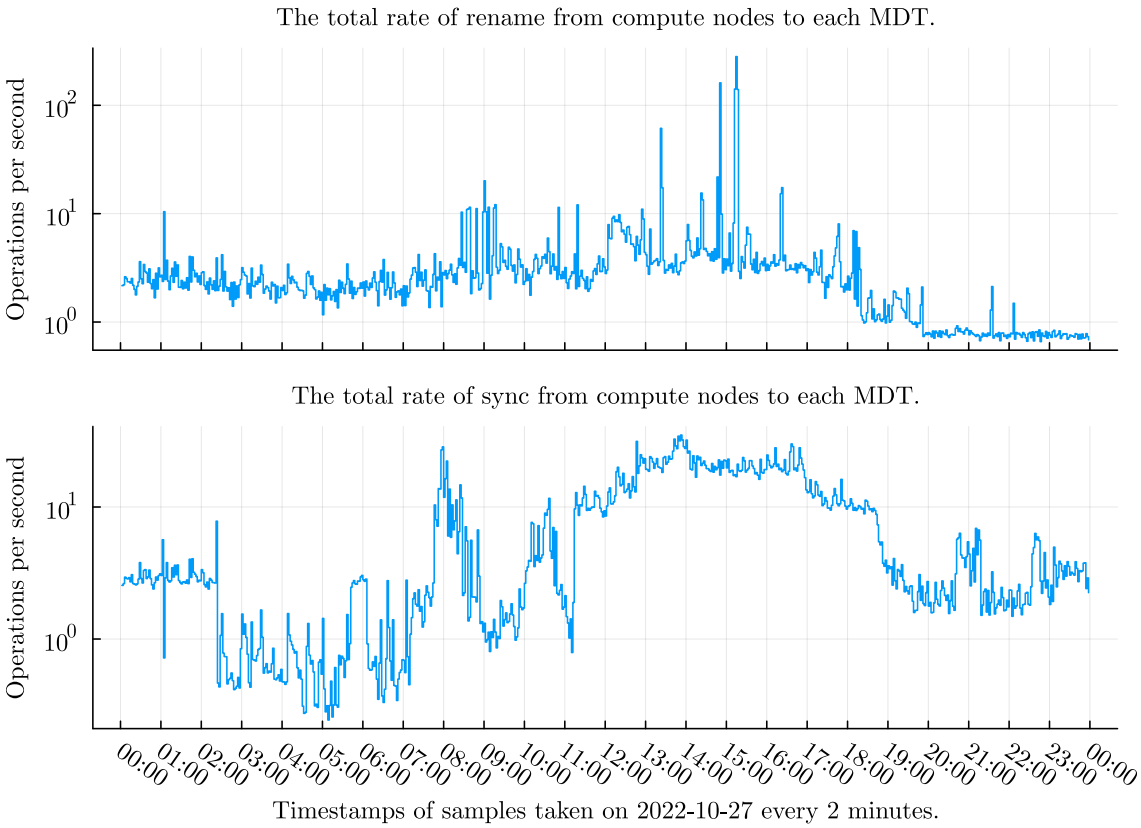

Figure 13: Total rates of `rename` and `sync` operations from compute nodes to each MDT. Both rates are consistent throughout the period, and the magnitude is relatively small.

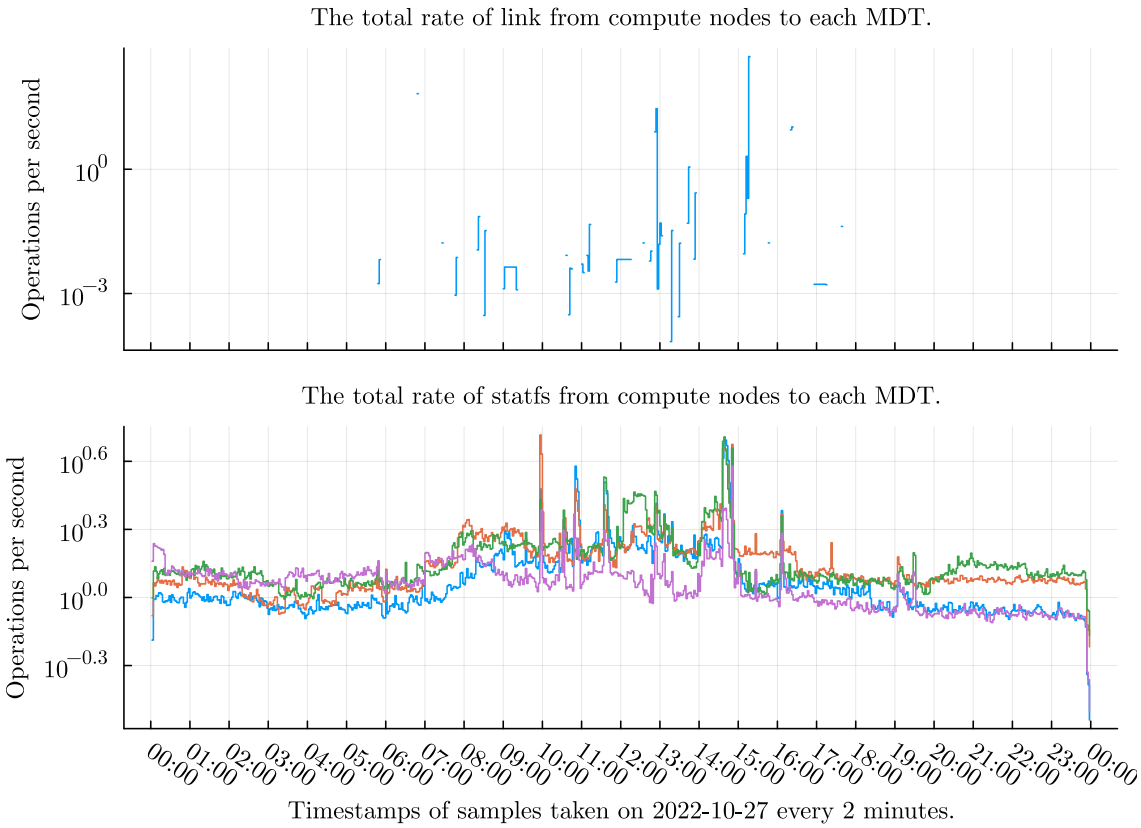

Figure 14: Total rates of `link` and `statfs` operations from compute nodes to each MDT. We can see almost no `link` operations; hence the line is sparse. On the contrary, `statfs` operations seem consistent and appear on all MDTs.

4.4 Object storage rates

We explore trends of different object storage operations by visualizing the data we obtained from object storage servers. Figures 15, 16, 17, 18, and 19 show the total rates of all operations from compute nodes to each of 24 OSTs during 24 hours of 2022-10-27. Since there are 24 OSTs, we can compare the variation of rate between OSTs and across time. By default, Lustre aims to balance the load between OSTs by assigning files to them equally. Significant differences between the rates of different OSTs mean that the load is unbalanced. An unbalanced load may lead to congestion when others try to access the same OST. We use a logarithmic scale due to large variations in the magnitude of the values. All plots share the same x-axis, making them easier to compare.

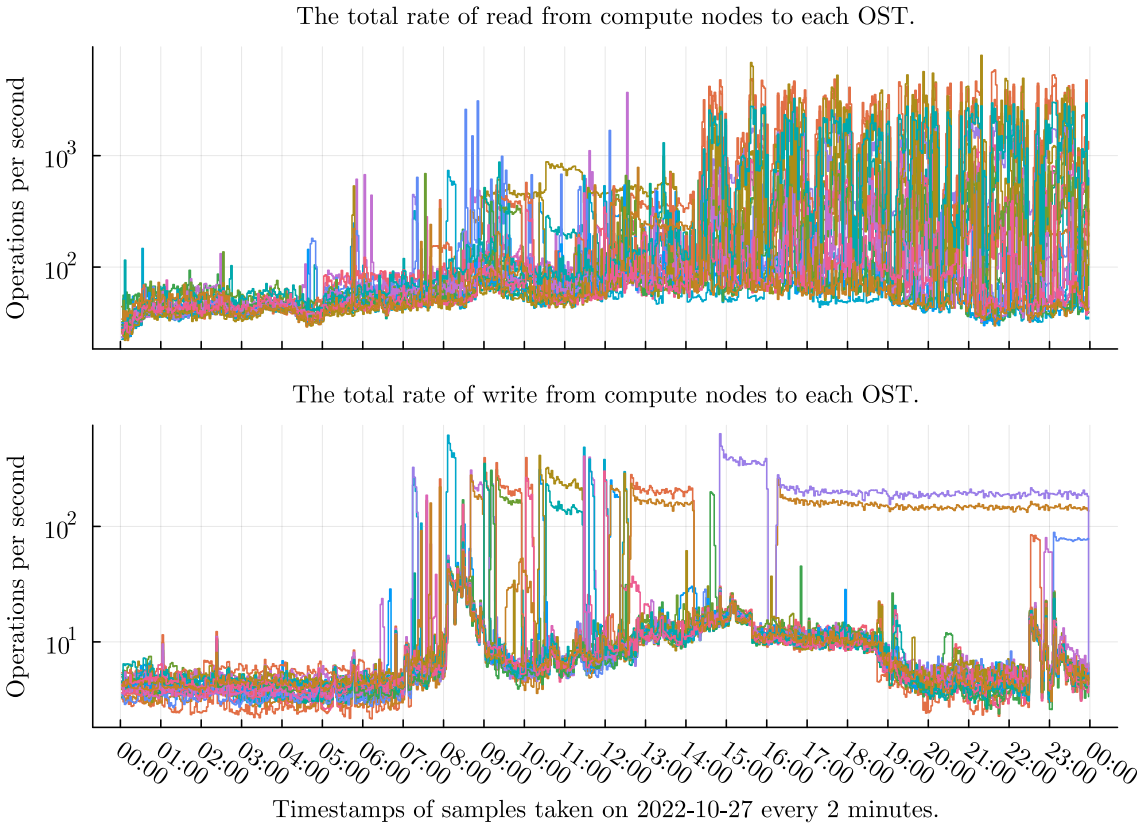

Figure 15: Total rates of `read` and `write` operations from compute nodes to each OST. In the top subplot, we can see that the rate of `read` operations does not vary across OSTs from 00:00 to 07:00, but after 07:00, the variance increases. Later in Figure 21, in the next section, we explore the components of the total read rate on OST0001. The bottom subplot shows us the rate of `write` operations which is smaller in magnitude. The rate of most OSTs does not vary much, but individual OSTs deviate from the base load by order of magnitude. The total write rate consists of individual `write` rates from many users' jobs, some of which are similar to the ones we saw in Figures 5, 6, and 7.

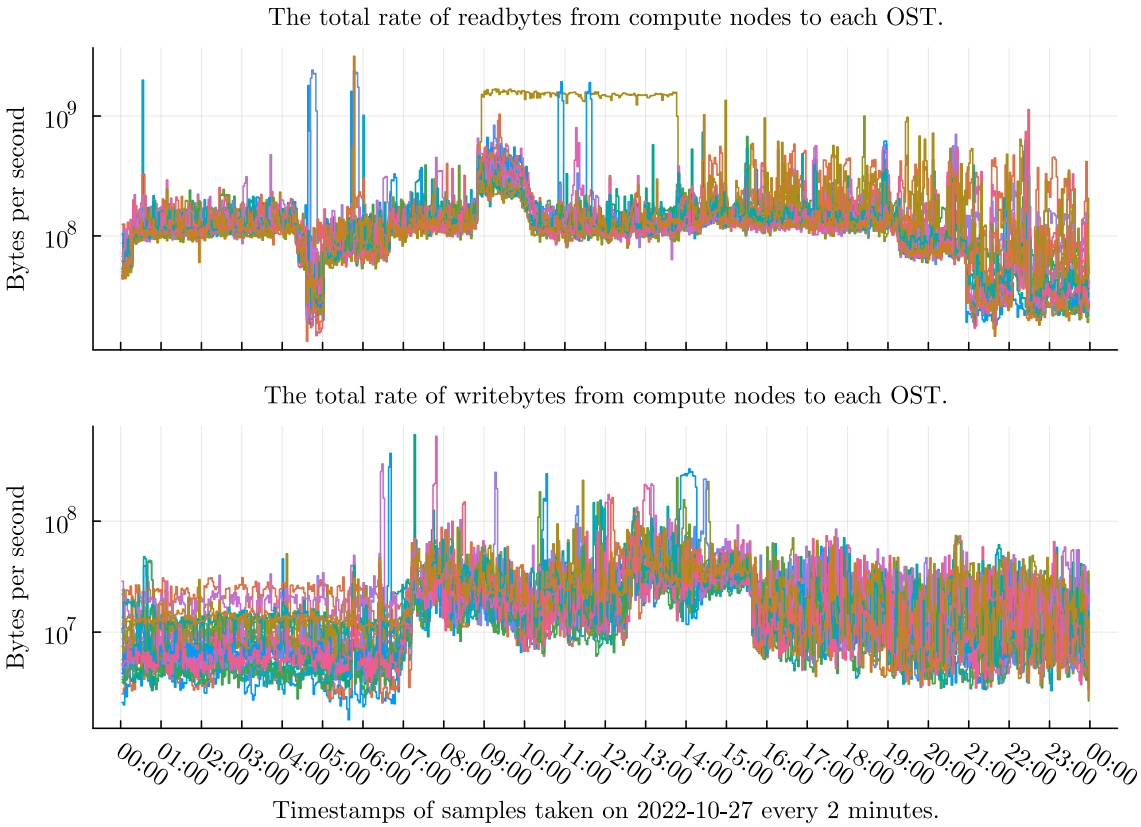

Figure 16: Total rates of `readbytes` and `writebytes` operations from compute nodes to each OST. We can see that the `readbytes` rate is mostly balanced over OSTs and consistent over the period, expect a few spikes and one long, heavy load from 9.00 to 14.00 to a single OST, which we inspect in more detail in Figure 22 in the next section. Heavy load on a single OST indicates that a large file is not properly striped over multiple OSTs. The `writebytes` rate is balanced over OSTs and consistent over the period with only a few spikes.

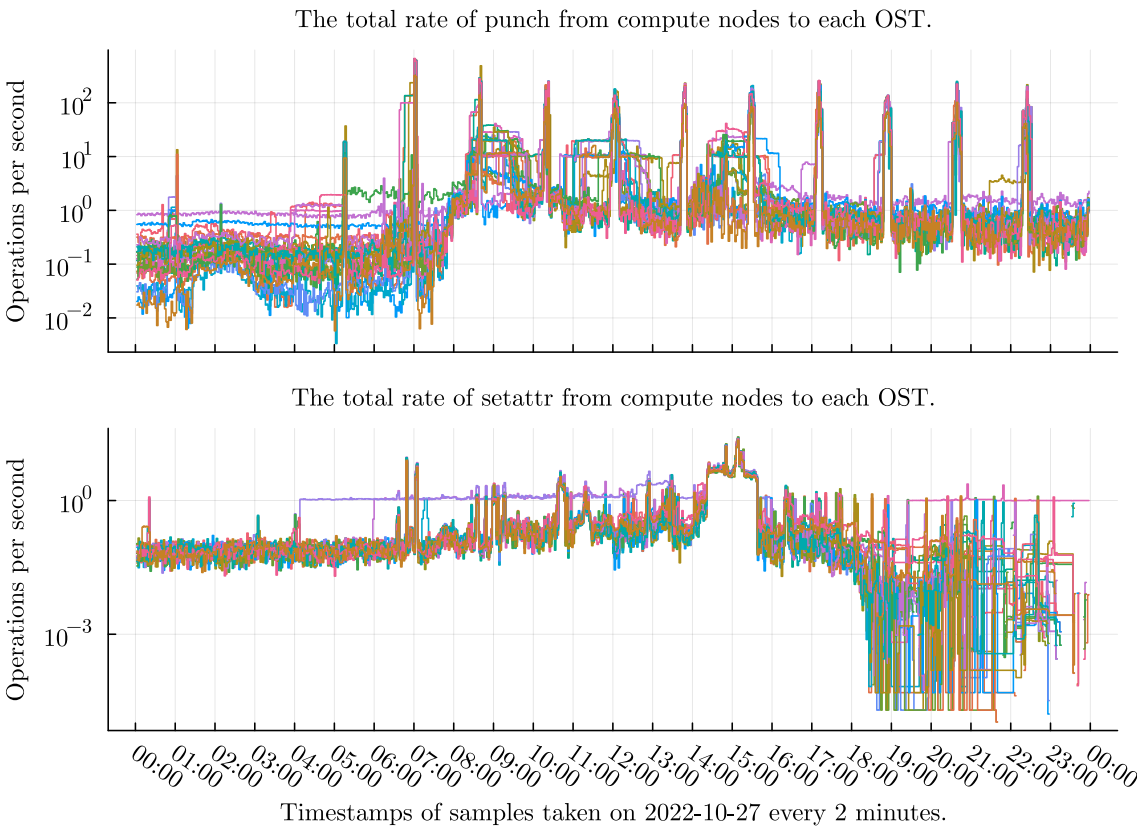

Figure 17: Total rates of `punch` and `setattr` operations from compute nodes to each OST. We can see that the `punch` rate spikes periodically simultaneously for all OSTs. The `setattr` rate is very low and does not exhibit interesting patterns.

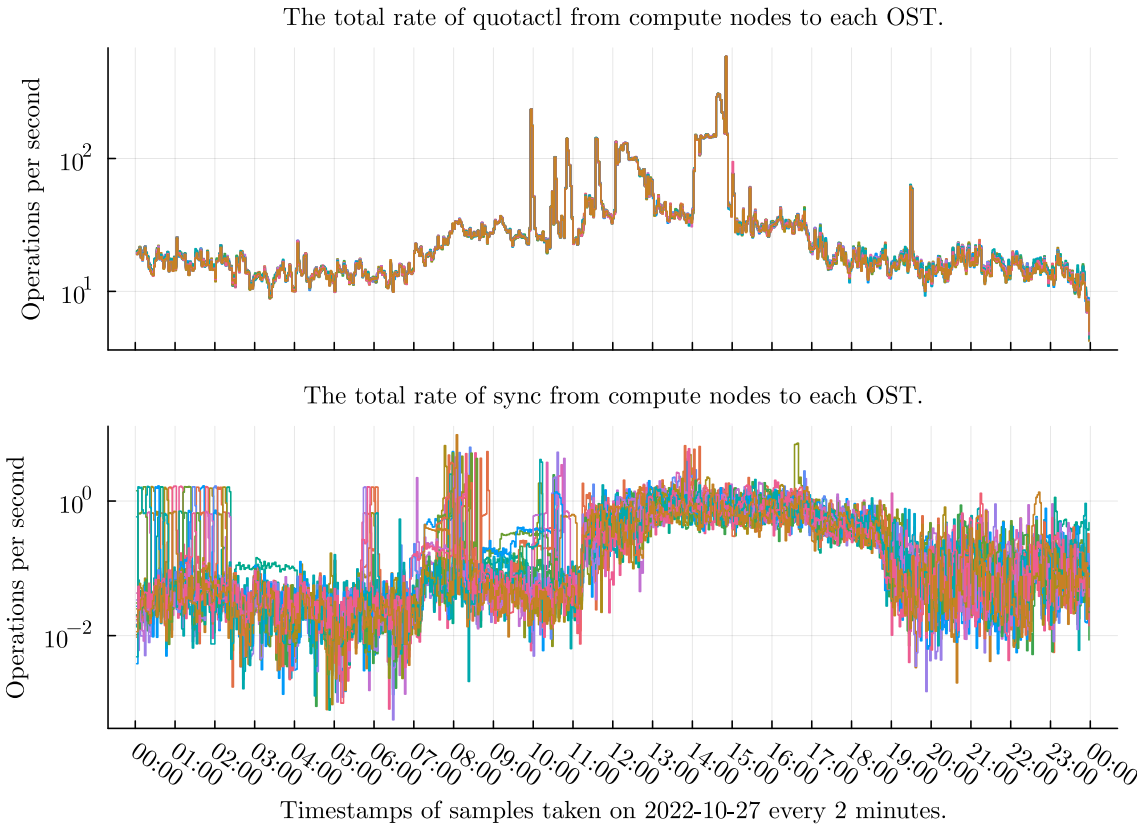

Figure 18: Total rates of `quotactl` and `sync` operations from compute nodes to each OST. We can see that the `quotactl` rate looks well balanced between OSTs, compared to other rates, such as `read` and `write` rates in Figure 15, likely because it does not operate on specific files, users cannot easily perform a different amount of `quotactl` operations of different OSTs. The `sync` rate is very low and does not exhibit interesting patterns.

Figure 19: Total rates of `getinfo` and `setinfo` operations from compute nodes to each OST. Both rates are very low and do not exhibit interesting patterns.

4.5 Identifying causes of changes in I/O trends

As discussed in Section 3, fine-grained file system usage monitoring produces data containing multiple time series. We identify each time series by a unique combination of metadata values. As previous results demonstrate, we can inspect the data in a high-level view as a sum of many time series or in a low-level, fine-grained view as individual time series that form the sum's components. Our goal is to find changes in trends from the high-level view and the causes for the changes in trends from the low-level view. We especially care about identifying causes for large increases in the I/O rates.

Because the fine-grained view consists of many time series, we compute a *density* to obtain information about the composition of the sum at each timestamp. Density tells us how many time series has a value in a specific range, called a *bucket*, at a particular time but omits information about individual time series. Importantly density allows us to distinguish differences, such as whether an increase in sum rate is due to a small number of large or a large number of small components.

We explore a part of the data set visually using Figures 20, 21, and 22. We demonstrate a process of identifying users who cause large relative increases in I/O rates. Furthermore, these figures show that a minority of users can perform most of the I/O load at a given time. Each figure consists of three subplots. The top subplot demonstrates the high-level trend by showing the total rate, the sum of rates from all users. We use it to identify periods where trends change, specifically the large relative increases. The middle subplot demonstrates a more fine-grained view by showing the rate of each user, that is, the components of the total rate from the top subplot. We analyze them to understand user-level causes of changes in the total rate. The bottom subplot shows the density computed from the values seen in the middle subplot. We use a density plot to analyze how many users perform the I/O rate in a specific range during a specific time. Specifically, we look at the bucket at the top of the plot to see how many users perform large I/O rates relative to others during an increase in the I/O rate. We identify the users by filtering the data set using the period and value range as conditions obtained from the subplots. Then, we look at the unique set of user metadata values from the filtered data.

We use a logarithmic scale for the density due to the significant variations in the magnitude of the values and omit zeros from the plot. We plot densities as heatmaps consisting of time on the x-axis, buckets on the y-axis, and color on the z-axis. In the density plot, lighter color indicates more users, a darker color indicates fewer users, and no color indicates zero users. The resolution of the density plots, that is, the upper and lower bounds of the buckets, uses a logarithmic scale in base 10.

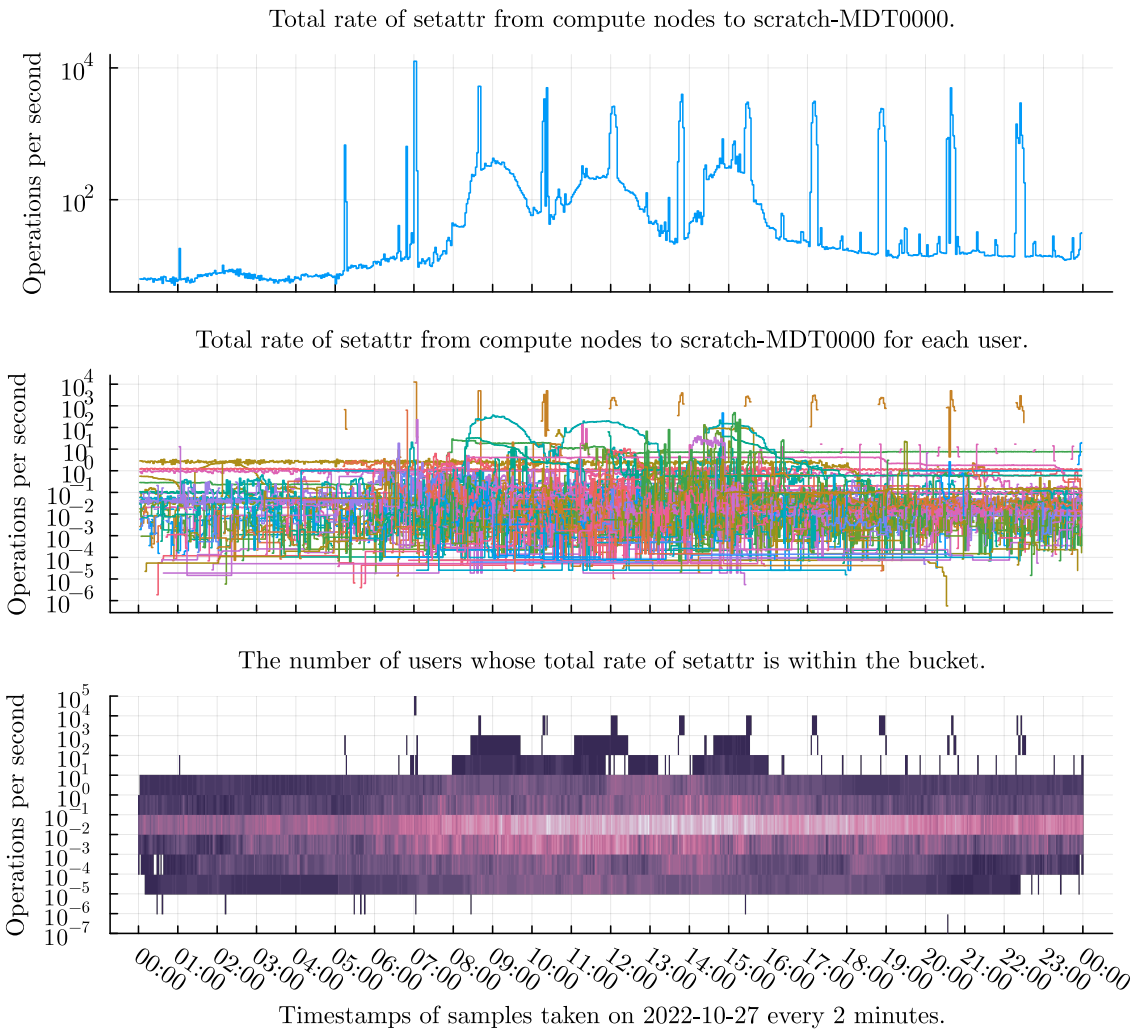

Figure 20: Rates of `setattr` from compute nodes to `scratch-MDT0000` during 24 hours of 2022-10-27. During the period, only one user performs above 10^3 operation per second, seen as the large spikes relative to other users. Furthermore, eight users perform within 10^2 and 10^3 operations per second, compared to the 310 users who perform a rate less than 10^2.

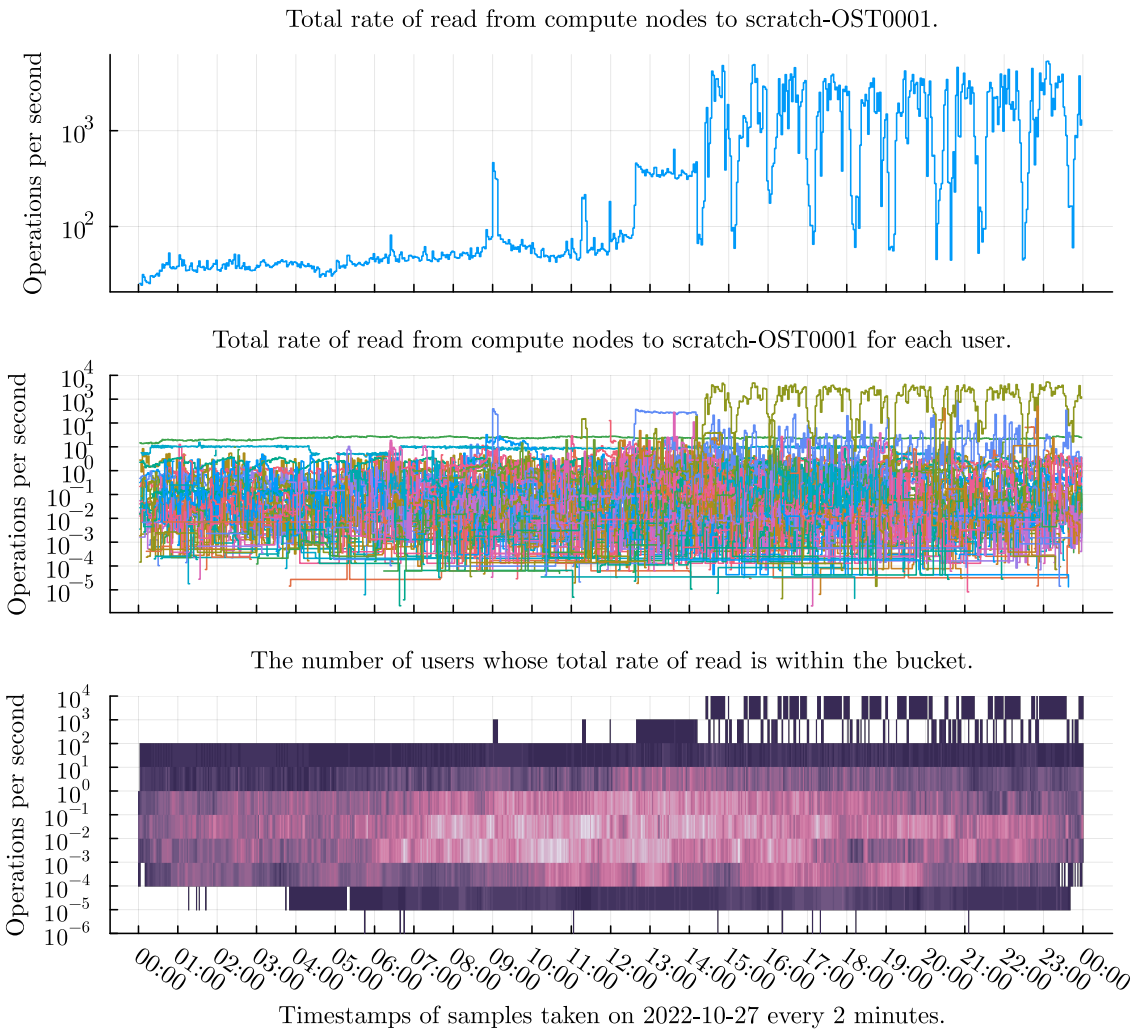

Figure 21: Rates of `read` from compute nodes to `scratch-OST0001` during 24 hours of 2022-10-27. During the period, only one user performs above 10^3 operation per second, seen as the bursts starting after 14:30. Furthermore, seven users perform within 10^2 and 10^3 operations per second, compared to the 285 users who perform a rate less than 10^2.

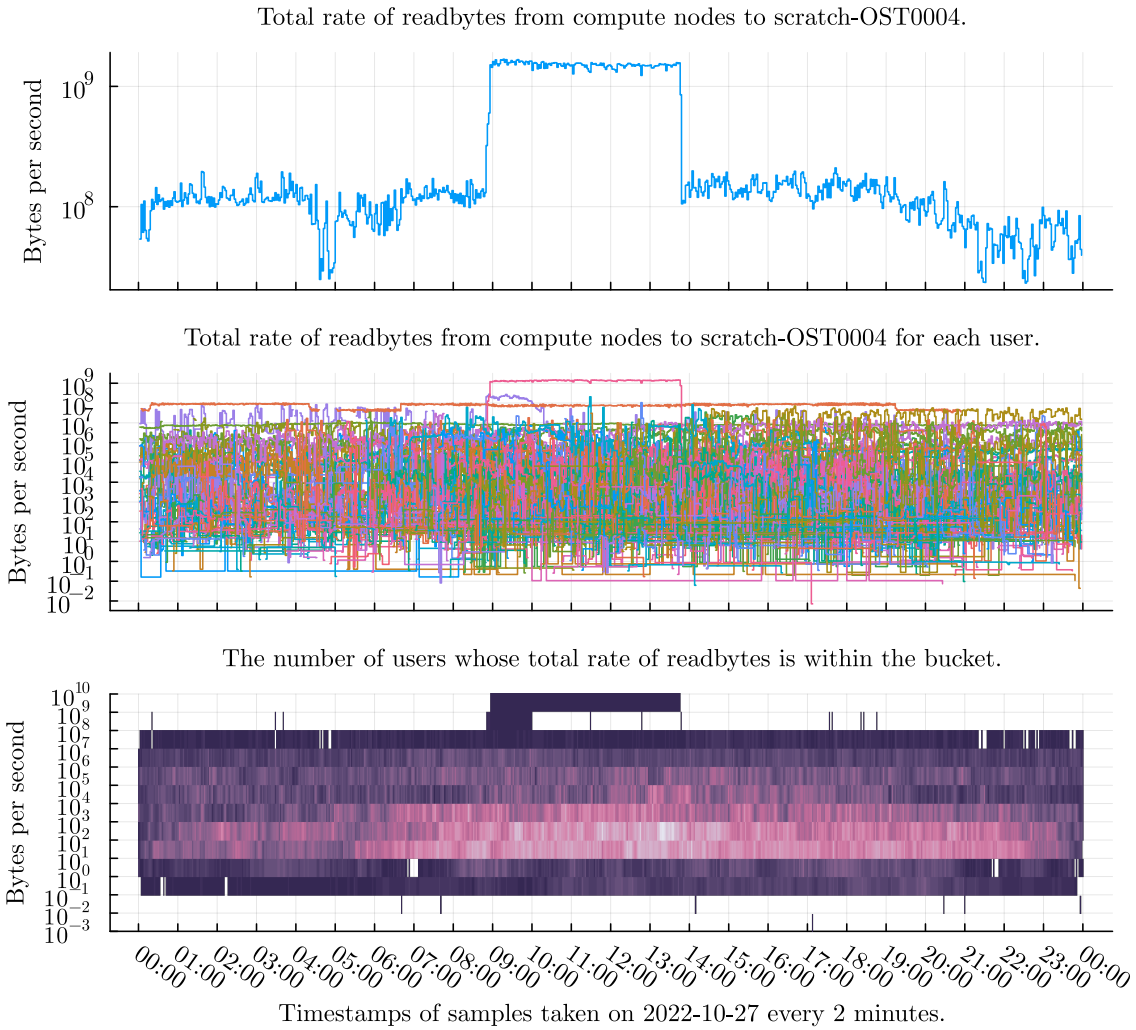

Figure 22: Rates of `readbytes` from compute nodes to `scratch-OST0004` during 24 hours of 2022-10-27. During the period, only one user performs above 10^9 bytes (gigabytes) per second, seen as the long, intense burst between 9:00 and 14:00. Furthermore, five users perform within 10^8 and 10^9 bytes (hundreds of megabytes) per second, most from a burst from 9:00 to 10:00. Compared to the 304 users who perform a rate less than 10^8.

4.6 Future work

In the future, we aim to gather reliable data over a longer period by using a patched version of Lustre Jobstats or a different monitoring tool. We should verify data correctness by running jobs with known I/O work and comparing them with the monitoring data. We can use the monitoring data to perform more extensive analysis and to develop automatic, real-time analysis methods. Here are some ideas for future analysis methods.

We analyzed operations independently. However, we should analyze total I/O rates by combining different operations of the same unit using a linear combination with appropriate weights. For example, we can combine the rates of metadata operations to a specific metadata server or target to analyze its total load.

We identified trends manually using data visualization. However, we should identify trends automatically from a time series by using a causal, impulse response filter such as a moving average with a finite time window. Furthermore, we can identify changes in trends across different timescales by filtering a time series with different time window lengths and comparing the filtered time series. The intersections between two filtered time series indicate points in time where the trends change. For example, given a time series such as a total rate of an operation or a linear combination of operations, we could compare its moving average with a short, ten-minute time window against a long, one-day time window to identify transient changes against a longer trend.

Furthermore, we can use data from login and utility nodes when we have reliable data. Also, we can use Slurm accounting data, such as project and partition information, as metadata for the analysis. For example, we can use project information to identify if members of a particular project perform heavy I/O relative to others and partition information to identify if jobs on a particular partition perform heavy I/O relative to others.

We should compare the file system usage data with file system performance metrics to identify what kind of usage causes a slowdown in the cluster. Currently, we have probes that periodically measure the parallel file system performance. We should measure the correlation between these measurements and file system usage data. Also, it might be interesting to collect and analyze latency values from Lustre Jobstats to see if they provide useful information.

Finally, analyzing monitoring data as a batch is fine for exploring the data and experimenting with different analysis methods. However, to build a real-time monitoring system, we must process and analyze the monitoring data as a stream. We must implement our analysis methods to stream computing, that is, computing the rates and analytics on new data as soon as it arrives from a monitoring client. We should also build real-time visualization and reporting of the analysis results.

5 Conclusion

The challenges in this thesis work were primarily practical engineering rather than deep theoretical ones. We collected, stored, and analyzed large amounts of data; we made installations to a live system requiring system admins' intervention and had issues with third-party software. Despite the challenges, we obtained meaningful results that will benefit future research and development of monitoring parallel file system usage. Furthermore, these efforts take us closer to achieving the long-term goal at CSC is to build real-time monitoring, visualization, and reporting deployed on a live system that administrators can use to identify causes of slowdowns originating from parallel file system usage. Also, file system usage and I/O metrics could provide helpful information that can guide future procurements and configuration changes such that the investments and modifications improve the critical parts of the storage system.

In the thesis, we explored monitoring and analyzing the usage of the Lustre parallel file system in the CSC's Puhti cluster. We described the necessary details of a high-performance computer cluster, the Lustre parallel file system, and the configuration of the Puhti cluster at CSC for collecting fine-grained file system usage statistics with Lustre Jobstats. We also described how our monitoring system works by running a monitoring client on each Lustre server to collect data from Lustre Jobtats and send it to the ingest server, which inserts the data into a time series database. These descriptions will help us improve future versions of our monitoring system.

During the thesis, we uncovered issues with data quality caused by a bug in Lustre Jobstats and potentially by configuration or other issues in Puhti. Due to the issues, we had to modify the monitoring system and data analysis workflows to correct the issues to the extent possible and discard previous data that would have been useful in our analysis. Consequently, we lost valuable time and effort and only met some of the original goals for the thesis. Unreliable data made it infeasible to build an automated monitoring and analysis system to identify the causes of slowdowns in the cluster in real-time. Fortunately, we obtained enough data that was sufficiently reliable for explorative data analysis and visualization.

The results from the analysis demonstrate different low-level file system usage patterns and high-level views of total rates for all the monitored file system usage statistics during 24 hours of 2022-10-27. Furthermore, we demonstrate that we can identify users who cause large relative increases in I/O rates from our obtained data. These results increase our confidence that building an automated, real-time analysis and warning system can work once we solve data quality issues. To achieve these goals, we discussed ideas for future work, such as analyzing data as a stream instead of a batch, analyzing operations together, automatically identifying changes in I/O trends, and measuring the correlation between the changes and measured slowdowns.

References

[1] J. Dongarra, "Performance of various computers using standard linear equations software," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 3, pp. 22–44, 1992, doi: 10.1145/141868.141871.

[2] J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003, doi: 10.1002/cpe.728.

[3] J. Kunkel, J. Bent, J. Lofstead, and G. S. Markomanolis, "Establishing the IO-500 benchmark," *White Paper*, 2016.

[4] A. Paul *et al.*, "I/O load balancing for big data HPC applications," in *2017 IEEE international conference on big data*, 2017, pp. 233–242, doi: 10.1109/BigData.2017.8257931.

[5] A. Paul, B. Wang, N. Rutman, C. Spitz, and A. Butt, "Efficient metadata indexing for HPC storage systems," in *2020 20th IEEE/ACM international symposium on cluster, cloud and internet computing*, 2020, pp. 162–171, doi: 10.1109/CCGrid49817.2020.00-77.

[6] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: A proposal for an exascale storage system," in *SC'16: Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2016, pp. 585–596, doi: 10.1109/SC.2016.49.

[7] H. Tang *et al.*, "Toward scalable and asynchronous object-centric data management for HPC," in *2018 18th IEEE/ACM international symposium on cluster, cloud and grid computing*, 2018, pp. 113–122, doi: 10.1109/CCGRID.2018.00026.

[8] S. Liu *et al.*, "Practice guideline for heavy I/O workloads with lustre file systems on TACC supercomputers," in *Practice and experience in advanced research computing*, 2021, doi: 10.1145/3437359.3465570.

[9] L. Huang and S. Liu, "OOOPS: An innovative tool for IO workload management on supercomputers," in *2020 IEEE 26th international conference on parallel and distributed systems*, 2020, pp. 486–493, doi: 10.1109/ICPADS51040.2020.00069.

[10] G. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. Wright, "A year in the life of a parallel file system," in *SC18: International conference for high performance computing, networking, storage and analysis*, 2018, pp. 931–943, doi: 10.1109/SC.2018.00077.

[11] J. Lüttgau, S. Snyder, P. Carns, J. Wozniak, J. Kunkel, and T. Ludwig, "Toward understanding I/O behavior in HPC workflows," in *2018 IEEE/ACM 3rd international workshop on parallel data storage & data intensive scalable computing systems (PDSW-DISCS)*, 2018, pp. 64–75, doi: 10.1109/PDSW-DISCS.2018.00012.

[12] P. Carns, J. Kunkel, K. Mohror, and M. Schulz, "Understanding I/O behavior in scientific and data-intensive computing," in *Dagstuhl reports*, 2021, vol. 11.

[13] P. Braam, "The Lustre storage architecture," *CoRR*, 2019, doi: 10.48550/arxiv.1903.01955.

[14] *Lustre monitoring and statistics guide*. OpenSFS; EOFS, 2022 [Online]. Available: https://wiki.lustre.org/Lustre_Monitoring_and_Statistics_Guide

[15] J. Hanley and R. Mohr, "Lustre job stats metric aggregation at OLCF," 2016. [Online]. Available: https://lustre.ornl.gov/ecosystem-2016/documents/tutorials/Hanley-OLCF-JobStats.pdf

[16] D. Rodwell and P. Fitzhenry, "Fine-grained file system monitoring with lustre jobstat," 2014. [Online]. Available: https://www.opensfs.org/wp-content/uploads/2014/04/D3_S31_FineGrainedFileSystemMonitoringwithLustreJobstat.pdf

[17] A. Paul, O. Faaland, A. Moody, E. Gonsiorowski, K. Mohror, and A. Butt, "Understanding HPC application I/O behavior using system level statistics," in *2020 IEEE 27th international conference on high performance computing, data, and analytics*, 2020, pp. 202–211, doi: 10.1109/HiPC50609.2020.00034.

[18] *View for ClusterStor installation and configuration guide*. Cray Inc, 2022 [Online]. Available: https://support.hpe.com/hpesc/public/docDisplay?docId=a00114944en_us

[19] *DDN Insight*. DataDirect Networks [Online]. Available: https://www.ddn.com/products/storage-monitoring-ddn-insight/

[20] "TOP500 - operating system family / Linux," 2022. [Online]. Available: https://www.top500.org/statistics/details/osfam/1/

[21] L. Torvals *et al.*, *Linux kernel*. 1991 [Online]. Available: https://github.com/torvalds/linux

[22] M. Kerrisk, *The Linux programming interface: A Linux and UNIX system programming handbook*, 1st ed. USA: No Starch Press, 2010 [Online]. Available: https://man7.org/tlpi/

[23] M. Kerrisk, *The Linux man-pages project*. 2022 [Online]. Available: https://www.kernel.org/doc/man-pages/

[24] *Lustre* software release 2.x: Operations manual*. OpenSFS; EOFS, 2022 [Online]. Available: https://doc.lustre.org/lustre_manual.xhtml

[25] A. Yoo, M. Jette, and M. Grondona, "Slurm: Simple Linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*, 2003, pp. 44–60, doi: 10.1007/10968987_3.

[26] *Slurm workload manager*. SchedMD, 2022 [Online]. Available: https://slurm.schedmd.com/

[27] R. Griesemer, R. Pike, K. Thompson, *et al.*, *Go programming language*. 2009 [Online]. Available: https://go.dev/

[28] *InfluxDB*. InfluxData, Inc, 2013 [Online]. Available: https://www.influxdata.com/

[29] *TimescaleDB*. Timescale, Inc, 2018 [Online]. Available: https://www.timescale.com/

[30] *TimescaleDB documentation*. Timescale, Inc, 2022 [Online]. Available: https://docs.timescale.com/

[31] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017, doi: 10.1137/141000671.

[32] J. Bezanson, A. Edelman, S. Karpinski, V. Shah, *et al.*, *The Julia programming language*. 2012 [Online]. Available: https://julialang.org/

[33] B. Kamiński *et al.*, *DataFrames.jl - In-memory tabular data in Julia*. 2012 [Online]. Available: https://github.com/JuliaData/DataFrames.jl

[34] S. Christ, D. Schwabeneder, C. Rackauckas, M. K. Borregaard, and T. Breloff, "Plots.jl – a user extendable plotting API for the julia programming language," 2022, doi: 10.48550/arxiv.2204.08775.

[35] J. Heinen *et al.*, *GR Framework*. 1985 [Online]. Available: https://gr-framework.org/

[36] J. Le Dem *et al.*, *Apache Parquet*. The Apache Software Foundation, 2014 [Online]. Available: https://parquet.apache.org/

[37] "Fill jobid in an atomic way," 2022. [Online]. Available: https://jira.whamcloud.com/browse/LU-16251

A System calls

The next two examples present examples of performing file I/O using system calls. Flags and modes are constants that modify the behavior of a system call. The bitwise-or of two modes or flags means that both of them apply. Please note that these examples do not perform any error handling that proper programs should do. For in-depth documentation about system calls, we recommend the Linux Man Pages [23, Sec. 2].

```c
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>

int main()
{
    int n;  // number of bytes read
    int fd1, fd2;  // file descriptors
    const int size = 4096;  // buffer size
    char buffer[size];  // reserve memory for reading bytes
    // Read input file
    fd1 = open("input.txt", O_RDONLY);
    n = read(fd1, buffer, size);
    close(fd1);
    // Write output file
    fd2 = open("output.txt", O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
    write(fd2, buffer, n);
    close(fd2);
}
```

The first example program demonstrates opening and closing file descriptors, reading bytes from a file, and writing bytes to a file. It opens `input.txt` in read-only mode, reads at most `size` bytes to a buffer, and then creates and writes them into the `output.txt` file in write-only mode. The code performs the system calls `open`, `close`, `read`, and `write` with the flags `O_RDONLY`, `O_CREAT`, `O_WRONLY`, and modes `S_IRUSR` and `S_IWUSR`. Furthermore, if the file does not already exist, the code calls `mknod` to create it.

```c
#define _GNU_SOURCE
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>

int main()
{
    int fd;  // file descriptor
    fd = open("output.txt", O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
    write(fd, "hello hole world", 16);
    fallocate(fd, FALLOC_FL_PUNCH_HOLE|FALLOC_FL_KEEP_SIZE, 5, 5);
    close(fd);
}
```

The second example demonstrates a less common feature of punching a hole in a file, creating a sparse file. The hole appears null bytes when reading the file without taking any space on the disk. This feature is supported by certain Linux file systems such as ext4. The following code writes `hello hole world` to an `output.txt` file. It then deallocates bytes from 5 to 10 such that the file keeps its original size using `fallocate` with modes `FALLOC_FL_PUNCH_HOLE` and `FALLOC_FL_KEEP_HOLE`.

B Slurm job scripts

In Puhti, we can submit a job to the Slurm scheduler as a shell script via the `sbatch` command. We can specify the options as command line arguments as we invoke the command or in the script as comments. The script specifies job steps using the `srun` command. Next, we show three job script examples for different job sizes

---

```
#!/usr/bin/env bash
#SBATCH --job-name=<job-name>
#SBATCH --account=<project>
#SBATCH --partition=small
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=10G
srun <program>
```

The above script is an example of a small, sequential batch job with a single job step, that is, `srun` command.

```
#!/usr/bin/env bash
#SBATCH --job-name=<job-name>
#SBATCH --account=<project>
#SBATCH --partition=small
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=10G
#SBATCH --array=1-100
srun <program> $SLURM_ARRAY_TASK_ID
```

It is also common to run multiple similar sequential batch jobs independent from one another with slight variations, for example, in initial conditions. We can achieve that by turning it into an array job by adding the `array` argument with the desired range and accessing the array ID via an environment variable.

```
#!/usr/bin/env bash
#SBATCH --job-name=<job-name>
#SBATCH --account=<project>
#SBATCH --partition=large
#SBATCH --time=02:00:00
#SBATCH --nodes=2
#SBATCH --tasks-per-node=2
#SBATCH --cpus-per-task=20
#SBATCH --mem-per-cpu=2G
#SBATCH --gres=nvme:100
# 1. job step
srun --nodes 2 --ntasks 1 <program-1>
# 2. job step
srun <program-2>
# 3. job step
srun --nodes 1 --ntasks 2 <program-3> &
# 4. job step
srun --nodes 1 --ntasks 2 <program-4> &
# Wait for jobs 2. and 3. to complete
wait
```

Finally, the we can run large parallel batch jobs. The above script is an example of a large parallel batch job with four job steps. For example, it could perform the following steps. The first program runs on the first job step and could load data to the local disk. The second program runs on the second job step utilizing all given nodes, tasks, and CPUs and the majority of the given time. It is a large parallel program, such as a large, well-parallelizing simulation communicating via MPI. The third and fourth programs' job steps run parallel after the first step, utilizing all tasks and CPUs from a single node. These programs could be post-processing steps, for example, processing and backing up the simulation results.

C Computing and analyzing rates

We compute rates from counter values. A rate tells us how much the value changes on average during an interval. We referer to the values obtained from Jobstast as *observed values* in contrast to implicit zero values of counters outside the observation intervals, such as the initial counter. The observation time is called a *timestamp*. We identify each time series of counter values by a unique *identifier*. We refer to the identifiers of the observed counters as *observed identifiers*. The size of the set of observed identifiers tells us how many individual time series Jobstats is tracking at a given time. It is proportional to how much data we accumulate at each observation. We regard the observed identifiers as a subset of *all identifiers*, which is the set of all possible identifiers, depending on the chosen identifier scheme.

Let $\mathcal{K}$ denote the set of *all identifiers* and $t \in \mathbb{R}$ denote a *timestamp*. Then, we define $K(t) \subseteq \mathcal{K}$ as the set of *observed identifiers* at time t and $c_k(t) \in \mathbb{R}$ such that $c_k(t) \geq 0$ as the *observed counter value* at time t for observed identifier $k \in K(t)$.

We denote *counter value* as $v_k(t)$ for an arbitraty identifier $k \in \mathcal{K}$ at time t. Its value is the observed counter value if we observe the identifier k and zero if we do not. Formally, we have

$$v_k(t) = \begin{cases} c_k(t), & \text{if } k \in K(t) \\ 0, & \text{if } k \notin K(t) \end{cases}. \tag{C1}$$

We can sample the counter value over time as a streaming time series. Given previous timestamp t' and current timestamp t in the stream such that $t' < t$, we can calculate the *observation interval* as

$$\tau(t', t) = t - t'. \tag{C2}$$

Given an identifier $k \in K(t') \cup K(t)$, if the new counter value $v_k(t)$ is greater than or equal to the previous value $v_k(t')$, the previous value was incremented by $\delta_k(t', t)$ during the interval, that is, $v_k(t) = v_k(t') + \delta_k(t', t)$. Otherwise, the counter value has reset, and the previous counter value is implicitly zero, hence $v_k(t) = 0 + \delta_k(t', t)$. Combined, we can define the *counter increment* during the interval as

$$\delta_k(t', t) = \begin{cases} v_k(t) - v_k(t'), & \text{if } v_k(t) \geq v_k(t') \\ v_k(t), & \text{if } v_k(t) < v_k(t') \end{cases}. \tag{C3}$$

Then, we can calculate the *rate of change* during the interval as

$$r_k(t', t) = \frac{\delta_k(t', t)}{\tau(t', t)}. \tag{C4}$$

Note that the rate of change is always non-negative given $t > t'$, since we have $\tau(t', t) > 0$ and $\delta_k(t', t) \geq 0$, which implies $r_k(t', t) \geq 0$.

Generally, we can represent the rate of change as a step function over continuous time t with identifier $k \in K$ given a sampling $(t_1, t_2, ..., t_n)$ where $t_1 < t_2 < ... < t_n$ and $n \in \mathbb{N}$ and $K = K(t_1) \cup K(t_2) \cup ... \cup K(t_n)$ as

$$r_k(t) = \begin{cases} r_k(t_{i-1}, t_i), & \text{if } t_{i-1} < t \leq t_i, \quad \forall i \in \{2, ..., n\} \\ 0 & \text{otherwise} \end{cases}. \tag{C5}$$

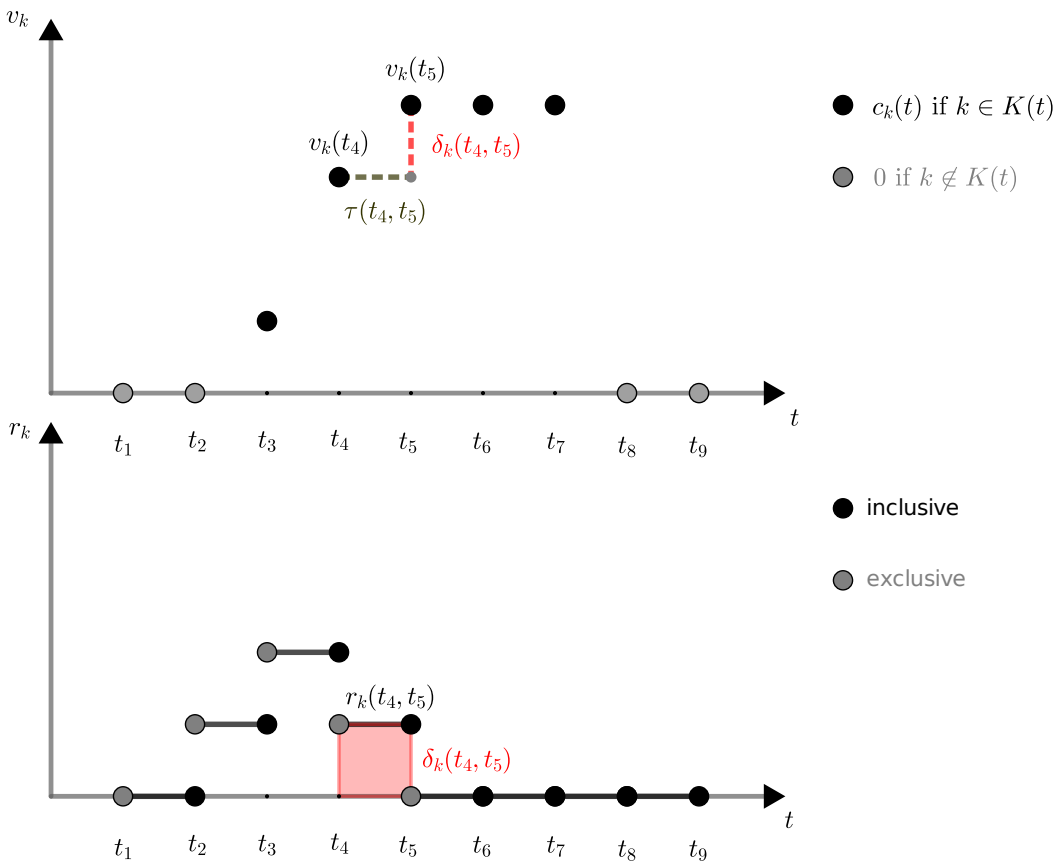

Figure C1: The upper graph shows a sampling of a counter values $v_k(t)$ from Equation (C1) with timestamps $t_1, t_2, ..., t_9$. The lower graph shows the computed rate $r_k(t)$ as described in Equation (C5). The graphs also show the observation interval $\tau(t_4, t_5)$ from Equation (C2), counter increment $\delta_k(t_4, t_5)$ from Equation (C3), and individual rate $r_k(t_4, t_5)$ from Equation (C4) The red box demonstrates the relationship $\delta_k(t_4, t_5) = r_k(t_4, t_5) \cdot \tau(t_4, t_5)$, which recovers the counter increment with a definitive integral, seen in Equation (C6).

We can recover the counter increments from the step function using a definite integral

$$\delta_k(t_{i-1}, t_i) = \int_{t_{i-1}}^{t_i} r_k(t)\, dt = r_k(t_{i-1}, t_i) \cdot \tau(t_{i-1}, t_i), \quad \forall i \in \{2, ..., n\}. \tag{C6}$$

Using the property (C6), we can transform a step function $r_k(t)$ into a step function $r'_k(t)$ with timestamps $t'_1, t'_2, ..., t'_m$ where $t'_1 < t'_2 < ... < t'_m$ and $m \in \mathbb{N}$ such that it preserves the change in counter values in the new intervals by first setting as

$$\delta'_k(t'_{i-1}, t'_i) = \int_{t'_{i-1}}^{t'_i} r'_k(t)\, dt = \int_{t'_{i-1}}^{t'_i} r_k(t)\, dt, \quad \forall i \in \{2, ..., m\}. \tag{C7}$$

Then, by computing the rate of change using (C4). This transformation is helpful if we have multiple step functions with steps as different timestamps, and we need to convert the steps to happen at identical timestamps. In practice, we can avoid the transformation by querying the counters simultaneously.

We define the sum of rates of change over identifiers $K \subseteq \mathcal{K}$ as

$$r_K(t) = \sum_{k \in K} r_k(t). \tag{C8}$$

If all step function r_k where $k \in K$ have the steps at the identical timestamp, the summation is easy. We sum all the values at each timestamp. Otherwise, we have to resort to a more complex algorithm whose description is out of the scope of this thesis. In practice, we transform the timestamps of all rates to identical timestamps using (C7) before summation if necessary.

We define a function that indicates if the logarithmic value of $x \in \mathbb{R}$ with *base* $b \in \mathbb{N}$ where $b > 1$ belongs to the *bucket* $y \in \mathbb{Z}$ as

$$\mathbf{1}_{b,y}(x) = \begin{cases} 1, & \text{if } b^y \leq x < b^{y+1} \\ 0, & \text{otherwise} \end{cases}. \tag{C9}$$

Let R be a set of step functions. Then, we define the density over time as a counting function as

$$z_{b,y}(t) = \sum_{r_k \in R} \mathbf{1}_{b,y}(r_k(t)). \tag{C10}$$

The base parameter determines the *resolution* of the bucketing.

In practice, we can use the logarithmic floor function to compute the bucket y of a value $x > 0$, because the relationship $\lfloor \log_b(x) \rfloor = y$ is logically equivalent to $b^y \leq x < b^{y+1}$.